Caching Techniques for Streaming Multimedia over the Internet

Markus Hofmann[1], T.S. Eugene Ng[2], Katherine Guo[1], Sanjoy Paul[1], Hui Zhang[2]

[1]Bell Laboratories
101 Crawfords Corner Road
Holmdel, NJ 07733, USA
(hofmann, kguo, sanjoy)@bell-labs.com

[2]Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
(eugeneng, hzhang)@cs.cmu.edu

Abstract—**Existing solutions for streaming multimedia in the Internet do not scale in terms of object size and number of supported streams. Using separate unicast streams, for example, will overload both network and servers. While caching is the standard technique for improving scalability, existing caching schemes do not support streaming media well. In this paper, we propose a complete solution for caching multimedia streams in the Internet by extending existing techniques and proposing new techniques to support streaming media. These include segmentation of streaming objects, dynamic caching, and self-organizing cooperative caching. We consider these techniques in an integrated fashion. We implemented a complete caching architecture called SOCCER using the network simulator ns-2 and evaluate the effectiveness of each proposed technique and compare them to alternative caching solutions.**

Keywords—**Streaming Media, Caching, Proxy.**

I. Introduction

Internet and World-Wide-Web are becoming the ubiquitous infrastructure for distributing all kinds of data and services, including continuous streaming data such as video and audio. A significant increase of commercial products for playback of stored video and audio over the Internet [1], [2] has occurred over the past several years, as well as a proliferation of server sites that support audio/video contents. However, existing solutions for streaming multimedia are not efficient because they use a separate unicast stream for each request, thus they require a stream to travel from the server to the client across the Internet for every request. From the content provider's point of view, server load increases linearly with the number of receivers. From the receiver's point of view, she must endure high start-up latency and unpredictable playback quality due to network congestion. From the ISP's point of view, streaming multimedia under such an architecture poses serious network congestion problems.

Multicast and caching are two common techniques for enhancing the scalability of general information dissemination systems. However, neither of them can be directly applied to support streaming media playback over the Web. In multicast, receivers are assumed to be homogeneous and synchronous. In reality, receivers are generally heterogeneous and asynchronous. This problem can be solved by batching multiple requests into one multicast session, thus, reducing server load and network load. Unfortunately, this solution does increase the average start-up latency.

Caching of web objects for improving end-to-end latency and for reducing network load has been studied extensively starting with CERN httpd [3], followed by improvements in *hierarchical caching* and *co-operative caching* under the Harvest project [4], [5] and the Squid project [6], respectively. Surprisingly, there has been very little work to extend cache systems to support streaming media. Existing caching schemes are not designed for and do not take advantage of streaming characteristics. For example, video objects are usually too large to be cached in their entirety. A single, two hour long MPEG movie, for instance, requires about 1.4 Gbytes of disk space. Given a finite buffer space, only a few streams could be stored at a cache, thus, decreasing the hit probability and the efficiency of the caching system. In addition, transmission of streaming objects needs to be rate regulated[1] and these timing constraints need to be considered in the design of a caching system for streaming media.

In this paper, we explore and assess three techniques to enhance caching systems to better support streaming media over the Internet, namely *segmentation of streaming objects*, *dynamic caching*, and *self-organizing cooperative caching*. We consider these techniques in an integrated fashion and define a unified streaming architecture called *Self-Organizing Cooperative Caching Architecture (SOCCER)* that can realize each of these techniques and allow

[1]In this paper, we assume the transmission rate of streaming objects to be a constant bit rate (CBR). However, the described techniques can be extended to also support streaming objects with variable bit rate (VBR).

us to use any combination of them. The architecture can use both caching and multicast techniques, and it takes advantage of unique properties of streaming media. The key components of the architecture are so-called *helper* machines, which are caching and streaming agents inside the network, communicating via a novel scalable state distribution protocol and forming dynamic meshes for forwarding streaming data.

The rest of the paper is organized as follows. Section II discusses techniques for extending caching systems to better support streaming media. Section III describes the design of our unified streaming architecture. To evaluate various streaming techniques, we have implemented the architecture in the ns-2 simulator and built a prototype implementation on the Unix platform. Section IV presents the simulation results. Related work is discussed in Section V and we conclude the paper in Section VI

II. Streaming Extensions for Caching Systems

In this section, we discuss techniques to better support streaming media in caching systems.

A. Segmentation of Streaming Objects and Smart Segment Replacement

Rather than caching entire streaming objects in an all-or-nothing fashion, we propose a more promising way by dividing the objects into smaller segments for caching and replacement purposes. Suppose the minimal allocation unit of a cache disk block is size S, then we can let the streaming object segment size be any multiple of S. For the rest of this discussion, we simply assume a segment size of S. By doing so, segments of a streaming object can be cached and replaced independently and therefore contention for disk space is greatly reduced and disk space can be most efficiently used to cache popular portions of large streaming objects.

A drawback of such independent segment caching is that when a streaming request (e.g. playback from time 0 sec to 354 sec) arrives at a cache, the request will most likely result in a partial cache hit in that only certain parts of the requested data are in the cache. Therefore, satisfying such requests requires searching for the missing segments in other caches. This not only increases signaling cost, but also increases the probability of losing synchronization. To address this issue, we need to control the number of missing gaps. One way to accomplish this is to increase the segment size. However, this also increases the contention for disk space, and in the extreme case, this degenerates into caching all or nothing. This points to a need for a large logical unit for caching, while still retains a fine granularity for disk allocation and replacement. We propose a logical unit for caching called *chunk*. Figure 1 illustrates the various caching units. A chunk is simply a number of contiguous segments within an object. Thus, starting from the beginning of an object, every k segments form a chunk. Each chunk is then cached independently using a *prefix caching* allocation and replacement policy. That is,

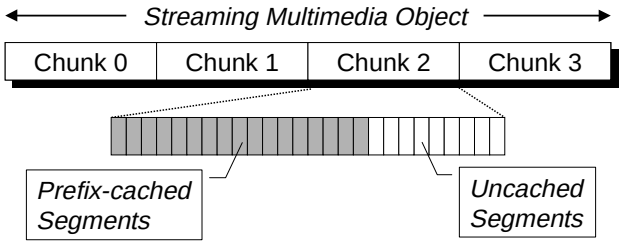

Fig. 1. Segmentation and chunking of streaming objects

- the basic unit of caching and cache replacement is a segment,
- segments allocated for a chunk always form a prefix of the chunk,
- when any segment within a chunk is being accessed, no segment within the chunk can be ejected from the cache,
- when any segment within a chunk is chosen by the replacement algorithm, the last segment of the prefix-cached segments is always the actual ejected victim.

