A Segmentation-Based Fine-Grained Peer Sharing Technique for Delivering Large Media Files in Content Distribution Networks

Zongming Fei, *Member, IEEE,* and Mengkun Yang, *Student Member, IEEE*

***Abstract*— Delivering large media files over the Internet is a challenging task because it has some unique features that are different from delivering conventional web documents. In this paper, we propose a fine-grained peer sharing technique for dealing with the problem in the context of content distribution networks. The key difference of the technique from conventional peer to peer systems is that the unit of peer sharing is not a complete media file, but at a finer granularity. By doing so, we improve the flexibility of replica servers for handling client requests. We analyze the storage requirement at replica servers and design a scheduling algorithm to coordinate the delivery process from multiple replica servers to a client. Our simulations show that the fine-grained peer sharing approach can reduce the initial latency of clients and the rejection rate of the system significantly over a simple peer sharing method.**

***Index Terms*— Peer-to-peer, multimedia streaming, video segmentation.**

I. Introduction

Content distribution networks (CDNs) have been successful in delivering conventional web documents to clients by distributing content from an origin server to replica servers located at the edges of the network and directing clients to a nearby replica server. Multimedia streaming presents new challenges to content distribution because of its unique features, such as large file size, high bandwidth requirement, sustained streaming duration, and high quality of service requirement. These characteristics lead us to reconsidering design issues in the CDNs for delivering large media files.

The CDNs for delivering conventional web documents usually distribute the whole file of a document to replica servers. The upcoming streaming media files tend to be much larger and the number of files a replica can store is limited, even with the increasing disk space. The result is that either we have to store only a small portion of contents at replica servers and the rest will be delivered directly from the origin server, or we have to frequently remove previously stored files from the storage to accommodate new files. This consumes quite a lot of network bandwidth and can also overload the origin server.

To improve the service capabilities of resource-constrained replica servers, we employ the peer sharing technique of peer-to-peer systems at the replica servers. Peer sharing among clients is an attractive approach, as proposed by most peer-to-peer systems. However, the difficulty lies in that clients are more willing to get the files than serving other clients. In this paper, we explore the possibility of peer sharing among replica servers, which content service providers have direct control of and can equip with the new technique expecting their full participation. Usually the unit of peer sharing is a complete file. Peer replica servers can best help each other if they store different files. This can be done by careful planning so that their capacities to serve clients can be fully utilized [1].

We go one step further in this paper by proposing a fine-grained peer sharing technique. The unit of peer sharing is not a complete media file. Rather, a media file is divided into segments, which will be distributed to different replica servers. We analyze the effect on the storage requirement at replica servers by different segmentation schemes, which determine the size of each segment and the number of copies of each segment distributed to a group of replica servers. Our goal is to minimize the storage requirement at the replica servers, reduce the initial latency, guarantee the smooth playout and accommodate the bursty arrival of client requests. We observe that the number of copies distributed to replica servers for each segment should reflect the timing constraints and the frequency of requests for it. For example, the initial segment has to be delivered to clients as soon as possible to reduce the initial delay, while streaming of other segments can be more flexible because we can use techniques such as prefetching.

In the fine-grained peer sharing scheme, a client still makes the request to its nearest replica server, which will act as a coordinator to schedule which segment will be streamed from which cooperating replica server to the client at what time. We design a protocol for peers to exchange information about available resources with each other, and a scheduling algorithm to coordinate the delivery process from multiple replica servers to the client. Simulations show that the fine-grained peer sharing approach can reduce the initial latency of clients and the rejection rate of the system significantly over a simple peer sharing method.

The rest of this paper is organized as follows. Section II proposes the fine-grained peer sharing technique. Section III analyzes the storage requirement at replica servers. Section IV and V describe the scheduling algorithm and performance results, respectively. We discuss related work in Section VI and conclude the paper in Section VII.

Manuscript received February 7, 2004; revised August 13, 2004. This work was supported in part by the National Science Foundation under Grants CCR-0204304 and EIA-0101242, and a grant from the Kentucky Science and Engineering Foundation as per Grant Agreement #KSEF-148-502-05-139 with the Kentucky Science and Technology Corporation.

Zongming Fei and Mengkun Yang are with the Department of Computer Science, University of Kentucky, Lexington, KY 40506 USA (e-mail:fei@cs.uky.edu; myang0@cs.uky.edu).

II. Fine-Grained Peer Sharing in CDNs

A. Peer Sharing in CDNs

In a CDN, a client is normally directed to the nearest replica server. Under the limited storage constraint, we assume that a replica server can store N media files. Usually it will store the most popular or most recently accessed objects. For those files that are not stored at the replica server, the client or the replica server has to get them from the origin server. This may impose a tremendous load on the origin server because of the number of potential clients and the long streaming duration.

One approach to dealing with this problem is *peer sharing*, i.e., cooperating replica servers store different objects and they will serve the requests made to their peer servers. If we have m servers peering with each other, we can store $m * N$ different objects, instead of N in the non-sharing case. An observation is that a larger m will result in more objects being served from replica servers. However, m cannot be too large. One reason is that grouping more servers together means a longer distance from one of them to the client making the request. This may affect the quality (delay, delay jitter, bandwidth) of the streaming path. Another reason is that the management cost will be higher because of the increased complexity of communication between replica servers.

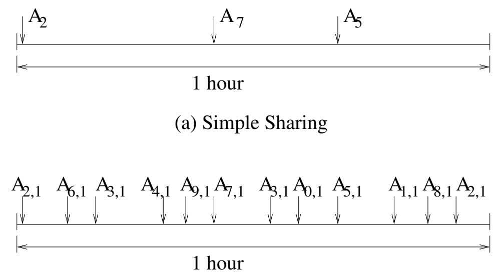
We envision a two-tier model for organizing the replica servers in the CDN. The replica servers are no longer in a flat structure but divided into groups. The replica servers in a group should be relatively close in the network distance, and they will cooperate to serve the content to clients attached to any replica server in the group.

B. Fine-Grained Peer Sharing

The problem with the simple peer-sharing technique described above is that it is possible that the load on one of the replica servers may be extremely high because it stores one or several hot media files. In the non-sharing case, the requests to these hot media files are distributed to all replica servers. In the sharing case, all these requests are now concentrated on a particular replica server that stores the hot files. Therefore, clients may experience excessive delay in obtaining the media file requested.

Instead of storing the whole media file in one server, we divide each media file into segments [2]. Generally one replica server will store only one segment of a media file. A client still makes the request to the nearest replica server, which will communicate with those peering servers having segments of the requested file and schedule streaming of each segment to the client.

