

Video Management in Peer-to-Peer Systems

Ying Cai

Zhan Chen

Wallapak Tavanapong*

Department of Computer Science

Iowa State University

Ames, Iowa 50011, USA

Email: {yingcai, zchen, tavanapo}@cs.iastate.edu

Abstract

Providing scalable video services in a peer-to-peer (P2P) environment is challenging. Since videos are typically large and require high communication bandwidth for delivery, many peers may be unwilling to cache them in whole to serve others. In this paper, we address two fundamental research problems in providing scalable P2P video services, namely (1) how a host can find enough video pieces, which may scatter among the whole system, to assemble a complete video, and (2) given a limited buffer size, what part of a video a host should cache. We propose a new distributed video management technique. Our scheme organizes hosts into a number of cells, each of which is a distinct set of hosts which together can supply a video in its entirety. A client looking for a video can stop its search as soon as it finds a host that caches any part of the video. Caching operations can be coordinated within each cell to balance data redundancy in the system. Our extensive study on a Gnutella-like simulation network shows convincingly the performance advantage of the new scheme.

1. Introduction

Unlike text or image files, streaming a video to a remote client takes a significant amount of communication bandwidth. A video server typically can sustain only a very limited number of concurrent video streams. This problem, known as *server or network-I/O bottleneck*, limits the scalability of video services. To improve server throughput, one can leverage IP multicast to allow multiple clients to share a server stream [3][5]. Unfortunately, the deployment of such facility beyond local area networks has been shown to be difficult. Alternatively, the server can ask a client to buffer

and forward its incoming video stream to serve other clients. Such client forwarding mechanism achieves the effect of using IP multicast in the sense that a server can use one stream to serve many clients simultaneously. Therefore, it is called *Application-Layer Multicast (ALM)* [9][1], although each video stream is actually unicast. By leveraging client computing resource, ALM can effectively reduce the workload of the source server. However, this approach places high demand on client bandwidth. In order to serve other clients, a client needs to have a bandwidth of at least two times the video playback rate, one for downloading and the other for forwarding. Such requirement may cause many clients accessing today's Internet unable, or unwilling, to participate in data forwarding.

For video distribution over today's Internet, where the deployment of IP multicast has been slow and especially, the receiving ends are in vastly different network domains, the concept of *Peer-to-Peer (P2P)* video sharing provides another means of tackling the server bottleneck problem. The idea put in a simple way is to allow hosts to share their videos directly. In a P2P video system, a host can be served by any other host that has the video it requests. Later this host can supply the video data it caches, if any, to serve future requests. This service model is different from ALM in taking advantage of client computing resources. In ALM, a client forwards an on-going video stream to serve other clients. Besides the high bandwidth requirement, the client can only contribute the time when it is downloading a video itself. After playing back a video, the client does not help further in distributing this video. In contrast, P2P video services amplify the serving capacity of a video server by *duplicating* its videos on its clients. When a client downloads a video from a server, the client can cache the video and serve the whole community, just like the original server of this video. Thus, a client does not have to forward its incoming video stream, while downloading it, in order to contribute in video services.

The strength of a P2P video system relies on the effective

* This work is partially supported by the U.S. National Science Foundation under Grant No. 0092914.

aggregation of communication bandwidth and disk space contributed by its participating hosts. Ideally, after a host downloads and plays back a video, it caches the whole video and becomes a supplier of this video. In reality, however, very few hosts are willing to retain a complete video and supply it back to the community. This is not just because a video is usually very large in size, but also because serving a video request takes a significant amount of communication bandwidth, which seems to be the major concern of most users. Most likely, a user wants to use the bandwidth for his/her own interest rather than serving other peers. As reported in [8], 25% of hosts in Gnutella are simply *free-riders*, i.e., they provide no data back to the community. In the remaining 75% of hosts, 7% of them offer more data than all of the other hosts combined. It also shows that most hosts contribute less than 1GB of its disk space. Thus, even for those who keep a video in its entirety, they will recycle the occupied disk space soon after they start to download other videos.