By varying the chunk size, we achieve a trade-off between the maximum number of gaps and the segment replacement flexibility. In the extreme case, this degenerates into performing prefix caching of the entire object. In practice, finding the missing gaps in the system might require some form of intelligent prefetching to meet the timing requirements of streaming.

B. Dynamic Caching

As in classical web caching, the contents of cached multimedia data segments do not change over time and their ejection is controlled by a cache replacement policy. We refer to this as *static caching*. However, the streaming nature of multimedia offers new opportunities for further bandwidth savings than performing static caching alone. The key observation is that playback requests for streaming media are related by *temporal distances*. Normally, two playback requests require two separate data streams. However, if we can hide the temporal distance between the requests, then only one data stream is needed to serve both requests. This observation was exploited in [7], [8] for video servers in video-on-demand systems. We generalize this technique for caching systems and name it *dynamic caching*. Figure 2 demonstrates the basic dynamic caching technique. Receiver R_1 has requested a certain streaming object from server S at time t_1. At a later time

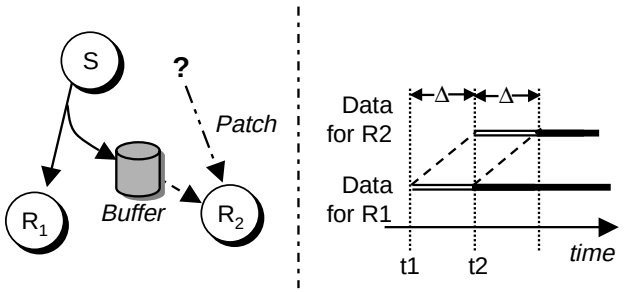

Fig. 2. Example illustrating dynamic caching

t_2, R_2 requests for the same object. Let $\Delta = t_2 - t_1$ be the temporal distance between the requests. At t_2, the first Δ seconds of the stream have already been received by R_1. However, notice that all subsequent data being streamed to R_1 will also be needed to satisfy R_2's request. Thus, by allocating a *ring buffer* in the network to cache a moving window of Δ seconds (starting at playback time t_2) of the data stream for R_1, the same data stream can be shared to satisfy R_2's request Δ seconds later. The ring buffer has essentially hidden the temporal distance between the two requests. Of course, R_2 will have to obtain the initial Δ seconds of missing data (called a *patch*) either from S or from some cache in the network. This is known as *patching*. These basic techniques have been proposed in [8] in the context of receivers buffering data *locally* in order to join an in-progress multicast streaming session. We generalize the techniques to allow any network cache to perform dynamic caching for any receivers or any other network caches, and to allow a stream to flow through any number of dynamic caches in the network. In other words, dynamic caches form a stream distribution mesh in the network. Of course, dynamic caching also enables IP multicast delivery of streams in our caching system exactly like that proposed in [8].

Because dynamic caching helps to stream data to multiple destinations efficiently, it is a complementary technique to static caching. It is most useful when data are being streamed to caches or between caches. This happens when the static caches are being filled, or when caches are highly loaded. It is remarkable that a small ring buffer can be enough to deliver a complete streaming object however large it might be. Nevertheless, when presented with a choice between accessing a dynamic cache versus a static cache, there are several interesting trade-offs. First, the moving window nature of a dynamic cache ensures the delivery of the entire streaming object. On the other hand, static cache usually holds only portions of a streaming object, requiring more complex operations in order to retrieve the entire streaming object from within the system. Second, using dynamic caches always requires a feeding stream into the dynamic cache. This possibly increases network load compared to accessing static caches only.

C. Self-Organizing Cooperative Caching

Cooperation among distributed caching systems requires the ability to identify other caches and to learn about their current state. Optimally, a cache would always know the current state of all the other caches. This would allow it to choose the most suitable cache to cooperate with at any time. However, it is impractical to keep a consistent and up-to-date view of all distributed caches.

Existing caching systems solve this problem by statically defining the neighbors of a cache using a configuration file [6]. This approach limits the ability of dynamically adapting to changes in network load, system load, and cache contents. Furthermore, it does not support instant identification of active data streams between caches, which is prerequisite to sharing multicast streams for multiple user requests.

For these reasons, we propose a mechanism for scalable state distribution that enables caches to learn about the state of remote caching systems. Using this mechanism, loosely coupled *cache meshes* can be formed to forward a streaming object to various receivers. We call this *self-organizing cooperative caching*. The details of scalable state distribution and self-organizing cooperative caching are given in the following section.

III. Unified Caching Architecture for Streaming Media

In order to explore and assess streaming extensions for caching systems, we propose a unified architecture called *Self-Organizing Cooperative Caching Architecture (SOCCER)* that embodies all the techniques discussed in Section II. Its core elements are the so-called *helper* machines, which are caching and data forwarding agents inside the network. Helpers serve requests for streaming objects by using cached (static or dynamic) data as much as possible. Close cooperation among helpers is enabled by scalable state distribution. A receiver interested in getting a certain streaming object simply sends its request to the origin server. The request will be redirected to the receiver's *proxy helper* either transparently by a layer-4 switch or by configuring the proxy in the client's software. On receiving the request, the proxy helper identifies locally available caches and other helpers that can serve the request through static and dynamic caching. If no appropriate helper is found, the request is forwarded directly to the origin server. Section III-A illustrates the interaction between static and dynamic caching. The mechanisms and criteria for enabling helper cooperation are explained in Section III-B and III-C.