We use a simplified example to illustrate the idea of the fine-grained peer sharing technique. Assume four replica servers A, B, C and D peer with each other, and each of them can stream 3 media files simultaneously at the playout rate. Each of them can hold 10 media files lasting 1 hour each. First consider the *simple peer sharing* case. With careful placement, each of them holds different files. Specifically, we assume that A stores files $A_1, \cdots, A_{10}$, B stores files $B_1, \cdots, B_{10}$, etc. Because each file lasts 1 hour and a server can stream 3 media files at the same time, the total number of requests for files $A_1, \cdots, A_{10}$ that we can start to serve within one hour is 3. The same is true of files stored at servers B, C and D.



(a) Simple Sharing

(b) Fine−grained Sharing

Fig. 1. Comparison between simple sharing and fine-grained sharing

Now consider the *fine-grained peer sharing* case. We divide each file into 4 segments and store them into different servers. For example, file A_1 is divided into $A_{1,1}, A_{1,2}, A_{1,3}$ and $A_{1,4}$, which will be stored to servers A, B, C and D, respectively. All other files are divided and distributed in a similar way, except that $B_{j,1}$ will be kept in server B, $C_{j,1}$ in C, $D_{j,1}$ in D ($j = 1, 2, 3, \cdots, 10$) and the rest of the segments are distributed in a round robin fashion. Now consider the total number of requests for files $A_1, \cdots, A_{10}$ that we can start to serve within one hour. Since each segment only lasts 1/4 hour, in each 1/4 hour we can start 3 streams. Over 1 hour period of time, we can start 3*4 = 12 streams for requests for $A_1, \cdots, A_{10}$. As an example, Figure 1(a) shows that the simple sharing method can only start three video files (A_2, A_7, A_5) within one hour, while Figure 1(b) shows that the fine-grained peer sharing method can start 12 video files within one hour.

It is true that the total number of requests that can be served by a given set of cooperating replica servers within a certain period of time is not necessarily increased. However, because the load for serving a given file is more evenly distributed, the number of requests for *a specific file* or *a specific set of files* that can be served within a certain period of time by the fine-grained peer sharing scheme is increased.

The above example illustrates that the fine-grained peer sharing can improve the capacity of the system to serve a specific file or a given set of files. Each server depends on the help from other servers to accomplish the task. However, these other servers may need to stream segments of other files. We are interested in how these streamings will interact with each other and the resulting overall system performance measured in waiting time in those scenarios. The simple sharing can be modeled as m queues and each request will be processed by one of the queues. The fine-grained sharing can be modeled as m queues with the processing time being only $1/m$ of that of the queue in the simple sharing case, but each request needs to be processed by m queues. Using the M/G/1 queuing model [3], it can be shown that the waiting time of the fine-grained peer sharing is much shorter than that of the simple sharing. The intuition behind the fine-grained sharing is that the load is more balanced and the shorter segments provides more flexibility for servers to schedule. Therefore, the fine-grained sharing can better serve clients than the simple sharing.

III. Segmentation Schemes

Using the fine-grained peering technique requires dividing video files into segments. Different ways to do the segmentation can be distinguished by two parameters. The first one is the relative size of these segments and the second one is the number of copies of each segment stored in a group of replica servers. They are usually treated as independent design parameters [4]. In contrast, we will show that they are related and give series functions for these two parameters [5].

Consider a video that is divided into n segments. We want to distribute different segments of a file to different replica servers. Assume that the number of replica servers in a cluster is m. Usually, $n \leq m$.

The length of each segment is defined by a weight function w and the length $L(i)$ of segment i is proportional to $w(i)$. Therefore, we have $L(i) = \frac{w(i)}{\sum_{j=1}^{n} w(j)} * \mathcal{L}$, where $\mathcal{L}$ is the total size of the video. The number of copies of each segment is given by a function f. With segment i having $f(i)$ copies, the total storage required for a video in the cluster is $S = \sum_{i=1}^{n} f(i)L(i)$. The goal is to minimize the storage requirement in a cluster of replica servers and at the same time to be able to accommodate the bursty arrival of client requests.

For a given segment, it can only be distributed to m servers. More than one copies at the same replica server do not improve its ability to serve client requests. Therefore, the maximal number of copies that a segment can have in a cluster is m, i.e., $f(i) \leq m$. In the following discussion, this constraint should always be applied, though we will not mention it every time.

We assume the arrival rate of requests for a video in a cluster is λ (requests per second) and the playout rate of the video is ρ (megabits per second). We use $\mathcal{T}$ to represent the playout duration of a video. Obviously, $\mathcal{T} = \mathcal{L}/\rho$. Replica servers may have different capacities and serve different numbers of clients at the same time. To simplify our analysis, we assume that each replica server can serve k clients at the same time, i.e., it can send out k streams at the playout rate of a video.

A. Lower Bound on Storage Requirement

We first consider an ideal case that gives the minimal storage requirement (a lower bound) for a specific file, under a given arrival rate. We assume that a video is divided *equally* into n segments, i.e., $w(i) = w(1)$ for $1 \leq i \leq n$, and then the size of each segment is $L(i) = \mathcal{L}/n$. The duration for streaming one segment is $t_s = \mathcal{T}/n$. In this ideal case, the requests come *evenly* every t_s second. The total number of requests during a video playout is $\lambda\mathcal{T}$. The maximal possible number of requests in one segment time is $\lambda\mathcal{T}/n$. Considering that a replica server can serve k requests at the same time, we will be able to meet the request requirement by setting $f(i) = f(1) = \lceil \lambda\mathcal{T}/(kn) \rceil$. The total storage requirement is $S = \sum_{i=1}^{n} L(i)f(i) = \sum_{i=1}^{n} \frac{\mathcal{L}}{n} * \lceil \frac{\lambda\mathcal{T}}{kn} \rceil = \mathcal{L} \lceil \frac{\lambda\mathcal{T}}{kn} \rceil$

For comparison, we also derive the storage requirement in the *simple peer sharing* case where the whole video will be stored in a single replica without segmentation. To satisfy $\lambda\mathcal{T}$ requests during the video playout time, we need to have $\lceil \lambda\mathcal{T}/k \rceil$ copies of the video, because a server can serve k requests at the same time. The storage requirement is $S = \mathcal{L} \lceil \frac{\lambda\mathcal{T}}{k} \rceil$. To serve the same number of requests for a specific video, the *simple peer sharing* (or *unsegmented*) approach will need about n times storage as the fine-grained peer sharing approach in the ideal case discussed above.