Apparently, a P2P video system cannot simply rely on the few hosts that cache videos in their whole to serve all video requests. Otherwise, it will create the server bottleneck problem just like in a central server architecture. To materialize the advantage of P2P computing, a host should be allowed to participate in video services as long as it caches some amount of video data, instead of a whole video. For this purpose, a video management technique must be in place to address the following two problems:

- *Video lookup*: How can a host find the video pieces that can complement each other to make a complete video? Such video lookup could be expensive since the hosts caching video data may scatter around the whole network. The search scope of a video lookup is dictated by its requester's distance to the host that holds the last missing piece of the video.
- *Video caching*: For a host with a limited buffer size, which part of the video should it cache? This is another crucial question. If every host caches or deletes video data in the same sequence, e.g., always from the beginning or the tail of a video, then a video could become extinct from the community, even though a large amount of video data is still being retained in the system.

A possible solution to the above problems is using some super node for centralized video management. The super node maintains a list of hosts and the information about the video data they cache. To retrieve a video, a host contacts the super node, which then gives a list of candidates who can supply the video. Each host also negotiates with the super node about what data to cache or delete. This Napster-like solution, however, is not scalable. When the system involves a large number of hosts and videos, maintaining the

caching status for each video, which can be updated very frequently, imposes an overwhelming workload on the super node and can easily bring it kneel down. This architecture also presents a single point of failure. Although fault tolerance may be achieved by deploying a set of geographically distributed management nodes, complicated consistency checking may be required. Also, the management-related network traffic will further increase because every caching operation performed in the system would require to contact and update multiple places.

In this paper, we assume a pure P2P system without any global or regional super nodes and present a fully distributed video management technique. The key element of our technique is the concept of *cell*, which is defined to be a cluster of hosts which together can supply a complete video. Each cell is created dynamically and managed individually. The advantages of our technique are twofold. First, a host requesting a video can locate a complete set of video pieces from its *nearest* host that caches some part of the video. Thus, the search scope is dramatically reduced. Second, caching video data can be coordinated at the cell level to balance data redundancy in each cell. Likewise, when deleting video data, the most redundant data will be expunged first. While such coordination maximally protects the video integrity of a cell, it causes very minimal communication overhead because our scheme splits a cell whenever possible to limit its size, i.e., the number of its member hosts. Although the proposed technique is intended for video management, it can also be used in regular unstructured P2P systems for distribution of large files such as software packages.

The remainder of this paper is organized as follows. We present our Cell technique in Section 2. In Section 3, we present our simulation study. We discuss more related work in Section 4 and then give our concluding remarks in Section 5.

2. Proposed Technique: Cell

2.1. System Design Goals

We assume a P2P system in which the participating hosts are heterogeneous and fully autonomous. When a host receives a video service, it can cache some data and supply the data upon request to other hosts in the system. Each host decides on its own the amount of disk space to contribute and can change its contribution at any time. In other words, a host assumes no obligation in keeping any data it caches. Even a seeding host, which provides the first copy of some video to the system, can recycle the occupied space whenever it deems necessary. For video management in such a system, the following two performance metrics are of concern:

- *Video Accessibility*: The number of hops that a host has to search in order to find a set of peers that can complement each other to supply a complete video.
- *Video Availability*: The number of *distinct* sets of peers that can provide a complete video.

We note that video accessibility determines the cost of video lookup. A higher accessibility means a smaller system scope needs to be searched in order to assemble a complete video. Video availability, on the other hand, measures the effectiveness of a video caching solution. It is a measure of video redundancy in the whole system and also a reflection of the system capacity of serving video requests. Because each supplying set of peers can provide a complete video independently, maintaining as many distinct sets as possible not only makes the supplying of a video more fault tolerant, but also reduces the chance of creating server bottleneck. Obviously, a good video management technique should be able to maximize both video accessibility and availability.

2.2. Cell Overview

Without loss of generality, we consider only one video. In our solution, we partition the video into n segments, S_1, S_2, \dots , and S_n , each having the same size in storage. The operations on video data, such as caching and deleting, are performed segment by segment. We will call a host a *caching host* if it caches any part of the video; otherwise, it is a *non-caching host*. We say a set of video segments is complete if they can make up a complete video.

Our main idea is to group the caching hosts into *cells* based on the segments they cache. Each cell is a cluster of coordinated caching hosts that together can supply a complete set of video segments. A cell may contain a single host if the host caches the whole video. For instance, a seeding host by itself may be a cell. Each cell is associated with a binary relation, called *CacheTable*, which tracks the cell members and the corresponding data segments they cache. Each row of the *CacheTable* is a tuple of (h, s) , where h represents a member host and s denotes a segment cached by this host.