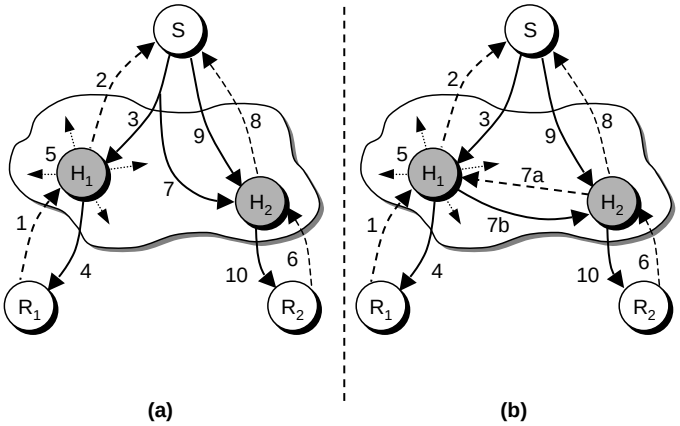

Fig. 3. Example illustrating the unified architecture (a) using multicast (b) using unicast

A. Interaction between Static and Dynamic Caching

SOCCER makes use of both static caching and dynamic caching. The interaction between static and dynamic caching is illustrated in the example shown in Figure 3. There are two receivers R_1 and R_2, a video server S, and two helpers H_1 and H_2. No data is cached at any helper initially. Figure 3a illustrates an example using multicast, while Figure 3b shows an example for unicast only.

Receiver R_1 requests streaming object O at time t_1. The request is either directly addressed or transparently redirected to the proxy helper H_1 (step "1" in Figure 3a). Because the requested data is not available from H_1's local cache nor from any other remote helper, a request is sent to server S (step "2"). On receiving the request, server S initiates a new data stream for object O (step "3"). The new data stream can either be unicast or multicast. For now, we assume it is multicast.

At time t_1', H_1 receives the data stream and starts caching incoming data in an infinitesimally small, default-sized ring buffer. It also starts caching the data segments in static cache (as discussed in Section II). Simultaneously, it streams the data to R_1 (step "4"). H_1 also begins advertising its ability to deliver segments of object O (step "5").

At some later time, receiver R_2 issues a request for streaming object O. The request is redirected and reaches proxy helper H_2 at time t_2 (step "6"). Because of state distribution through advertisements, H_2 knows the state of helper H_1 and the multicast stream transmitting the desired object. Interesting is that H_2 has several options, depending on whether it is preferable to exploit static or dynamic caches, or to make use of unicast or multicast techniques:

- *Unicast from server*: Helper H_2 can contact the server and request the entire streaming object just like what H_1 did. Obviously, this does not require any additional buffer space, but it increases server and network load.
- *Multicast from server*: Dynamic caching enables H_2 to exploit the ongoing multicast transmission. To join the multicast group, H_2 allocates a ring buffer that can hold $\Delta + \epsilon$ seconds of data, where $\Delta = t_2 - t_1'$. The extra ϵ seconds is added to absorb network delays and other random factors. H_2 starts buffering received multicast data (step "7") and, at the same time, requests one or more patches to get the first Δ seconds of the object. H_2 might request a patch either from the static cache of H_1 or directly from the sender (step "8"). Upon receiving the patch (step "9"), H_2 forwards it to R_2 (step "10"). As soon as H_2 has forwarded all patching data, it starts forwarding data from the ring buffer (step "10").
- *Unicast from helper's dynamic cache*: From state distribution, H_2 knows about the content of H_1's static and dynamic caches. H_2 might decide to exploit the dynamic cache of H_1. To do this, H_2 again allocates a ring buffer of size $\Delta + \epsilon$ seconds and sends a request to H_1 (step "7a" in Figure 3b). On receiving the request, H_1 starts forwarding the data to H_2 (step "7b"). Like before, H_2 now has to request a patch either from the static cache of H_1 or directly from the sender. The rest of the operations proceed as before (step "8", "9" and "10").
- *Unicast from helper's static cache*: It is also possible for H_2 to serve R_2's request solely by accessing static caches in H_1 and by contacting the server in case it becomes necessary.

B. Mechanisms for Scalable State Distribution

Self-organizing helper cooperation is enabled by scalable state distribution. Helpers periodically send advertise messages to a well-known multicast address indicating their ability to serve requests for certain media objects. State information is then used to select the best helpers to cooperate with.

A mechanism for scalable state distribution strives to balance two conflicting goals: network load should be kept as low as possible, and the freshness of state information should be as high as possible. Obviously, frequent sending of global advertisements improves freshness but increases network load. Notice that a helper would prefer getting support from a nearby helper rather than from a far away helper. Thus, to reduce the overhead caused by global advertisements, a good trade-off is to have a helper receive advertisements from other nearby helpers more frequently than from helpers far away.

These observations motivated our design of the *Expanding Ring Advertisement (ERA)*, which was originally de-

Interval No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
TTL	15	31	15	63	15	31	15	127	15	31	15	63	15	31	15	254	15	31