B. Storage Requirement with Bursty Traffic

We consider a more general case in this section. In practice, requests for a given video can come at any time. For example, the average number of requests during a video playout time is $\lambda\mathcal{T}$, and it is possible that all the requests arrive within one segment time. The arrival can be more bursty than this case. However, with the limited resource in a cluster we cannot satisfy all arbitrary arrival patterns. Our assumption is that the arrival during a segment time is at most n times the average rate, or equal to the average number of requests that may arrive during the whole video playout time.

We leave w open and find out the number of copies for each segment in order to accommodate the above-mentioned burstiness of the traffic. The worst case scenario is that all requests within a video playout time come during the first segment time. To satisfy these requests ($\lambda\mathcal{T}$), the number of copies for the first segment $f(1) = \lceil \lambda\mathcal{T}/k \rceil$.

While the first segment must be served during the first segment time, the second segment has more flexibility. It can be either prefetched by the client during the first segment time, or served by a replica server during the second segment time. Therefore, the time we can serve the second segment is the total time of playing out the first segment and the second segment. As long as the second segment can be streamed to clients within the second segment time, the continuous playout requirement is satisfied. Therefore, the number of clients that can be served by one copy of the second segment is $k * \lfloor \frac{L(1)+L(2)}{L(2)} \rfloor = k * \lfloor \frac{w(1)+w(2)}{w(2)} \rfloor$. The number of copies for the second segment is: $f(2) = \lceil \frac{\lambda\mathcal{T}}{k * \lfloor \frac{w(1)+w(2)}{w(2)} \rfloor} \rceil$.

In general, the i-th segment can be served during the first, second, until i-th segment. Therefore, one copy of the i-th segment can serve $k * \lfloor \frac{\sum_{j=1}^{i} L(j)}{L(i)} \rfloor = k * \lfloor \frac{\sum_{j=1}^{i} w(j)}{w(i)} \rfloor$ client requests. The number of copies for the i-th segment is: $f(i) = \lceil \frac{\lambda\mathcal{T}}{k * \lfloor \frac{\sum_{j=1}^{i} w(j)}{w(i)} \rfloor} \rceil$.

The determining factor then is the series function w. We first consider a simple case where all the segments are of equal size, i.e., $w(i) = 1$ for all i. We will use EQ to denote this series function in the figures. We can get $L(i) = \mathcal{L}/n$ and $f(i) = \lceil \frac{\lambda\mathcal{T}}{k * i} \rceil$. So the total storage requirement is $S = \sum_{i=1}^{n} \lceil \frac{\lambda\mathcal{T}}{k * i} \rceil * \mathcal{L}/n$.

We further study the cases in which segments are of unequal size. Here is a list of series functions for w. We derive the corresponding series function f.

1) The size of segments increases exponentially.
   - EXP1: The simplest series is 1, 2, 4, 8, etc. That is, $w(i) = 2^{i-1}$. We can get $f(i) = \lceil \frac{\lambda\mathcal{T}}{k} \rceil$.
   - EXP2: The size of segment i is the sum of all segments up to $i-1$, i.e., 1, 1, 2, 4, 8, etc. That is,

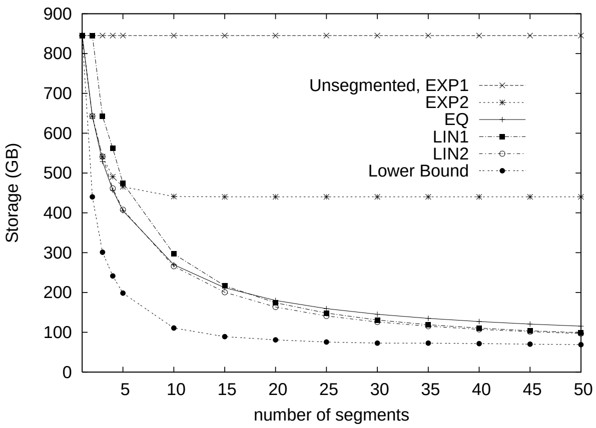


Fig. 2. Size of Segments

$w(i) = \sum_{j=1}^{i-1} w(j)$. We can get $f(i) = \lceil \frac{\lambda \mathcal{T}}{k*2} \rceil$ for $i \geq 2$, and $f(1) = \lceil \frac{\lambda \mathcal{T}}{k} \rceil$.

2) The size of segments increases linearly.
   - LIN1: The size of segment i is proportional to i, i.e., $w(i) = i$. We can get $f(i) = \lceil \frac{\lambda \mathcal{T}}{k * \lceil i/2 \rceil} \rceil$.
   - LIN2: The series function is 1,1,2,2,3,3,4,4, ..., i.e., $w(i) = \lceil i/2 \rceil$. We can get $f(i) = \lceil \frac{\lambda \mathcal{T}}{k*(\lfloor i/2 \rfloor + 1)} \rceil$.

C. Dealing with Multiple Files

The previous analysis focuses on the storage requirement by a single video file. Usually we need to deliver multiple files in a CDN. In this case, we may only know the overall request rate (Λ) to all files. To get the storage requirement discussed above, we need to derive the request rate (λ) to an individual file. One model widely used in the CDNs is the Zipf-like distribution [6]. That is, the probability of requesting the i-th most popular file is proportional to $1/i^{\alpha}$. Using this model, we can derive the request rate for each individual file. Specifically, the request rate to the i-th most popular file will be $\lambda_i = \Lambda \frac{1/i^{\alpha}}{\sum_{j=1}^{M} 1/j^{\alpha}}$, where M is the total number of video files. Based on the results, we can get the number of copies for each segment of the file.

After dividing each file into segments, they will be distributed to replica servers in the cluster. We distribute one file at a time. At each replica server, we have an array of n counters. The i-th counter records the number of files having its i-th segment distributed to this replica server. When distributing i-th segment of a file, we choose a replica server with the smallest value in its i-th counter, i.e., with the least number of i-th segments residing on it. Though we may have a better placement if we consider multiple (or all) files at the same time, this simple solution is good enough to achieve the goal of balancing the load on these replica servers.

D. Numerical Results

In Fig. 2, we compare the storage requirement of segmentation schemes, assuming the video length equals 120 minutes, each replica server can serve $k = 10$ clients at the same time, the playout rate is 1.5Mbps, and the request rate is 50/minute. We also assume that m is large enough so that it will not constrain either n or $f(i)$. Though the ideal case described in Subsection III-A usually will not be able to meet the requirement of client requests, it provides a comparison baseline (denoted as "lower bound" in the figure). We can find that the simple equal segmentation requires much less storage than the unsegmented approach. For example, if the number of segments is 35, we need only one sixth of storage required by the unsegmented approach. Also the storage requirement decreases as the number of segments increases. All exponential series need much larger storage than the two linear series, which are very close to the equal size series. Because of the simplicity of the equal size series, we will use it in the performance evaluation unless otherwise noted.