By organizing caching hosts into cells, the video lookup cost can be dramatically reduced for two reasons. First, to look for a complete set of video segments, a client just needs to locate a cell by finding a caching host. Thus, the search scope of a video lookup is now determined by the client's distance to its *nearest* caching host, instead of the *furthest* one that caches the last missing segment. Second, because searching for a video is now simplified as searching for any part of the video, the operation of searching video v , say $Search(v)$, can be implemented using any advanced P2P file search algorithms. When a caching host receives a video

query, it can immediately return its cell information, from which the query sender can find a complete set of video segments.

In addition to reducing video lookup cost, cell organization also makes it possible to coordinate video caching at cell level to balance segment redundancy. For example, in Figure 1, when host H_6 downloads data from $cell_1$ and is willing to cache one segment, it can cache either S_2 or S_3 . Likewise, deleting video segments can also be coordinated at the cell level. In Figure 1, if host H_3 needs to delete two segments, it will delete S_1 and S_4 , since these two segments are also cached by other two members in the cell. As we will see shortly, a cell usually is very small in size (i.e., the number of its members). Thus, the coordination of video caching within one cell does not incur much computation and communication overhead. Yet, such coordination can balance the cache redundancy in a cell and maximally protect a cell's integrity in terms of supplying a complete set of video segments.

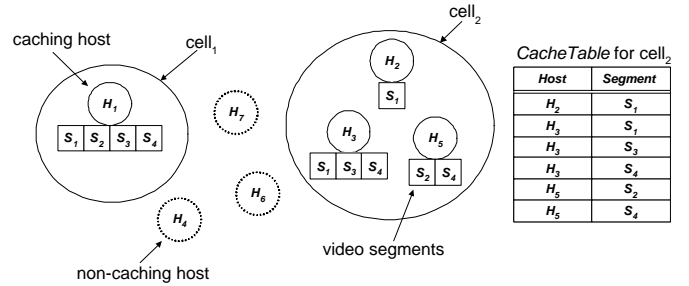


Figure 1. Cell Example

2.3. Cell Operations

Cache: Initially, the system has only one cell, from which all clients download video data. To request a video v , a client can call $Search(v)$, which may return several caching hosts belonging to different cells. Since each cell can independently provide a complete video, the client can select any cell as its service provider. Alternatively, it can choose to download data from the members in different cells, based on their current available bandwidth or underlying physical network topology to balance link stress. For simplicity, we consider a cell as a serving unit in this paper. Before downloading video segments from a cell, the client retrieves its *CacheTable* from the cell's member found by $Search(v)$. A host h served by a cell c can become a member of a cell by caching some video data. To cache n segments, it calls the following $Cache(n, c)$ procedure:

$Cache(n, c)$

1. Retrieve $c.CacheTable$ from the cell that serves the host;

2. Check $c.CacheTable$ and for each video segment, calculate its *redundancy*, i.e., the number of hosts in the cell that cache this segment;
3. Cache the n least redundant segments;
4. For each newly cached segment, say s , add tuple (h, s) to $c.CacheTable$.
5. Update each cell member with the new $c.CacheTable$.

Split: When a host brings some new segments into a cell (e.g., calling $Cache()$ procedure), the host needs to check if the cell can be split. In our implementation, we split a cell if its members can be grouped into two distinct subsets, each can provide a complete set of video segments. There are two reasons to split a cell whenever possible. First, keeping a cell as small as possible minimizes the size of $CacheTable$ and reduces its management costs, such as cache coordination. Second, because each cell is an independent video supplier and each caching host belongs to only one cell, creating as many cells as possible reduces the chance of creating network bottleneck.

When host h in cell c caches some new segments, it calls $Split(c)$ procedure. A simple way to check if a cell can be split is to try all possible combinations of its members. This approach, however, may require intensive computation. Given a cell with k hosts, totally there are 2^{k-1} different splits. In contrast, the heuristic algorithm in our following $Split()$ procedure takes only $O(k^2)$ computation, where k is the number of members in a cell.