TABLE I
TTL VALUES USED TO SEND ADVERTISEMENTS

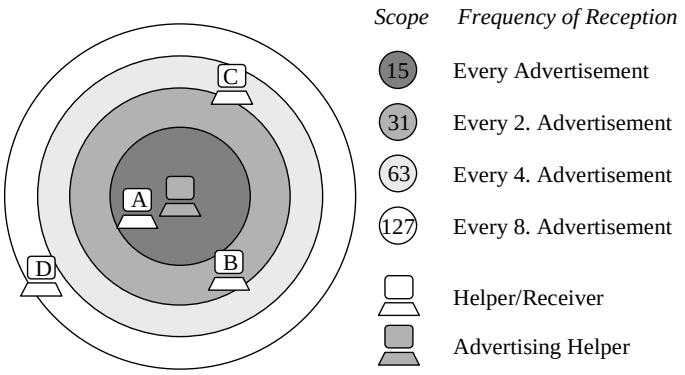

Fig. 4. Different scoping areas

veloped in the context of reliable multicast protocols [9]. The ERA algorithm makes use of TTL-based scope restriction. However, it could easily be modified to use administratively scoped multicast addresses or any other scoping mechanism. According to the ERA algorithm, helpers send their advertisements with dynamically changing TTL values given in Table I. The given TTL values are defined according to the scope regions in the current MBone. They can easily be adjusted to conform to specific network infrastructures. In the given scheme, the first message is sent with a scope of 15 (*local*), the second one with a scope of 31 (*regional*), etc. The effect of this scheme is illustrated in Figure 4. Listening helpers within a scope of 15 (e.g., helper A) will get every advertise message (assuming there are no packet losses). Those who have distance between 16 and 31 (e.g., helper B) will receive every other advertisement. And every 16^{th} advertise message will be distributed worldwide. This ensures that the frequency of advertise messages exponentially decreases with increasing scope. Therefore, the scheme reduces network load while allowing short reaction time within a local scope. However, it still ensures global visibility of helpers over a larger time scale. By observing sequence numbers, the advertisements can also be used to estimate the number of hops between helpers without requiring access to the header of received IP packets.

C. Helper Selection

Upon receiving a request for a streaming object, a helper has to decide where to get the requested data. Data can be retrieved from its own local cache, directly from the server, or from any static or dynamic cache of other helpers. The decision making process is called *helper selection*. In general, a helper may get portions of the requested data from multiple sources. It is the goal of the helper selection algorithm to find a near optimal sequence of static and/or dynamic cache accesses in order to serve the request. We propose a helper selection algorithm that uses a step-wise algorithm to find one cache at a time and finishes streaming data from a selected cache before determining the next cache in sequence, thus, achieving a step-wise local optimum.

C.1 Algorithm for Helper Selection

During each iteration of our step-wise algorithm, a helper looks for the best cache (either dynamic or static) to use. Once it chooses a static cache, it gets as much data as there is in the selected static cache up to the start of the next available local cache or up to the end of the request. Once it chooses a dynamic cache, it gets data from it up to the end of the clip. The algorithm is outlined in Figure 5. During each iteration, using a cost function (discussed in the next section), the algorithm finds the lowest cost cache currently in the helpers to serve the request. Selection is restricted to static caches[2] only if the $staticOnly$ flag is set. To restrict the number of switches between static caches, the algorithm requires a static cache to have at least M segments of data after $tStart$ to be considered. If the cache found is static, the helper gets as much data as possible from this static cache, up to the point where its own local static cache can be used, and advances $tStart$ for the next iteration to the end of the current request. If the cache found is dynamic, the helper sends a request to get data up to $tEnd$ from this dynamic cache. At the same time, it continues searching to get any required patches from one or more static caches. Using a dynamic cache generally involves prefetching of future data because the content of a dynamic cache changes over time. While prefetching data, the selected dynamic cache might become suboptimal. However, since the helper has already started to prefetch data, it keeps using the previously selected dynamic cache.

C.2 Cost Function for Helper Selection

Making use of a static or a dynamic cache is associated with many cost factors. For example, accessing a cache might cause additional network load and increase the processing load at the cache, using a dynamic cache

[2]The origin server is considered a static cache as well.

```
tStart = start playback time;
tEnd = end playback time;
done = false;
staticOnly = false;

while not(done) do
   if (staticOnly) then
      (H,C) = (helper,static cache) of min
      cost;
   else
      (H,C) = (helper,any cache) of min cost;
   fi
   if (isStatic(C)) then
      send request Rs to H for data from tStart
      to min(tEnd,endOf(C),start of next
         available local cache)
      if (endOf(Rs) == tEnd) then
         done = true;
      else
         tStart = endOf(Rs);
         sleep for the playback duration of Rs;
      fi
   else if (isDynamic(C)) then
      send request Rd to H for data from
         startOf(C) to tEnd;
      if (tStart == startOf(C)) then
         done = true;
      else
         tEnd = startOf(C);
         staticOnly = true;
      fi
   fi
od
```

Fig. 5. Helper selection algorithm

requires buffer space to absorb the temporal distance, etc. To keep complexity low, it is reasonable to define a subset of useful indices and to specify a good heuristic to find appropriate caches. Content providers, network service providers, and content consumers generally have different, sometimes conflicting optimization goals. We define one possible cost function that tries to find a balance between network and server/helper load. Before describing the function, we first specify two important cost factors that we consider – network load and server/helper load.

Network Load: The network distance N between two helpers H_i and H_j is related to the number of hops on the path from H_i to H_j. Network distance is used as an approximation for the network load associated with data transmission from one host to another. In SOCCER, helpers implicitly use the ERA algorithm to estimate network distance among themselves and the sender. They classify remote helpers based on the observed scoping level and assign distance values to each of the classes according to Table II. In addition, to approximate network load for receiving multicast data, the network distance between helper H_i and multicast group G is defined to be the network distance to the closest member of group G.

class	ttl (scope)	network distance ($d(H_i, H_j)$)
local	0 - 15	1
regional	16 - 31	2
national	32 - 63	3
global	64 - 255	4

TABLE II
CLASSES OF NETWORK DISTANCE

Server/Helper Load: It is desirable to evenly distribute load among the helper machines and to avoid overloading popular helpers. The total number of incoming and outgoing streams is a good indicator of the current processing load at a server/helper and is, therefore, included in the advertise messages. Each server/helper has a maximum number of streams it can serve concurrently. When the number of streams served is far less than this processing capacity limit, serving one more stream has little impact on performance. However, when the server/helper is operating close to its capacity, serving one more stream will degrade the performance significantly. As a result, the cost associated with server/helper load should increase mildly for low load and increase sharply for high load. Therefore, we define it as follows: If the cache is local or the resulting stream is multicast, the load related cost L for requesting a stream from helper H (or server S) is defined as one. If the resulting stream is unicast, it is defined as

$$L = \frac{maxLoad}{maxLoad - currentLoad} \quad ,$$

where $currentLoad$ is the number of streams at helper H and $maxLoad$ is the maximum allowable number of streams at H. Notice that the cost associated with server/helper load is minimized for both local cache (static or dynamic) and multicast streams, because these scenarios do not incur any additional load at other helpers or the server.