An interesting observation from the figure is that the storage decreases sharply when the number of segments increases from 1 to 10 in the fine-grained scheme using EQ segmentation. When we further increase the number of segments, the storage requirement decreases very slowly. When the number of segments increases beyond 35, the storage benefit is hard to observe. This gives us a hint that the number of segments should not be too large, because the benefit in the saving of storage is only marginal while the management cost will increase significantly. A guideline we obtain from this observation is that the number of segments should be around 10, where the storage saving is obvious while the cost of segmentation is manageable.

Combining with the constraint $n \leq m$ given at the beginning of this section, we get the method for selecting the number of segments n, which will be used in the performance evaluation. If the number of replica servers in a group (m) is smaller than 10, we set the number of segments (n) to the number of replica servers (m). Otherwise, we set n to 10.

IV. Communication Protocol and Scheduling Algorithm

When a media file is delivered to a group of peering replica servers, it is divided into segments based on the segmentation schemes. The information about locations of each segment of each file will be distributed to all replica servers in the group, so that they know where to ask for help when needed. Accessing different segments from different servers needs coordination. The nearest replica server of the client acts as the *coordinator*. We design a protocol for communications between the client, the coordinator and other replica servers, and a scheduling algorithm for the coordinator to decide streaming times from replica servers to the client.

The server and client capacities are abstracted as channels. By a *channel*, we mean the bandwidth used to stream data at the *typical* playout rate of media files. If the server bandwidth is B and the typical playout rate (also called channel rate later in the paper) is ρ, then the number of channels this server has is $B_s = \lfloor B/\rho \rfloor$. Similarly, we have a parameter B_c, which represents the number of streams (at the playout rate) that a client can receive from replica servers at the same time.

Note that here we assume that ρ is the typical playout rate in the sense that more media files have this playout rate than other rates. However, we allow media files to have a playout rate different from ρ. In that case, the length of a slot reserved for a segment on server or client channels will be different from the time interval for playing out this segment. Our scheduling algorithm can accommodate these cases.

A. Communication Protocol

Streaming a media file from replica servers to a client involves several steps of communications among the client, the nearest replica server and other peer replica servers. The communication protocol can be described as follows.

First, the client sends its request for a file to its nearest replica server, which acts as the coordinator for this request.

Second, the coordinator looks up in the table about the locations of segments of this file, and forwards *segment requests* to those peering servers having segments of this file.

Third, after receiving segment requests, the peering replica servers try to find available slots that can be used to stream the segment. These slots will be reserved for a period of time τ for the coordinator. It is a policy of the replica server that determines how many slots and how long these slots will be reserved for the coordinator.

Fourth, upon receiving replies from these peers, the coordinator uses the *scheduling algorithm* described in the next subsection to choose an appropriate time slot for streaming each segment.

Fifth, confirmation messages are sent back to peer servers from the coordinator. They will inform each peer which slot (or part of it) has been scheduled to stream the requested segment. At the same time, the coordinator sends the client a message about when to start playing out and where to get each segment.

Finally, after receiving the confirmation messages, each peer server confirms the reservation of the selected time slot for transmitting the segment of the file and releases all other channel slots previously reserved for this request. If a certain server does not receive the confirmation message from the coordinator before the lifetime τ expires, it will cancel all channel slots reserved for the segment.

One issue worth further discussion is how to deal with concurrent requests from multiple peers. In the third step, a replica server may receive segment requests from multiple coordinators, which may ask for the same segment or different segments. Obviously, reserving all available slots for one coordinator will block requests from all other coordinators. Our strategy is to count the total length of reserved slots every τ time period. Assume the length in the most recent period is Γ_c and the variable used to represent the load on this server is Γ. We calculate the current load as weighted sum, i.e., $\Gamma = \beta * \Gamma_c + (1-\beta) * \Gamma$, where $\beta = 0.3$. The proportion of slots that will be reserved for each coordinator will be $1/(\Gamma/s)$ of all available slots up to a certain time in the future, where s is the size of the segment currently being requested.

B. Scheduling Algorithm

Upon receiving information about available time slots for streaming each segment, the scheduling algorithm of the coordinator will determine the time slot that will be actually used. For the time being, we assume the clocks of these replica servers are synchronized. Assume that the client makes the request for file γ at time t_{rqst}. The bit rate of the video is ρ_γ, which can be different from the channel rate ρ. The length of the media file is $\mathcal{L}_\gamma$, which is divided into n segments. The playout time of segment i is $s_{\gamma,i} = L_\gamma(i)/\rho_\gamma$. If all segments are of equal size, we have $s_{\gamma,i} = s_\gamma = \frac{\mathcal{L}_\gamma}{n\rho_\gamma}$. Note that different files may have different file size ($\mathcal{L}_\gamma$), segment size ($L_\gamma(i)$), and segment playout time ($s_{\gamma,i}$). Let $t_i = t_{rqst} + \sum_{j=1}^{i-1} s_{\gamma,j}$ for $1 \le i \le n$. Ideally, there will be no delay if the client can start downloading segment i ($1 \le i \le n$) at time t_i and finish downloading at time t_{i+1}. However, we may have to start downloading segment i before or after t_i because the time slots starting at time t_i on all channels of the replica server may have already been reserved for other requests. While an actual start downloading time before t_i and a finish downloading time before t_{i+1} will not cause any problem, a start downloading time after t_i or a finish downloading time after t_{i+1} will cause the client to start playing out the whole media file after some initial latency. Actually, the final initial latency is the maximum of the latencies caused by all segments. Our algorithm will try to reduce this latency. We start with some notations used in the algorithm.

- For any request being processed, the available server channel slots returned from peer replica servers for segment i ($i = 1, \cdots, n$) is $\Omega_i = \{(a_{i1}, b_{i1}), (a_{i2}, b_{i2}), (a_{i3}, b_{i3}), \cdots\}$, where $a_{i1} \le a_{i2} \le a_{i3} \le \cdots$.
- We assume that the client making the request can receive from B_c streams at the same time. The capability for downloading each stream is modeled as a channel. At one point of time, the available slots on the l-th channel of the client is $\Psi_l = \{(x_{l1}, y_{l1}), (x_{l2}, y_{l2}), (x_{l3}, y_{l3}), \ldots, (x_{lh_l}, \infty)\}$, where $l = 1, \cdots, B_c$ and $x_{l1} < x_{l2} < x_{l3} < \cdots < x_{lh_l}$. Initially, all channels are (t_{rqst}, ∞).