$Split(c)$

1. Make a copy of $c.CacheTable$ and name it $Table_r$;
2. Create a new empty cache table and name it $Table_l$;
3. Set $RemoveMore$ to be *true*;
4. Repeat the following steps until $RemoveMore$ becomes *false*:
 - Check each host listed in $Table_r$, mark it if all segments it caches are also cached by other hosts listed in the table;
 - If no host is marked, set $RemoveMore$ to be *false*;
 - Otherwise, perform the following steps:
 - For each marked host, calculate the *benefit* of including it in $Table_l$ by checking each of its cached segments:
 - * If the segment can be found in $Table_l$, decrease *benefit* by 1;
 - * Otherwise increase *benefit* by 1;
 - Recruit the host with the largest *benefit* by moving all its tuples from $Table_r$ to $Table_l$;
 - Unmark all marked hosts;
 - If $Table_l$ has contained all segments of the video, set $RemoveMore$ to be *false*;
5. If $Table_l$ contains a complete set of video segments, then split the cell as follows:
 - Discard $c.CacheTable$;
 - For each host listed in $Table_r$, replace its $CacheTable$ with $Table_r$;
 - For each host listed in $Table_l$, replace its $CacheTable$ with $Table_l$;

6. Otherwise, discard $Table_r$ and $Table_l$.

Delete: When a host h in cell c needs to delete n segments, it calls the following $Delete(n, c)$ algorithm to delete the most redundant segments first:

$Delete(n, c)$

1. Check $c.CacheTable$ and for each video segment cached by this host, calculate its redundancy;
2. Delete the n most redundant segments;
3. For each deleted segment, say s , delete tuple (h, s) from $c.CacheTable$;
4. Update each cell member with new $c.CacheTable$.

Merge: A cell is regarded broken when it cannot provide a complete set of video segments. This happens when a cell member deletes some non-redundant segments or puts itself off-line. A broken cell can be found either by the host who deletes data, or by a host who tries to download video segments from the cell. When a host finds a broken cell, it calls $Search(v)$, the same procedure used for video search. The cells found during this search will be used to house the remaining members of the broken cell. We call this process as *merge*. Now the problem is, given a caching host and a list of cells, which one should this host join? One consideration is to balance cache redundancy in a cell. We say a segment is *redundancy- i* in a cell if it is cached by i members of the cell. For each segment cached by this host, we can check its redundancy in each cell. A cell is selected if it has the largest number of redundancy-1 segments cached by this host. If two cells have the same number of redundancy-1 segments cached by this host, we select the cell with the larger number of redundancy-2 segments cached by this host, and so forth.

To merge a broken cell bc with a list of cells $cList$, we call the following $Merge(bc, cList)$ procedure. Again, when a new member joins a cell, the cell acquires more segments and this may result in a split.

$Merge(bc, cList)$

1. For each caching host h in the broken cell bc , perform the following steps:
 - Call $Select(h, cList)$ to find a cell, say c , from $cList$ to accommodate host h ;
 - Make host h a new member of cell c :
 - For each segment s cached by host h , add a tuple (h, s) to $c.CacheTable$;
 - For each host listed in $c.CacheTable$, update it with new $c.CacheTable$;
 - Call $Split(c)$;
 - If any new cell is created, append it to $cList$.

$Select(h, cList)$

1. Set $r = 1$;
2. Repeat the following steps until only one cell remains in $cList$:
 - For each cell in $cList$, say c , mark its segments whose redundancy in the cell is r ;

- Let $benefit_c$ be the number of redundancy- r segments in cell c that are cached by host h ;
 - Remove cell c from $cList$ if its $benefit_c$ is not the largest;
 - Increment r by 1;
3. Return the cell in $cList$;

2.4. Implementation Issues

In this subsection, we discuss some implementation issues of Cell technique. To efficiently retrieve the list of data segments cached by a cell member in the *CacheTable*, or vice versa, we can hash or build a B⁺-tree index on each field of the *CacheTable*. There are also other options such as storing the data of the *CacheTable* in two adjacency matrices. A cell's *CacheTable* can be replicated among all its members so that it can be located through any member of the cell. When a member updates the table, it propagates the update to other members in its cell.