Calculating the Normalized Cost: We define the normalized cost function for using a cache as the cost of getting a single segment from that cache. Using the two cost factors introduced above, the normalized cost is defined as $C = N \times L$, where N is the network cost and L is the cost associated with server/helper load. This metric is suitable for static caches. On the other hand, since using a dynamic cache involves prefetching data that are potentially not needed, care must be taken to discourage the use of a dynamic cache of a very large temporal distance. For example, a helper might have prefetched r segments when the user aborts the transmission such that only x segments

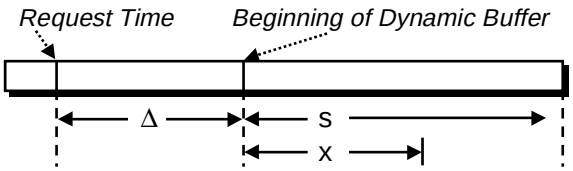

Fig. 6. Illustration of variables used in normalized cost

are actually used. In this case, the normalized cost for getting those x segments is increased by a overhead factor of

$$c = \frac{r}{x} \tag{1}$$

In general, the larger the temporal distance, the larger the potential waste. Figure 6 illustrates how we estimate this potential waste when a dynamic cache is selected. The notations used are as follows:

- Δ: The temporal distance in segments between the requesting and the serving helper is denoted by Δ, where $0 \leq \Delta$.
- s: The number of segments from beginning of dynamic cache up to the end of the streaming object is denoted by s, where $0 < s$.
- x: The number of segments that are actually used from the dynamic cache is denoted by x, where $0 < x \leq s$.

When a helper has received all the patching data and starts using data from a dynamic cache, it has already prefetched Δ segments. If x segments are used, the helper will have prefetched x additional segments, until it reaches the end of the object. Therefore, the number of prefetched segments r after x segments are used is given by $r = \min(\Delta + x, s)$. Thus, it follows that

$$c = \begin{cases} \frac{s}{x} & , \quad \text{if } (s - \Delta) < x \\ \frac{\Delta + x}{x} & , \quad \text{if } (s - \Delta) \geq x \end{cases} \tag{2}$$

$x > 0$ is assumed. If the considered cache is static or the temporal distance Δ is zero, the overhead reduces to 1. At the time the cost calculation is done, a helper does not know whether or when a user might terminate transmission. Therefore, a helper needs to estimate the value of x. Assuming that the playback lengths are uniformly distributed, then playback will on average continue for another $s/2$ segments. By substituting $x = s/2$ in equation (2), we estimate the overhead factor as

$$c = \begin{cases} 2 & , \quad \text{if } \frac{s}{2} \leq \Delta \\ (2 \cdot \frac{\Delta}{s}) + 1 & , \quad \text{if } \Delta < \frac{s}{2} \end{cases} \tag{3}$$

Finally, the normalized cost function used by our helper selection algorithm is

$$C = \begin{cases} 2 \cdot N \cdot L & , \quad \text{if } \frac{s}{2} \leq \Delta \\ ((2 \cdot \frac{\Delta}{s}) + 1) \cdot N \cdot L & , \quad \text{if } \Delta < \frac{s}{2} \end{cases} \tag{4}$$

A different request length distribution will give a different value of x, resulting in a different normalized cost function. It is important to note that, this cost function is minimized for local static cache, where Δ is zero by definition and N and L are one. As a result, whenever local static cache is available, the algorithm always makes use of it to the fullest.

IV. Simulation

We use simulations conducted in ns-2 version 2.1b2 [10] to evaluate the effectiveness of the various techniques for streaming media. Helpers can be configured to perform proxy caching, hierarchical caching, or dynamic cooperative caching, using any combination of the techniques described in Sections II and III. For state distribution, we use the ns-2 centralized multicast feature with sender based trees. This option eliminates the periodic flooding and pruning messages of DVMRP [11]. Multicast routers in ns-2 always decrement the TTL value of packets being forwarded by one. Therefore, the ERA algorithm uses TTL values of 1, 2, 3, and 20 (to ensure delivery to all helpers) in the simulation instead of 15, 31, 63, and 127, as described in Section III-B.

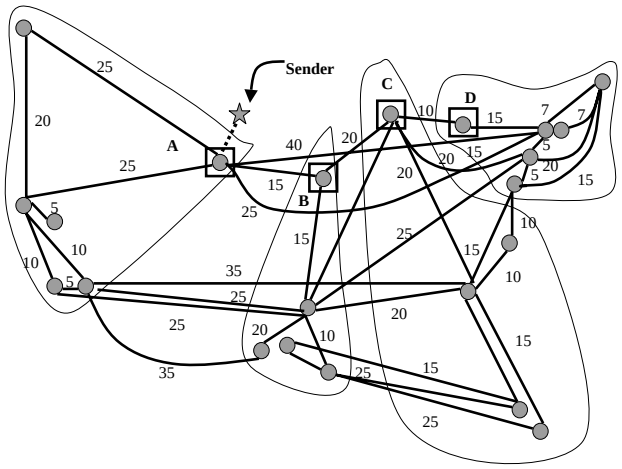

Fig. 7. MCI backbone topology and the static hierarchy built on top of it. The numbers on the link represent the latency in milliseconds. A, B, C, and D are parent helpers in the hierarchy.

We perform our simulation on the MCI backbone network[3] as shown in Figure 7. Link capacities are irrelevant since we do not consider network congestion. A helper and a stub-router are attached to each backbone router. Each stub-router simulates a subnet of receivers. The latency between each backbone router and its corresponding stub-router is 2 ms. The sender is attached to router A through six additional hops of latency 10 ms each.

