Caching Collaboration and Cache Allocation in Peer-to-Peer Video Systems

Ying Cai *Zhan Chen* *Wallapak Tavanapong*

Department of Computer Science
Iowa State University
Ames, Iowa 50011
U.S.A.
Email: {yingcai, zchen, tavanapo}@cs.iastate.edu

May 22, 2006

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: (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 and what existing data should be expunged to make necessary space. We address these problems with two new ideas: *Cell* caching collaboration and *Controlled Inverse Proportional* (CIP) cache allocation. The Cell concept allows cost-effective caching collaboration in a fully distributed environment and can dramatically reduce video lookup cost. On the other hand, CIP cache allocation challenges the conventional caching wisdom by caching unpopular videos in higher priority. Our approach allows the system to retain many copies of popular videos to avoid creating hot spots and at the same time, prevent unpopular videos from being quickly evicted from the system. We have implemented a Gnutella-like simulation network and use it as a testbed to evaluate the proposed technique. Our extensive study shows convincingly the performance advantage of the new scheme.

KEYWORDS: peer-to-peer video services, file lookup, caching collaboration, cache allocation.

1 Introduction

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 [13, 20, 15]. 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) [37, 8, 7, 34, 11, 41], 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 during 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 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 for most users. Most likely, a user wants to use the bandwidth for his/her own interest rather than serving other peers. As reported in [35], 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. Thus, the problem of *partial caching* arises: given a limited buffer, what should a peer cache after downloading a video? If every peer caches or deletes video data in the same sequence, e.g., always from the beginning or the tail of a video, the system may end up with a large amount of redundant video data but only a few critical pieces. In addition to partial caching, another problem is *cache replacement*. Assume a peer downloads a video and its cache is full. Should this video be cached? If yes, how much caching space should be allocated for this video? If the peer currently caches several videos, either in whole or partial, which one should be replaced? Partial caching also makes *video lookup* a challenge: how can a requesting peer find enough video pieces in order to assemble a complete video? Such video lookup could be very expensive. As the peers caching video data may scatter around the whole network, a large scope of the system may have to be searched in order to find enough pieces to assemble a complete video.

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 an unstructured P2P video system without any global or regional super nodes and present a fully distributed video management technique [5]. Our technique consists of two key elements: *Cell* caching collaboration and *Controlled Inverse Proportional* (CIP) cache allocation. A cell is defined to be a set of hosts which together can supply a complete video. Each cell is created dynamically and managed individually. The advantages of our cell 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. While the concept of cell allows cost-effective caching collaboration, our CIP approach addresses the issues of cache allocation. This scheme challenges the conventional caching wisdom by caching *less popular* files in *higher priority*. Since the popular files are requested by many more peers, many more copies of the popular files are retained in the system, effectively preventing hot spots. At the same time, a small number of copies of the less popular files can be retained in the system. We propose two variants of CIP, a heuristic approach called *CIP-H* and an analysis-based approach named *CIP-α*. By tuning the parameter α, CIP-α can achieve the behavior similar to that of *Uniform* allocation or *Proportional* allocation [10, 23].

It is worth mentioning that the proposed technique, although intended for video management, can be used for distributing regular files such as software packages. Thus, we will use the terms video and file interchangeably. The remainder of this paper is organized as follows. We discuss more related

work in Section 2. Two proposed schemes, Cell caching collaboration and CIP cache allocation, are presented in Section 3 and 4, respectively. We discuss our performance study in Section 5 and give our concluding remarks in Section 6.