To avoid concurrent updates on *CacheTable*, some mutual exclusion mechanism should be used. There are many such approaches. For instance, we can use the *majority quorum* algorithm [11], which works as follows. Before a host enters a critical section, it needs to other members in its cell and get a majority of them to approve. A host that has issued permission will deny future requests until the previously approved host exits the critical section. As a cell usually contains only a few members, this simple approach can be applied without much overhead. Alternatively, we can use some optimistic control mechanism to allow concurrent updates on the table. In the worst case, multiple members delete the same segment, causing the cell to be broken. When this happens, a merge operation can be invoked (e.g., by a client requesting a download, etc.). We also notice that *Merge* and *Split* operations can be expensive when cells are large in size. Our simulation shows that a cell usually contains only a few members, except in some rare cases in which each host contributes only tiny disk space. To avoid frequent invocation of such operations, we can introduce some data redundancy to cells. For instance, we may choose not to split a cell unless the redundancy of each segment in the cell exceeds some threshold.

In our split algorithm, a cell is split based on the video data cached by its members. Such split is suitable when users do not require to play a video while downloading it. We note that in today's P2P video sharing systems such as Kazaa, users usually download videos in the background first and examine their content later, because for most users, the underlying network simply cannot support them to stream a video at its regular playback rate. According to [4], 30% of downloads of small objects (less than 10MB) take over an hour, and 10% take nearly a day. For large objects (more than 100MB), 50% of downloads take more than a day and nearly 20% of users take more than

a week to complete. It is possible, however, to configure our cell to support play-while-downloading when the bandwidth is available. For example, we may choose not to split a cell unless the outbound bandwidth of the hosts in each new cell can be aggregated to deliver their cached video at its regular playback rate. We may also take other factors into consideration, such as network topology, when such information can be obtained, say using topology probing [6]. In this case, the hosts that are physically close to each other may be grouped into a cell. We leave these options for future investigation.

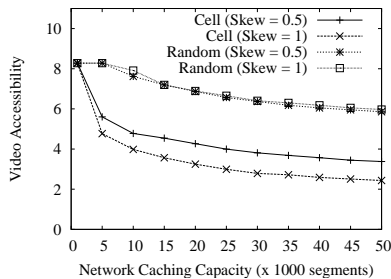
3. Performance Study

In this section, we evaluate the performance of the proposed Cell scheme. To evaluate its effectiveness on improving video *accessibility* and *availability*, we compare it with a baseline approach in which each host caches and deletes video data without collaboration. To cache k segments of a video, a host can have several options. For instances, it can cache first k segments, last k segments, or just randomly choose k segments. Similar options are available when deleting segments. The first two options are commonly used for proxy server to cache video data. However, they will perform poorly here because they cache data bially, preferring only certain portion of a video. Thus, we choose to implement the third option, i.e., each host caches and deletes video segments randomly. We will simply refer to this baseline implementation as *Random*.

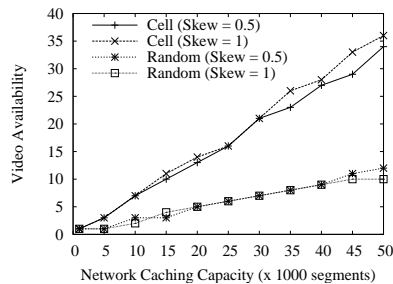
3.1. Simulation Model

To facilitate our performance study, we have implemented a Gnutella-like P2P network simulator. The network in our simulation consists of 10,000 hosts, which we believe is large enough for us to see the performance trend of the two techniques. In our simulation, each host connects to at least 3 other hosts and has about 7 neighbors on average. Roughly, we simulate a network with 15-hop distance. We consider only one video and partition it into 1000 segments, each has an equal size of 1MB. The host caching capacity is assumed to be skewed, as indicated in [8], and follows a Zipf distribution. We consider two different skews, 0.5 and 1.0, where a higher skew means more hosts have less caching capacity. In our implementation, each caching host uses two equal-size buffers, one holding the segments cached by using Cell and the other for the segments cached by using Random. This allows us to test the two techniques simultaneously and capture their exact performance difference.