For any request, given server slots $\Omega_1, \Omega_2, \cdots, \Omega_n$ together with client channels $\Psi_l = \{(t_{rqst}, \infty)\}$ for $l = 1, \cdots, B_c$, we need to find the streaming slot (u_i, v_i) for segment i ($1 \le i \le n$), where $v_i = u_i + \frac{\rho_\gamma}{\rho} s_{\gamma,i}$, such that (u_i, v_i) is a time slot that does not collide with time slots for downloading other segments in the same client channel, and there exists a j such that $(u_i, v_i) \subseteq (a_{ij}, b_{ij})$. If $u_i \le t_i \wedge v_i \le t_{i+1}$, the streaming time ($(u_i, v_i)$) is before the ideal playout time and no delay will be caused by segment i. If $u_i > t_i$ or $v_i > t_{i+1}$, the streaming time is after the ideal playout time and segment i will require the playout to delay $\max(u_i - t_i, v_i - t_{i+1})$. In summary, the latency caused by segment i is $d_i = \max(u_i - t_i, v_i - t_{i+1}, 0)$, and the final initial latency of the client will be $d = \max\{d_i | i = 1, \cdots, n\}$. The client can start playing out the media file at time $t_{rqst} + d$ without jitter for each segment. The goal of the algorithm is to minimize d.

The basic idea of the algorithm is to reserve the *earliest possible slot* on the client channels for each segment first, then try to make adjustments to minimize the initial latency d. Given the current available slots Ω_i from peer servers and client channels $\Psi_1, \cdots, \Psi_{B_c}$, the current *earliest possible slot* for segment i is (u_i, v_i), where $v_i = u_i + \frac{\rho_\gamma}{\rho} s_{\gamma,i}$, satisfying:

Scheduling Algorithm

1: **for** $(i = 1 \text{ to } n)$ **do**
2: find (u_i, v_i), where $v_i = u_i + \frac{\rho_\gamma}{\rho} s_{\gamma,i}$, satisfying *earliest condition* for segment i on a client channel and record that the slot is on l_i-th client channel.
3: $\Psi_{l_i} \leftarrow \Psi_{l_i} \ominus (u_i, v_i)$
4: $d_i \leftarrow \max(u_i - t_i, v_i - t_{i+1}, 0)$
5: Sort d_i and fill in $Q[1 \dots n]$ accordingly.
6: $i \leftarrow n$
7: **while** $i > 1$ **do**
8: $j \leftarrow Q[1]$, $j' \leftarrow Q[i]$
9: $\Psi_{l_j} \leftarrow \Psi_{l_j} \oplus (u_j, v_j)$
10: $\Psi_{l_{j'}} \leftarrow \Psi_{l_{j'}} \oplus (u_{j'}, v_{j'})$
11: find (u, v), where $v = u + \frac{\rho_\gamma}{\rho} s_{\gamma,j}$, satisfying *earliest condition* for segment j on a client channel and record that the slot is on l-th client channel.
12: find (u', v'), where $v' = u' + \frac{\rho_\gamma}{\rho} s_{\gamma,j'}$, satisfying *earliest condition* for segment j' on client channel and record that the slot is on l'-th client channel.
13: **if** $\max(u - t_j, v - t_{j+1}, 0) < d_j$ and $\max(u' - t_{j'}, v' - t_{j'+1}, 0) < d_j$ **then**
14: $u_j \leftarrow u$, $v_j \leftarrow v$, $l_j \leftarrow l$
15: $u_{j'} \leftarrow u'$, $v_{j'} \leftarrow v'$, $l_{j'} \leftarrow l'$
16: Update d_j, $d_{j'}$, and $Q[1 \dots n]$
17: $i \leftarrow n$
18: **else**
19: $\Psi_{l_j} \leftarrow \Psi_{l_j} \ominus (u_j, v_j)$
20: $\Psi_{l_{j'}} \leftarrow \Psi_{l_{j'}} \ominus (u_{j'}, v_{j'})$
21: $i \leftarrow i - 1$

Fig. 3. The scheduling algorithm

1) $u_i \geq t_{rqst}$; 2) $(u_i, v_i) \subseteq (a_{ij}, b_{ij}) \cap (x_{lh}, y_{lh})$ for some j, l, h; and 3) for any (e, e'), where $e' = e + \frac{\rho_\gamma}{\rho} s_{\gamma,i}$, such that $e \geq t_{rqst} \wedge (e, e') \subseteq (a_{ij'}, b_{ij'}) \cap (x_{l'h'}, y_{l'h'})$ for some j', l', h', we have $u_i \leq e$. We call such a slot satisfying the *earliest condition* in the algorithm.

The greedy reservation for the earliest possible slots favors early segments and may use up the only slots that can be used to prefetch some later segments that have a larger latency. So the next step in the algorithm is to gradually reduce the highest segment latency by moving a segment with zero or very low latency from its current reserved slot to a later free slot, and let the vacated slot on a client channel to accommodate the segment with the highest latency.

Two operations, $\oplus$ and $\ominus$, are defined on Ψ_l. Basically, $\oplus$ inserts (releases) a slot (x, y) into the channel Ψ_l, and $\ominus$ removes (reserves) a slot (x, y) from the channel. If inserting a slot makes some slots on a channel contiguous to each other, we further combine them to one larger slot. The priority queue $Q[1 \dots n]$ stores segment indices such that $d_{Q[1]} \geq d_{Q[2]} \geq \cdots \geq d_{Q[n]}$.

The scheduling algorithm is outlined in Figure 3. Lines 1-5 reserve slots on client channels for segments as early as possible. So at the time we schedule for segment i, (u_i, v_i) on channel l_i is the earliest possible slot. The priority for reservation decreases as the segment index increases. Some segments may get high latencies because their residing servers cannot stream them on time, and the only possible slots able to prefetch them have already been taken up by other segments. To solve this problem, lines 6-21 move the segment with the highest latency to an earlier slot previously reserved for another segment with lower latency. In this way, the initial latency of the client can be decreased. The process is continued until no such move is possible.