[3]Topology obtained from *www.caida.org* in October 1998.

Parameter	Default Value
Simulation Time	10 hours
Average Request Inter-Arrival Time Per Subnet	4 minutes
Stream Segment Size	1.67 MB
Stream Chunk Size	10 MB
Helper Static Cache Space	5 GB
Helper Dynamic Cache Space	64 MB
Static Caching	On
Dynamic Caching	Off
Advertisement Timer	60 seconds
Cost Function Parameter *maxLoad*	100
Minimum Static Cache Solution Size	1.67 MB
Cache Replacement Policy	LFU
Streaming Method	Unicast

TABLE III
DEFAULT VALUES OF PARAMETERS

For hierarchical caching, we construct a two-level hierarchy, where level 1 composes of nodes A, B, C, and D. The corresponding children level 2 nodes are assigned based on the grouping shown in Figure 7. Two versions of the hierarchy are used. Hierarchy 1 uses 5 GB as the helper static cache space for all caches. Hierarchy 2 uses 13 GB and 3 GB for level 1 and level 2 caches respectively (total cache space is the same as in Hierarchy 1).

The sender contains 100 100-minute long streaming media objects of 1 GB each. A media stream is a constant bit rate UDP stream with 1 KB packet size and 1.33 Mbps playback rate. Each subnet of receivers follows a Poisson request arrival process and the requested object follows the Zipf [12] distribution. All requests start from the beginning of an object, but vary in the ending time. We use a bimodal distribution that gives a 40%, 20%, and 40% chance of playing back 0 to 20 minutes, 20 to 80 minutes, and 80 to 100 minutes respectively.

Table III specifies a set of default simulation parameters. Whenever applicable, their default values are used unless otherwise noted.

We use two performance metrics to evaluate various techniques, they are network load and sender load. Network load is illustrated as the cumulative number of bytes of all types of packets transmitted on all links versus time, whereas sender load is illustrated as the total number of outgoing streams at the sender versus time. In general, good performance is indicated by low network and sender load.

A. Simulation Results

To illustrate the benefits of various multimedia caching techniques, a set of simulation results are presented in Figure 8. First, we consider the benefit of segmenting a large multimedia object by varying the segment size (Figure 8 (a) & (b)). Recall that a segment is the minimum unit of data cached. When no segmentation is performed, multimedia objects are cached in all-or-nothing fashion. As shown in the figure, no segmentation leads to the highest network load and server load due to high contention for disk space. As the segment size decreases, contention for disk space is reduced and thus caches perform more effectively leading to lower overall load. Depending on the request arrival frequency, continuing reduction in segment size ceases to provide significant additional benefit. Since reducing the segment size increases the disk management overhead, the segment size should be set based on the expected system load.

Next we consider dynamic caching (Figure 8 (c) & (d)). As noted before, dynamic caching is most useful when data are being streamed across the network when caches are being filled or the caches are heavily loaded. We simulate a highly loaded system by reducing the average request inter-arrival time per subnet to one second. Simulation time is reduced to 600 seconds to limit execution time, but as a result the data does not show the steady state of the system. Static caching is turned off while dynamic caching is turned on, and we vary the size of the dynamic cache. Two points worth noting in this experiment. First, as expected, network load reduces as the dynamic cache size increases. Second, by using dynamic caching, the sender load is decreased only by a small amount because of the extra patching streams.

The benefit of self-organizing cooperative caching is shown in Figure 8 (e) & (f). We compare our proposed scheme against proxy caching and hierarchical caching (using two cache size configurations). There are several points to note. First, we see that the performance of hierarchical caching is highly dependent on the cache size configuration. Hierarchy 1, with 5 GB of space at each helper, gives similar network load as proxy caching, while reducing the sender load slightly. Hierarchy 2, with 13 GB caches at level 1 and 5 GB caches at level 2, provides much better performance. Second, self-organizing cooperative caching yields the best performance overall. It reduces network load by 44% compared to proxy caching or hierarchical caching in variant 1, and by 35% compared to hierarchical caching in variant 2. Third, the network load reported already accounts for the state distribution data packets, which contribute less than 0.03% to the total data traffic. In reality, compression techniques can be used to further reduce the overhead. We have also experimented with other advertisement timer values from 240 seconds to 20 seconds and observed that network load (including state distribution overhead) reduces slightly as advertisement messages are sent more frequently because the information used by helpers is more up-to-date. These results