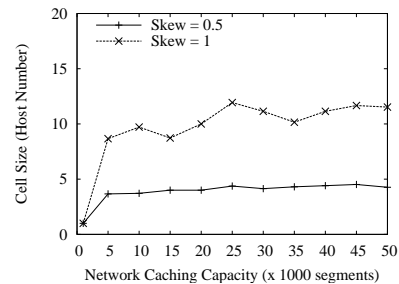
When collecting performance data, we take a snap shot of the entire network. Then for each host, we determine its video accessibility, i.e., the minimum TTL value a Gnutella



(a) Effect on Video Accessibility



(b) Effect on Video Availability



(c) Cell Size

Figure 2. Caching Phase

query needs to set in order to search a complete video, and calculate their average under two techniques. We use the number of cells created by Cell approach as its video availability. For Random scheme, we treat all caching hosts as one big cell and then split it recursively using our cell split algorithm. The number of cells it generates is then reported as the video availability under this approach.

3.2. Simulation Results

Our simulation consists of three distinct phases: *caching*, *caching-and-deleting*, and *deleting*. We explain them as follows and examine the performance of the underlying techniques accordingly.

Caching Phase: We start this simulation by randomly choosing a host to seed the first copy of the video. Then each time we randomly choose a non-caching host and make it generate a video request. After its request is satisfied, it caches a number of video segments corresponding to its given caching capacity. This process is repeated until *network caching capacity*, i.e., the number of segments cached by the hosts in the entire system, reaches $50 * 1000$ segments (i.e., 50 copies of the video).

Figure 2(a) shows the video accessibility under the two techniques with two different skews. In the beginning, the video accessibility is the same under both techniques - averagely each host is about 8 hops away from the seeding host. As hosts start to request the video and spread out video data, the two curves for Cell drop very quickly. For example, when the skew is 1.0 and the network caching capacity increases from 1 to 10 copies of the video, the search scope for a video lookup is reduced from 8 to 4 hops on average. In comparison, under the same setting, Random can reduce the average search scope less than 1 hop. This can result in a huge difference in the lookup cost, because the number of hosts increases exponentially with respect to the number of hops. We note that this performance difference comes from the fact that in Cell, the search scope of a requesting host is determined by its distance to the nearest caching hosts,

while in Random, this is determined by the furthest host among those where it downloads the video from. The figure also shows that Cell performs even better when the host caching capacity is skewer. This is simply because with a fixed network caching capacity, there are more caching hosts when the average host caching capacity reduces.

We now look at the performance of the two techniques in maintaining video availability. Figure 2(b) shows that the video availability under both schemes is most influenced by the network caching capacity and little by the skew factor. However, Cell performs much better than Random and their performance gap becomes larger as network caching capacity increases. When the aggregated network caching capacity can keep five copies of the video, our proposed scheme creates more than two times the number of cells than Random does. As the caching capacity increases to 50, the performance difference between Cell and Random is about three times. This is not surprising because the cell-level caching coordination in our scheme allows a host to cache the less redundant segments with a higher priority. It is worth mentioning that the computation and communication is incurred only when some host caches or deletes video data or finds a broken cell; and when this happens, such cost is low because the number of members within a cell is usually small. As Figure 2(c) shows, the average cell size is less than 5 when the skew is 0.5. When the skew increases to 1, the average cell size is still maintained below 15. We note that maintaining as many cells as possible not only makes the video service more fault-tolerant, but also reduces the chance of creating network bottlenecks. As a result, the overall system serving capacity increases.

Caching-and-Deleting Phase: After the caching phase, we continue with caching-and-deleting phase. This phase consists of a series of operations, each of which consists of two steps: caching and deleting. For each operation, we first randomly select a non-caching host to generate a video request and then cache some video segments. Then we randomly choose a caching host and make it randomly delete