2 Related Work

Although significant research work has been done on efficient file lookup and storage in decentralized P2P systems [32, 30, 33, 12, 22, 23, 39, 6, 38], most of them assume that each file is an *atomic* unit, i.e., a peer either caches a file in its whole or not at all. Applying these schemes for video storage will leave only a few powerful peers as video suppliers, creating the same server bottleneck problem as in central server architecture. One exception is *Cooperative File System* (CFS) [12], which stores files by blocks and spreads blocks evenly over the available peers. However, to look for a file with n blocks, CFS needs to launch n independent searches, one for each block. To reduce the search cost, CFS adopts Freenet's caching scheme, which replicates a file along its retrieval path. Such proactive file replication may not be feasible for handling large files such as videos. Since video objects are usually very large in size, proactive replication will generate a tremendous amount of network traffic and cause significant delay in serving requesting peers. Streaming a video through its query path is also problematic because peers logically close to each other may be actually far away in their physical connections.

Existing P2P cache allocation techniques give popular files higher priority to be cached [12, 29, 4, 21, 3]. For instance, the work in [10, 23] discusses two extreme cache allocation strategies: *Uniform* and *Proportional* allocation for unstructured P2P systems. Uniform allocation keeps about the same number of copies for all the files. Proportional allocation maintains the number of copies per file proportional to the popularity of the file. For Uniform allocation, files are given equal chance to stay in the system, but with the expense of severe load imbalance since peers that cache very popular files have to handle a much higher number of requests. On the contrary, Proportional allocation achieves optimal load balancing, but unpopular files may have little chance to survive in the system. This problem becomes more serious for video files because as long as one segment of a video disappears from the system, the video becomes incomplete and useless even though many other segments of the file are still available in the system.

Video caching has been studied intensively in past years [14, 31, 36, 18, 45, 28, 43, 24, 26, 1, 27]. However, their application scenarios are fundamentally different. These techniques allow proxy servers, which are usually reliable and equipped with a large amount of disk space, to cache popular video data and deliver them directly to local clients. They are used for various purposes, including reducing client service latency, smoothing video playback, and reducing server and network communication workload. In these techniques, when a client cannot find a video or some part of it in local proxies, the client can always request from some central server. In a P2P video system, however, there is no such central video server, nor does any peer guarantee the availability of any data it caches. Therefore, a main objective of our technique is to balance caching redundancy to maintain the existence of each video. Caching collaboration among a set of proxies was studied in [26, 1, 27]. These techniques use a central server to coordinate a small set of proxies in their caching operations. Applying such centralized coordination in a P2P video system is not feasible because it contains a large number of peers and these peers can be dynamically on and off.

Existing research in P2P video services mainly addresses the following two *video streaming* problems: (1) how to build a video distribution tree and avoid creating network bottleneck [37, 8, 7, 34, 11, 41], and (2) how to aggregate the communication bandwidth of low-end hosts to stream a video at its regular playback rate [42, 44, 19]. In contrast, our focus in this paper is on *video management*, another essential component of P2P video systems. Since our cell can be configured to support video streaming, we believe our technique complements the existing work in building a highly scalable P2P video system.

3 Caching Collaboration

Assume a host downloads a video and the disk space it contributes can cache only part of the video. One question is, which portion of the video should the host cache? 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 data is being retained in the system. Another question is, given the fact that a video may be cached partially by many hosts, how can a host find the pieces that can complement each other to make a complete video? Such video lookup could be very expensive, since the search scope is dictated by the video requester's distance to

the host that holds the last missing piece of the video. In this section, we address these two problems with a novel concept called *Cell* caching. In our discussion, we assume each video is partitioned into segments of equal size and hosts cache videos in segments. A host is called a *caching host* if it caches any segment of a video, and we say a set of video segments is complete if they can make up a complete video.

3.1 Cell Overview

Our main idea is to group the caching hosts into *cells* based on the segments they cache. Each cell is a set 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 may be a cell by its own. Each cell is associated with a binary relation, called *CacheTable*, which tracks the cell members and the corresponding 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.