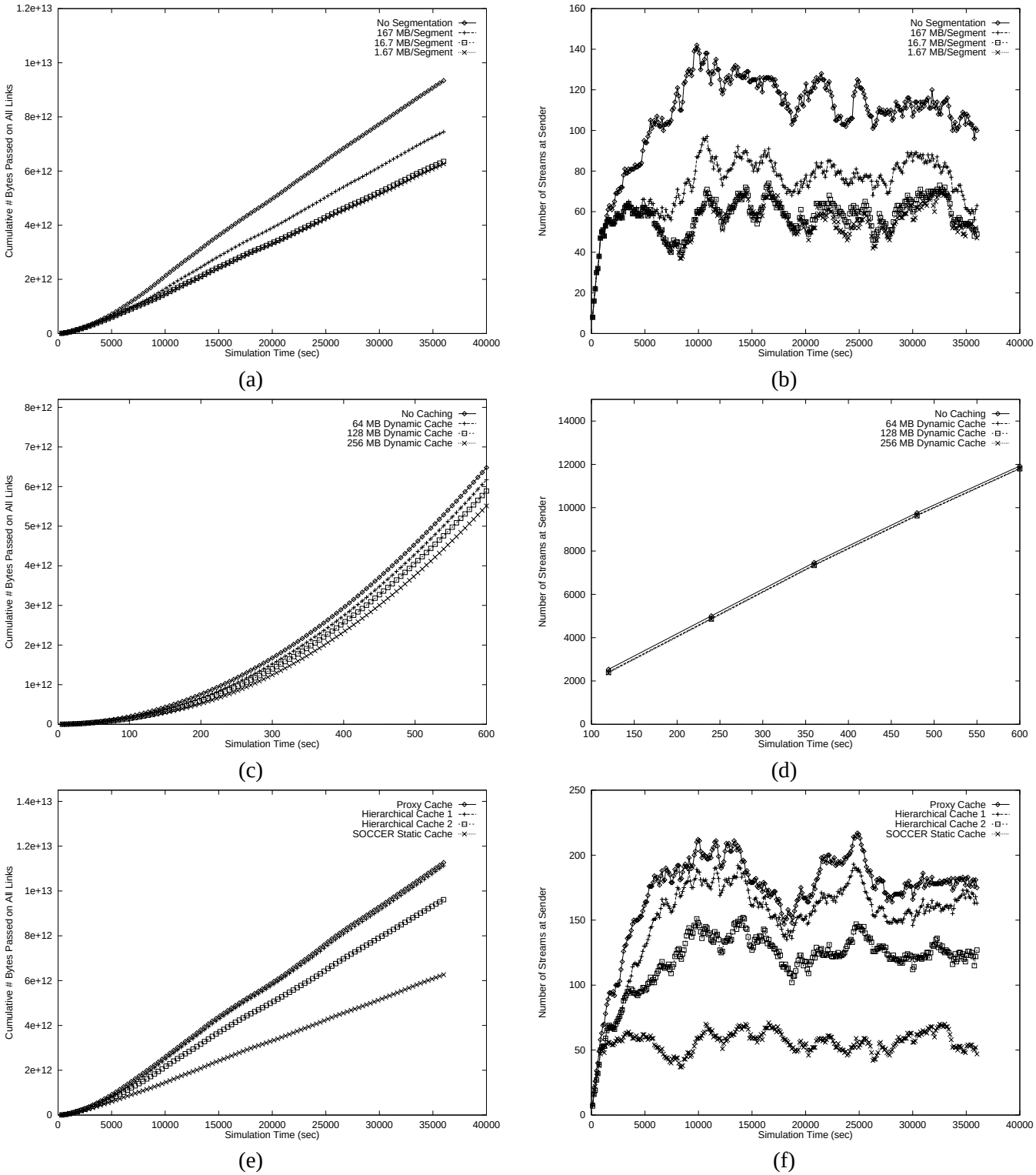

Fig. 8. Various simulation results. (a)/(b) Benefit of segmentation. (c)/(d) Benefit of dynamic caching. (e)/(f) Benefit of self-organizing cooperative caching.

Adver Timer (seconds)	240	120	60	20
Total Data (Bytes)	64.0e11	63.1e11	62.6e11	61.7e11
State Dist Data (Bytes)	31.8e6	81.0e6	184e6	733e6

TABLE IV
OVERHEAD OF STATE DISTRIBUTION

Scheme		Server	Level 1 Helpers	Level 2 Helpers
Proxy	Max	220	36	43
	Avg	173	18	20
Hier 1	Max	196	136	43
	Avg	152	85	20
Hier 2	Max	153	125	45
	Avg	119	80	21
SOCCER	Max	73	45	50
	Avg	56	25	27

TABLE V
SERVER AND HELPERS LOAD IN NUMBER OF STREAMS

are summarized in Table IV.

Finally, note that self-organizing cooperative caching can also distribute load much better than the other schemes. Table V illustrates the load on the server, level 1 helpers and level 2 helpers under the various caching schemes. Since helpers within the same level exhibit similar load, we provide only the aggregate statistics for each level. Under proxy caching, the server is extremely overloaded. Hierarchical caching reduces the server's load slightly, however, much of the burden is carried by the four Level 1 parent caches. The Level 2 children caches are not effective. In contrast, under self-organizing cooperative caching, the maximum and average server load are reduced by about 67% compared to proxy caching. Furthermore, the burden is spread quite evenly among the caches. As a result, network hot-spots are eliminated.

V. RELATED WORK

Related work on web caching systems such as CERN httpd [3], Harvest [4], [5] and Squid [6] has already been mentioned in Section I. These systems are designed for classical web objects and do not offer any support for streaming media. Furthermore, their static web caching architecture is very different from self-organizing cache cooperation and dynamic caching as defined in SOCCER. In these approaches, parent caches and sibling caches are statically configured using a configuration file. Thus, they do not adapt to changes in network and system loads dynamically.

Another body of related work is in the area of scalable video-on-demand systems [8], [13], [14], [15]. The idea is to reduce server load by grouping multiple requests within a time-interval and serving the entire group in a single stream. *Chaining* and *patching* refine the basic idea and define mechanisms to avoid the problem of increased start-up latency for those requests which arrive earlier in a time-interval. The proposed techniques are designed for video on demand systems, and they are not concerned about wide area bandwidth constraints nor about the client's buffer requirements. In fact, they can actually increase clients' storage requirements, while chaining can also lead to increased network load in wide-area networks. SOCCER, on the other hand, aims at reducing network load and client storage requirements in addition to reducing server load. This goal is achieved by utilizing both temporal distance as well as network distance (or spatial proximity) to form the so-called helper mesh. Overall memory requirement is reduced because helpers share buffers with one another.

Other related work has been done on memory caching for multimedia servers [16], [7]. While the basic principle of caching data in different memory levels of a video server has some similarities with storing data in a distributed caching system, there is a fundamental difference. The spatial distance between different memory levels in a server is zero. In contrast, spatial distance between distributed caching systems is not negligible and, therefore, has to be considered in the design of web cache management policies. This requirement is reflected in SOCCER's helper selection and cost function.

Recently, work has been presented on proxy caching architectures for improving video delivery over wide area networks. The scheme presented in [17] makes use of prefix caching in order to smooth bursty video streams during playback. Prefetching and storing portions of a stream in network proxies for the purpose of smoothing has also been proposed in [18]. In [19], a proxy caching mechanism is used for improving delivered quality of layered-encoded multimedia streams. All these approaches are based on a relative simple proxy architecture without any collaboration between the proxies. Our work is complementary in the sense that it focuses on the design of an efficient collaborative proxy/caching architecture. The proposed mechanisms for video smoothing and quality adaptation can use our architecture with the additional benefits of a collaborative caching environment.