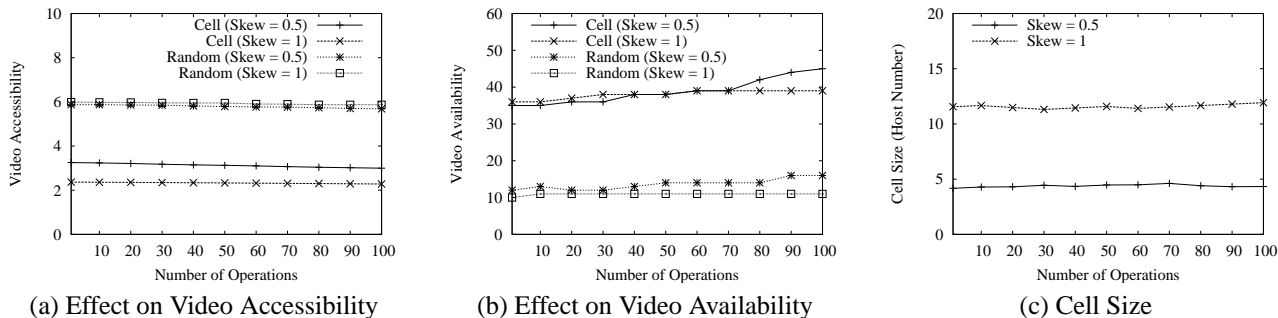


Figure 3. Caching-and-Deleting Phase

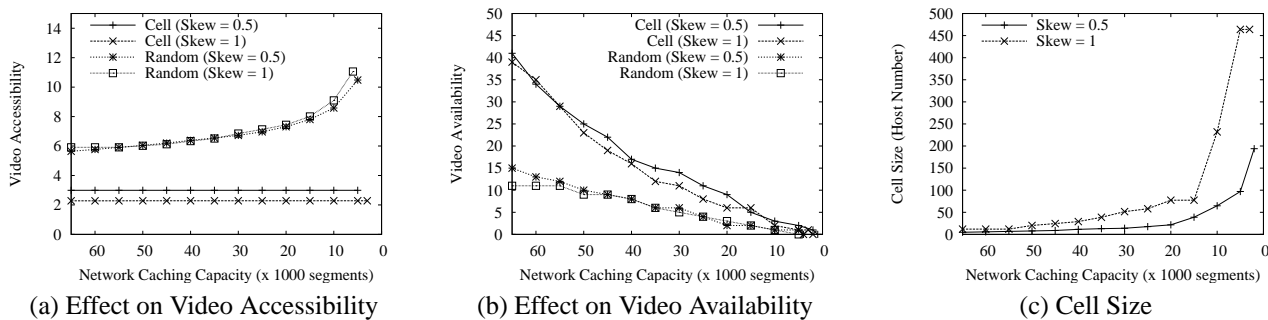


Figure 4. Deleting Phase

some video segments, from 1 to the number of segments it caches. As shown in Figure 3(a) and (b), both Random and Cell are quite stable in maintaining their performance. Their performance changes are hardly noticeable. The curves for the average cell sizes are also flat, indicating that Cell is very robust in maintaining cell integrity.

Deleting Phase: We continue the caching-and-deleting phase with a deleting phase. During this period, each time we randomly choose one caching host and make it delete some video segments, the number of which is randomly generated between 1 and the total segments cached by this host. The deleting process is repeated until the network cache capacity is reduced to 0. Figure 4(a) shows the video accessibility under the two schemes. An interesting phenomenon is, the curves for Cell are nearly flat, regardless of caching skew. This can be explained as follows. In Cell, unless a caching host deletes all its data, its deletion does not reduce video accessibility because it is still a gateway to a cell. In our simulation, each deletion most likely causes a host to delete just some part of its cached data. As a result, the video accessibility under Cell is not changed noticeable until at the end, all caching host has very few segments remain and every deletion turns one of them into a non-caching host. In comparison, Random performs worse as the network shrinks its cache capacity. In this scheme,

a host has to expand its search scope as long as any one of its supplying peers deletes a critical segment. The figure shows that the video accessibility under Cell with both skews is more than 100% better than that under Random. Again, this is very significant when we translate the search scope into the actual lookup cost.

The video availability under both schemes reduces when the network cache capacity shrinks, as shown in Figure 4(b). The figure again indicates that this performance metric is not very sensitive to the caching skew. However, in all stages, the number of cells maintained by the proposed scheme is more than two times of that by Random. With cache coordination, a caching host always deletes redundant segments first. A cell is broken only when a member has deleted all redundant segments and needs to recycle more cache space. Thus, our scheme can maximally protect the video integrity in a cell. We note that the average cell size, as shown in Figure 4(c), increases very quickly when the network caching capacity becomes very low. This is simply because when most caching hosts can cache only a few segments, a cell needs to aggregate more hosts in order to provide a complete video. While our Cell technique allows a host to contribute any amount of disk space, maintaining a large number of hosts in a cell, each contributing only a tiny disk space, may cause significant overhead. To control the

cell size, we can either include in a cell only the hosts that have some minimum contribution. Alternatively, we can simply partition a video into a less number of segments.