The scheduling algorithm uses a greedy strategy to reserve earliest possible slots for all segments. This may lead to later clients unable to get a slot which otherwise may be available. Notice the goal of the algorithm is to minimize the initial latency. Later segments only need to be delivered before its ideal playout time, rather than as early as possible. Therefore, we can use an alternative strategy that reserves the *latest in-time slot*. It can be defined as (u_i, v_i), where $v_i = u_i + \frac{\rho_\gamma}{\rho} s_{\gamma,i}$, satisfying: 1) $u_i \geq t_{rqst}$; 2) $u_i \leq t_i \wedge v_i \leq t_{i+1}$; 3) $(u_i, v_i) \subseteq (a_{ij}, b_{ij}) \cap (x_{lh}, y_{lh})$ for some j, l, h; and 4) for any (e, e'), where $e' = e + \frac{\rho_\gamma}{\rho} s_{\gamma,i}$, such that $e \geq t_{rqst} \wedge e \leq t_i \wedge e' \leq t_{i+1} \wedge (e, e') \subseteq (a_{ij'}, b_{ij'}) \cap (x_{l'h'}, y_{l'h'})$ for some j', l', h', we have $u_i \geq e$. We call such a slot satisfying the *latest in-time condition*. If no such slot exists, we then use the earliest possible slot.

A side effect of the latest in-time allocation is that it can reduce the buffer size at the client. If we use the earliest possible allocation, a client may have to prefetch and store quite a lot of segments in its local buffer. Though we consider it is not a big problem for today's desktop or laptop computers, we may have to consider the buffer constraint for those resource-limited clients, such as mobile handheld devices. The latest in-time allocation can be useful in these situations. Further, the scheduling algorithm can be easily extended to accommodate the buffer size of clients as a constraint, and find a slot allocation satisfying it, at the cost of having a possibly longer initial latency.

C. Delivery from Peer Servers to a Client

After the coordinator reserves slots for all segments, it is the client's task to make the request to peer servers. Even if a slot is confirmed, the client may fail to make the request at the beginning of the slot. Also possible is that a client may quit in the middle of playout. The task of garbage collection for those unused slots is performed by replica servers. To detect whether a client quits or not, we require the client periodically send a report message to the replica server. A timeout value δ is used. If a client does not show up δ after the beginning of the slot, or does not report to the replica server for δ during the streaming process, the replica server will recollect the slot reserved for this client and stop streaming data to it.

During the delivery of a segment from a peer server to a client, packets may be lost during transmission. The replica server can only find this information with the help from the client. For example, we can let the client send back acknowledgements to the replica server. In the request sent by the client to the replica server, it will include the scheduled playout time of this segment. Based on this playout time, the server can determine whether it is meaningful to retransmit the lost packet. Only if the retransmitted packet can arrive at the client before its playout time, the replica server will retransmit the packet, subject to the availability of bandwidth.

Another problem that may occur in transmission is the fluctuation of bandwidth along the path from the replica server to the client. This can be observed by the feedback from the client. An adaptive transmission scheme may be used at the replica server to deal with the situation. If the loss rate is higher than a threshold, the replica server can reduce the sending rate; if the loss rate is lower than another threshold, it can increase the sending rate. Adaptation within the channel bandwidth is not a problem. If it needs to be higher than the channel bandwidth, the replica server needs to calculate the current reservation and usage of other active streams. It is possible that the channels are not fully reserved or other channels are not fully utilized. In these cases, the adaptive scheme can let the replica server send at a higher rate (up to the client bandwidth $B_c * \rho$) and finish the delivery of the segment in a shorter time.

If the stream is sent at a lower rate in the earlier part of a segment, we need to do some catch-up in the later part of the segment. The replica server can calculate whether it will have to extend beyond the reserved slot. If there are channels not reserved after the current slot and the delivery of the content is still before the scheduled playout, the replica server can negotiate with the client to see whether the client channel is available and use the time after the current slot to finish the delivery. If these conditions are not met, the replica server has to get help with encoding techniques to reduce the rate so that it can finish in time. We will not go further with the discussion in this direction.

The communication protocol and the scheduling algorithm are based on CBR videos. To deal with VBR videos, we have two possible solutions. The first solution is based on the observation that we are not delivering live video in real time. Rather we expect that the client will prefetch quite a portion of videos and then play them. Therefore, we can send the VBR videos at a constant channel rate. For each segment, we may need to delay a certain time before playout so that each frame will arrive at the client before its playout time. This delay can be pre-calculated and has to be added to d_i. The final initial latency will be increased because of the VBR videos. The second solution will need a different reservation scheme. Instead of reserving slots from the constant bandwidth channels, we need to reserve a certain bandwidth for a period of time from peer servers based on the characteristics of the VBR video. This can work at a finer granularity, but will make the management of peer servers more complicated. The tradeoff is left for further study.

In the algorithm description, we assume that the clocks of the replica servers are synchronized. In reality they may be skewed. The simple solution is to use NTP [7] with the assumption of symmetric links. We design a method that can also deal with asymmetric links. In the third step of Section IV-A, when the replica server (R) sends the reply to the coordinator (C), it puts a timestamp (t_R) in the message. When the coordinator C receives the message, it also records the time (t_C) on its local clock. If the one-way latency from replica server R to coordinator C is ℓ, the accurate (or perfect) clock synchronization function mapping the clock (t) at the replica server to the clock at the coordinator should be $g_{acc}(t) = t + (t_C - t_R) - \ell$.

When the replica server streams the segment to the client, it also invokes one way latency ℓ, assuming that the client is co-located with the coordinator. The observation is that if the clocks of those replica servers are perfectly synchronized with the coordinator, the time shift behind the scheduled time will be different for different segments. This may lead to clients not being able to play out successive segments smoothly. Therefore, to compensate these differences, we should make the slots reserved at those replica servers earlier. The amount should be the one-way latency from each replica server to the coordinator. Therefore, in order to make the replica server send the segment ℓ second earlier, we have to change $g_{acc}(t)$ to $g(t) = g_{acc}(t) + \ell = (t + (t_C - t_R) - \ell) + \ell = t + (t_C - t_R)$. We should apply this function to the slots Ω_i's reported from replica servers. When we get slots (u_i, v_i) from the scheduling algorithm, we should use its inverse function $g^{-1}(t) = t - (t_C - t_R)$.