Finally, the "middleman" architecture presented in [20] is relevant to SOCCER. It forms clusters of caches where each cluster uses a *coordinator* to keep track of the state of caches in the cluster. This centralized approach contrasts sharply with SOCCER's distributed architecture in which each helper maintains global state information and makes local decisions.

VI. CONCLUSIONS AND FUTURE WORK

This research is a first step to understand the issues of extending the caching principle to support streaming me-

dia over the Internet. We study relevant streaming techniques, generalize and apply them in the context of distributed caching. Furthermore, we propose several novel techniques, including segmentation of streaming objects, combined use of dynamic and static caching, and self-organizing cooperative caching. We integrate all these techniques into a complete solution for caching streaming media in the Internet by defining the Self-Organizing Cooperative Caching Architecture (SOCCER). Our simulations showed that self-organizing cooperative caching can significantly reduce network and server load compared to proxy or hierarchical caching. It turns out that the overhead for state distribution becomes almost negligible when using the proposed distribution protocol based on Expanding Ring Advertisement (ERA). Our results also show that heavily loaded systems will benefit from segmentation of media objects and from dynamic caching. More work needs to be done on evaluating and assessing different cost functions for helper selection and to better understand the trade-offs between effectiveness and complexity.

Our current implementation of a streaming helper is in compliance with IETF standard protocols, notably *Real Time Streaming Protocol (RTSP)* [21] and *Real-time Transport Protocol (RTP)* [22]. Therefore, it can interoperate with any standard-compliant streaming server and client in a transparent way. Detailed information on the current implementation can be found in [23]. The implementation will be used for further evaluation in a production network.

REFERENCES

[1] Vxtreme, "Internet homepage," http://www.vxtreme.com, 1999.

[2] Real Networks, "Internet homepage," http://www.real.com, 1999.

[3] T Berners-Lee A. Lutonen, H.F. Nielsen, "Cern httpd," http://www.w3.org/Daemon/Status.html, 1996.

[4] C.M. Bowman, P.B. Danzig, D.R. Hardy, U. Manber, M.F. Schwartz, and D.P. Wessels, "Harvest: A scalable, customizable discovery and access system," Tech. Rep. CU-CS-732-94, University of Colorado, Boulder, USA, 1994.

[5] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Scwartz, and K.J. Worrell, "A hierarchical internet object cache," in *Proceedings of the 1996 Usenix Technical Conference*, 1996.

[6] D. Wessels, "Icp and the squid cache," National Laboratory for Applied Network Research, 1999, http://ircache.nlanr.net/Squid.

[7] A. Dan and D. Sitaram, "Multimedia caching strategies for heterogeneous application and server environments," *Multimedia Tools and Applications*, vol. 4, no. 3, 1997.

[8] K. A. Hua, Y. Cai, and S. Sheu, "Patching: A multicast technique for true video-on-demand services," in *Proceedings of ACM Multimedia '98*, Bristol, England, Sept. 1998.

[9] M. Hofmann and M. Rohrmuller, "Impact of virtual group structure on multicast performance," Fourth International COST 237 Workshop, Lisboa, Portugal, December 15-19, 1997.

[10] UCB/LBNL/VINT, "Network simulator, ns," http://www-mash.cs.berkeley.edu/ns, 1999.

[11] D. Waitzman, C. Partridge, and S. Deering, "Distance vector multicast routing protocol," Internet Request for Comments 1075, November 1988.

[12] G. K. Zipf, "Relative freqency as a determinant of phonetic change," Reprinted from the Harvard Studies in Classical Philology, Volume XL, 1929.

[13] A. Dan, D. Sitaram, and P. Shahabuddin, "Dynamic batching policies for an on-demand video server," *Multimedia Systems*, vol. 4, no. 3, pp. 51–58, June 1996.

[14] C.C. Aggarwal, J.L. Wolf, and P.S. Yu, "On optimal batching policies for video-on-demand storage servers," in *Proc. of International Conference on Multimedia Systems'96*, June 1996.

[15] S. Sheu, K. A. Hua, and W. Tavanapong, "Chaining: A generalized batching technique for video-on-demand systems," in *Procedings of IEEE International Conference on Multimedia Computing and Systems*, Ottawa, Ontario, Canada., 1997.

[16] A. Dan and D. Sitaram, "A generalized interval caching policy for mixed interactive and long video environments," in *Proceedings of IS&T SPIE Multimedia Computing and Networking Conference, San Jose, CA*, January 1996.

[17] S. Sen, J. Rexford, and D. Towsley, "Proxy prefix caching for multimedia streams," in *Proceedings of IEEE Infocom'99*, New York, USA., 1999.

[18] Y. Wang, Z.-L. Zhang, D. Du, and D. Su, "A network conscious approach to end-to-end video delivery over wide area networks using proxy servers," in *Proc. of IEEE Infocom*, April 1998.

[19] R. Rejaie, M. Handley, H. Yu, and D. Estrin, "Proxy caching mechanisms for multimedia playback streams in the internet," in *Proceedings of the 4th International Web Caching Workshop, San Diego, CA.*, March 1999.

[20] Soam Acharya, *Techniques for Improving Multimedia Communication Over Wide Area Networks*, Ph.D. thesis, Cornell University, Dept. of Computer Science, 1999.

[21] H. Schulzrinne, A. Rao, and R. Lanphier, "Real time streaming protocol," Internet Request for Comments 2326, April 1998.

[22] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "Rtp: A transport protocol for real-time applications," Internet Request for Comments 1889, January 1996.

[23] E. Bommaiah, K. Guo, M. Hofmann, and S. Paul, "Design and implementation of a caching system for streaming media," Tech. Rep. BL011345-990628-05 TM, Bell Laboratories, June 1999.