4. Related Work

To the best of our knowledge, the problem of caching collaboration has not been studied in the context of P2P file sharing. One related work is CFS [2], a structured P2P system built on top of Chord [10] file lookup technique. CFS does not consider caching collaboration, but allows partial caching. This approach partitions a file into many segments and peers can cache some of these segments. CFS treats each segment as an individual file and requires n searches to assemble a file that is partitioned into n segments. Such search overhead may be acceptable for structured P2P systems because the overhead for searching one segment is small. However, the search cost for one segment in unstructured P2P systems is much more significant. Hence, large files can severely increase the search overhead in such systems.

Another related work is proxy caching collaboration [7] used in traditional video-on-demand systems. Proxy caching aims at reducing network communication and server workload. In contrast, there is no such central video server in our environment. Our caching technique is designed to maintain the existence of a video by balancing its caching redundancy in a system where the available storage capacity fluctuates dynamically. Another major difference is, proxy caching collaboration is performed on top of a set of static and pre-defined proxy servers while in our scheme, any host in a P2P system could be a caching host and the member hosts in each cell are selected dynamically. In addition, our scheme does not rely on a centralized coordinator for caching collaboration.

5. Concluding Remarks

With the continuous drop of storage price, one may assume that a regular desktop can have abundant disk space to hold any large files. However, network bandwidth, especially that of the *last-mile* connections, will remain constrained for a long time. The scarcity of this resource may make a user reluctant in caching a large video in whole to serve others. Rather, a user may be willing to retain only a small amount of video data in return for the services it receives from participating a P2P video system. For large-scale and cost-effective video services, a P2P video system must be able to attract a large number of participants and effectively manage the resource they contribute. For this purpose, we presented a novel video management technique, called *Cell*, by which a peer can contribute by

caching any amount of video data. In our approach, looking for a video is transformed into looking for a host that caches any part of the video. Since a host can always contact its nearest caching host for a complete video, the search scope of a video lookup is minimized. In addition to bringing a video closer to its requester, our technique also significantly enhances the availability of a video in the whole system through caching coordination. Within a cell, data caching and deleting are performed with consideration of data redundancy in the cell. Since a cell is usually very small in size, such caching coordination incurs little communication and computation overhead. By maintaining as many cells as possible, our technique makes a video more fault-tolerant and reduces the chance of creating network bottleneck.

References

- [1] Y.-H. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *Proc. of ACM SIGMETRICS*, pages 1–12, Santa Clara, CA, June 2000.
- [2] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of 18th ACM Symposium on Operating Systems Principles (SOSP01)*, pages 202–215, Alberta, Canada, 2001.
- [3] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling Policies for an On-Demand Video Server with Batching. In *Proc. of ACM Multimedia*, pages 15–23, San Francisco, California, October 1994.
- [4] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 314–329, Bolton Landing, NY, June 2003.
- [5] K. A. Hua, Y. Cai, and S. Sheu. Patching: A Multicast Technique for True Video-on-Demand Services. In *Proc. of ACM Multimedia*, pages 191–200, Bristol, U.K., September 1998.
- [6] A. Nakao, L. Peterson, and A. Bavier. A Routing Underlay for Overlay Networks. In *Proc. ACM SIGCOMM*, pages 11–18, Karlsruhe, Germany, 2003.
- [7] S. Paknikar, M. Kankanhalli, K. R. Ramakrishnan, S. H. Srinivasan, and L. H. Ngoh. A Caching and Streaming Framework for Multimedia. In *Proc. of ACM Multimedia'00*, pages 13–20, CA, October 2000.
- [8] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. of SPIE's Conf. on Multimedia Computing and Networking (MMCN'02)*, San Jose, CA, January 2002.
- [9] S. Sheu, K. A. Hua, and W. Tavanapong. Chaining: A Generalized Batching Technique for Video-On-Demand. In *Proc. of the Int'l Conf. On Multimedia Computing and System*, pages 110–117, Ottawa, Ontario, Canada, June 1997.
- [10] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. In *Proc. ACM SIGCOMM*, pages 149–160, San Diego, CA, 2001.
- [11] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.