V. Performance Evaluation

We evaluate the performance of the *fine-grained* peer sharing and its protocol and scheduling algorithm by simulation. Each client is assumed to have B_c (with the default value 4 unless specified otherwise) channels for receiving media files, and every replica server has B_s (default value 20) channels to transmit data. Each server can store 50 video files, and each file is $\mathcal{T} = 5115$ second MPEG video (with size $\mathcal{L} = \rho\mathcal{T}$). Each peering group consists of m (default value 40) replica servers. So the total storage size of the group is $m * 50$ files. In the *fine-grained* scheme, a file is divided into n (default value 10) segments. In the *simple sharing* case, the whole file is always distributed to the server currently having the smallest sum of popularity rankings of files already on it, so long as the server has enough available storage. Request arrivals are modeled as a Poisson process and the file popularities are approximated by a Zipf-like distribution with $\alpha = 0.733$ [6]. We assume requests are equally likely originated from any client. The highest possible request arrival rate the system can support is $\Lambda_{max} = \frac{B_s \times m}{\mathcal{T}}$. That is the case in which *all* channels of *all* servers are kept busy *all the time*. The actual arrival rate Λ in our simulation is set as a percentage of Λ_{max}. Every experiment simulates a time period of $\mathcal{T} * 25$ second long.

The performance measures we study include: 1) *Initial latency*. It is the average interval between the time a client makes a request and the starting time of playout. 2) *Rejection rate*. It is the ratio of the number of rejected requests over the total number of requests.

Figure 4 shows the initial latency of the *simple sharing* and the *fine-grained* schemes, expressed as the percentage of the video length. As the request rate increases, the difference between the initial latencies of the *simple sharing* and the *fine-grained* schemes becomes much larger. When the request rate is 60% of the system capacity, the initial latency of the simple sharing is almost 22% of the video length, i.e., more than 18

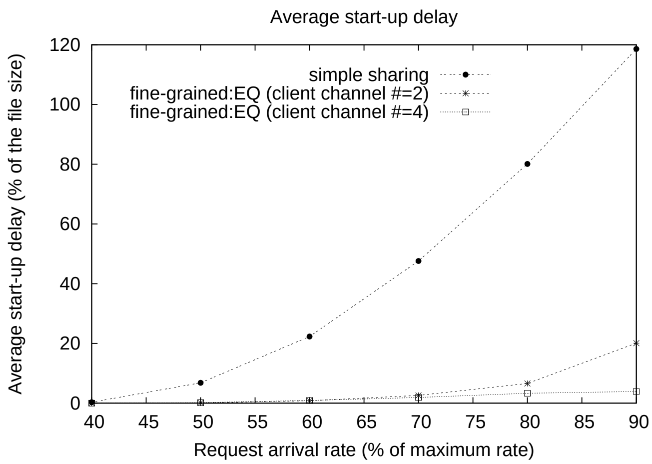


Fig. 4. Average latency using different approaches

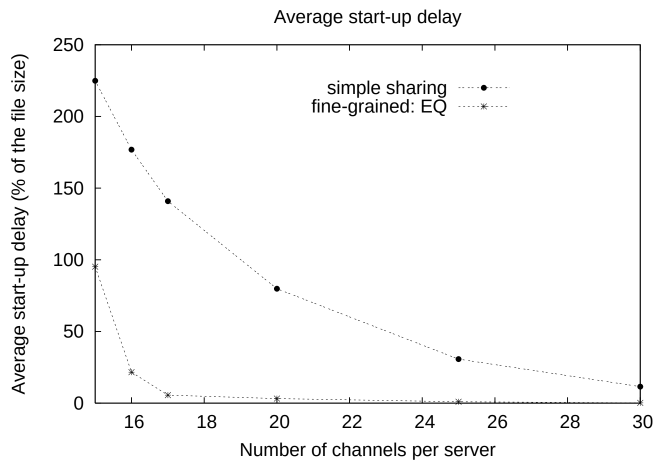


Fig. 6. Access latency when the number of server channels varies

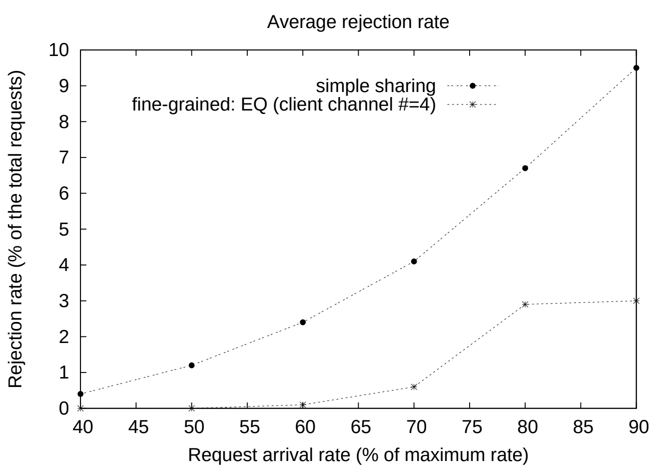


Fig. 5. Rejection rate using different approaches

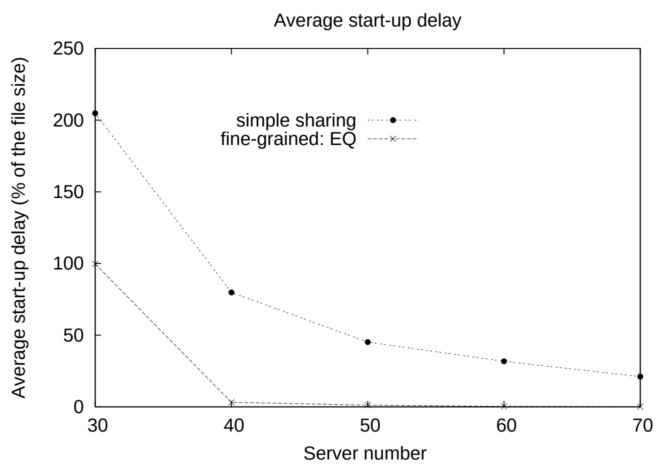


Fig. 7. Server number

minutes, while the initial latency of the fine-grained scheme is still less than 5 seconds.

Figure 5 plots the rejection rate when the admission control is applied. Specifically, if a coordinator finds that the initial latency is greater than a threshold $T_{thresh} = \mathcal{T}/10$, the request will be rejected. Once a request is rejected, the resource allocated to it will be released. We note that a much lower rejection rate can be achieved in the *fine-grained* scheme. For example, when the request rate is 70% of the system capacity, the rejection rate of the *fine-grained* scheme is about 0.6%, while the rejection rate of the *simple sharing* scheme is as high as 4.1%.

Figure 6 shows how the access latency is affected when the number (B_s) of server channels varies. In this and the next experiments, we fix the request rate at 80% of max rate under the default setting, i.e., $m = 40$ and $B_s = 20$. When the number of channels each server has increases from 15 to 30, the average delay sharply decreases to close to 0 by the fine-grained sharing, while the simple sharing still has a significant delay even when each server has 30 channels.

Figure 7 shows the effect when the server number changes from 30 to 70 and each server has 20 channels. Once again, the start-up delay in the fine-grained sharing scheme quickly reaches 0 when the number of servers increases. The simple sharing scheme can shorten the start-up delay when the number of servers increases. However, it still generates significant delay even when the number of servers increases to 70.

In the fine-grained sharing schemes, segments of a file are distributed to different replica servers. The requests to popular files are better served because the load is more evenly distributed to those servers. Figure 8 presents the load on servers over the whole simulation process. The x-axis is the server id. The y-axis is the seconds of video each server sends to clients. In the simple sharing, the load on servers varies a lot, from as high as 2.5 million seconds to 1.13 million seconds. In contrast, the load is more evenly distributed in the fine-grained sharing case. The highest load is within 10% more than the lowest load.

VI. Related Work

Streaming multimedia content over the Internet has been studied extensively [8]. Proxy caches have been used to

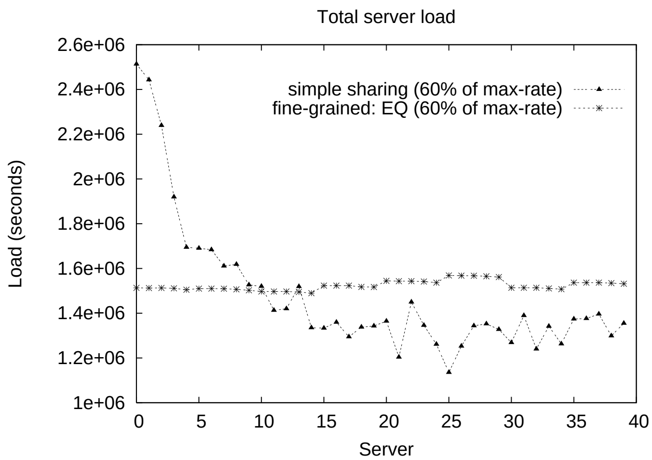


Fig. 8. Load on Servers

reduce the backbone bandwidth requirement for scalable video delivery [9]. Recently, prefix caching has been proposed to reduce the storage requirement for the proxy servers [4]. Only the prefix part is delivered to replica servers while the suffix part is kept at the origin server and usually depends on multicast for its delivery. Our approach divides videos into segments and not only the prefix part but the suffix parts are delivered to replica servers as well.

The peer sharing technique was originally used in peer-to-peer systems [10], [11]. However, the unit of sharing is a complete file. We propose a fine-grained file sharing technique and the sharing unit is a segment of a media file. This balances the load on replica servers and improves the capacities of the system for serving a giving set of media files.

Segmentation of large media files has been used in large video servers and video broadcast systems [12]–[14]. The data striping technique has been used in media servers to improve the disk access speed and reliability [13]. It allows multiple disk transfers to occur in parallel. Our scheduling algorithm is more complicated because it requires communication between replica servers. It is executed by each replica server and only involves those replicas having segments of the requested file.

VII. Concluding Remarks

In this paper we presented a novel fine-grained peer sharing technique for streaming large media files in the context of content distribution networks. Our scheme does not depend on the availability of multicast deployment. Our design employs peer-to-peer sharing at a finer granularity to improve the flexibility of peering replica servers to handle client requests. We designed a communication protocol and a scheduling algorithm to achieve the goal of reducing the initial latency and the rejection rate. The simulations demonstrated a significant performance gain of the fine-grained peer sharing technique.

References

[1] J. Kangasharju, J. Roberts, and K. W. Ross, "Object replication strategies in content distribution networks," in *Proceedings of Sixth International Workshop on Web Caching and Content Distribution (WCW'01)*, June 2001, pp. 39–53, Boston, MA.

[2] M. Yang and Z. Fei, "A fine-grained peer sharing technique for delivering large media files over the internet," in *Proceedings of Eighth International Workshop on Web Content Caching and Distribution (WCW'03)*, September 2003, Hawthorne, NY.

[3] L. Kleinrock, *Queueing Systems, Volume 1: Theory*. John Wiley & Sons, 1975.

[4] S. Ramesh, I. Rhee, and K. Guo, "Multicast with cache (mcache): an adaptive zero-delay video-on-demand service," in *Proceedings of IEEE Infocom'01*, 2001.

[5] M. Yang and Z. Fei, "A segmentation-based approach to delivering multimedia files in content distribution networks," in *Proceedings of the IEEE International Conference on Communications (ICC 2004)*, June 2004, Paris, France.

[6] D. Luparello, S. Mukherjee, and S. Paul, "Streaming media traffic: An empirical study," in *Proceedings of Sixth International Workshop on Web Caching and Content Distribution (WCW'01)*, June 2001, Boston, MA.

[7] D. L. Mills, "Network time protocol," *RFC 1305*, 1992.

[8] T. Kim and M. Ammar, "A comparison of layering and stream replication video multicast schemes," in *Proceedings of NOSSDAV'01*, 2001.

[9] Z.-L. Zhang, Y. Wang, D. H. Du, and D. Su, "Video staging: A proxy-server-based approach to end-to-end video delivery over wide-area networks," *IEEE/ACM Transactions on Networking*, vol. 4, no. 8, pp. 429–442, August 2000.

[10] Napster Inc., "http://www.napster.com."

[11] KaZaA, "http://www.kazaa.com."

[12] J. R. Santos and R. Muntz, "Comparing random data allocation and data striping in multimedia servers," in *Proceedings of ACM Sigmetrics*, June 2000, Santa Clara, CA.

[13] A. L. Drapeau, P. M. Chen, J. H. Hartman, E. K. Lee, E. L. Miller, K. Shirriff, S. Seshan, R. H. Katz, G. A. Gibson, and D. A. Patterson, "RAID-II: A high-bandwidth network file server," in *Proceedings of 21st International Symposium on Computer Architecture*, April 1994, Chicago, IL.

[14] L. Gao, J. Kurose, and D. Towsley, "Efficient schemes for broadcasting popular videos," in *Proceedings of NOSSDAV'98*, 1998.

Zongming Fei is an Assistant Professor in the Department of Computer Science at the University of Kentucky. He received his Ph.D. in computer science from Georgia Institute of Technology in 2000. His research interests include network protocols and architectures, overlay networks, mobile computing, and multimedia systems.

Mengkun Yang is a Ph.D. student in the Department of Computer Science at the University of Kentucky. She received her M.E. and B.E. in computer science from Sichuan University, China in 2000 and 1997, respectively. Her research interests include overlay networks, peer to peer systems, and multimedia systems.