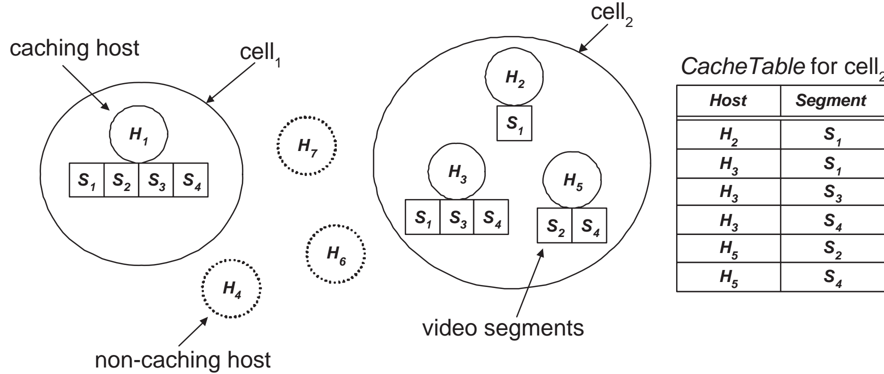

Figure 1: Cell Example

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 data 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 distance from the requesting client to its *nearest* caching host, instead of the *farthest* one that caches the last missing segment. Second, because searching for a video is now simplified as searching for any segment of the video, the operation of searching a video v can be implemented using any advanced P2P video search algorithms. When a caching host receives a search query, it can immediately return its cell information,

from which the query sender can find a complete set of data segments.

In addition to reducing video lookup cost, cell organization makes it possible to coordinate data caching at the cell level to balance segment redundancy. In Figure 1, when host H_6 downloads data from $cell_2$ and is willing to cache one segment, it can cache either S_2 or S_3. Likewise, deleting data segments can also be coordinated at 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. Since a cell usually is very small in size (i.e., the number of its members), caching coordination within a cell does not incur much computation and communication overhead. Yet, such coordination balances the cache redundancy in a cell and can maximally protect the integrity of the cell in terms of supplying a complete set of video segments.

3.2 Cell Operations

3.2.1 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:

$\underline{Cache(n, c)}$

1. Retrieve $c.CacheTable$ of the cell that serves the host;
2. Check $c.CacheTable$ and for each segment, calculate its *redundancy*, i.e., the number of hosts in the cell that cache this segment;
3. Cache the n segments with the least redundancy values;

4. For each newly cached segment, say s, add tuple (h, s) to $c.CacheTable$.

5. Update each cell member with the new $c.CacheTable$.

3.2.2 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 providing 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 $Table_o$;

2. Create a new empty cache table and name it $Table_n$;

3. Set $RemoveMore$ to be $true$;

4. Repeat the following steps until $RemoveMore$ becomes $false$:

   - Find out a host from $Table_o$ that satisfies the following two conditions:
     - All segments cached by this host are also cached by other hosts in $Table_o$;
     - Among all hosts in $Table_o$, this host has the largest number of segments that are not in $Table_n$;
   - If such a host is found, move all its tuples from $Table_o$ to $Table_n$;
   - Otherwise set $RemoveMore$ to be $false$;

5. If $Table_n$ contains a complete set of data segments, then split the cell as follows:
   - Discard $c.CacheTable$;
   - For each host listed in $Table_o$, replace its $CacheTable$ with $Table_o$;
   - For each host listed in $Table_n$, replace its $CacheTable$ with $Table_n$;
6. Otherwise, discard $Table_o$ and $Table_n$.

3.2.3 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:

$\underline{Delete(n, c)}$

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

After a member deletes some segments, it needs to check if the cell is broken. If yes, the member launch a $merge()$ operation, which will be explained in the next subsection.

3.2.4 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.

$\underline{Merge(bc, cList)}$

1. For each caching host h in the broken cell bc, perform the following steps:
   (a) Call $Select(h, cList)$ to find a cell, say c, from $cList$ to accommodate host h;
   (b) 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 $c.CacheTable$;
   (c) Call $Split(c)$;
   (d) If any new cell is created, append it to $cList$.

$\underline{Select(h, cList)}$

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 $cList$;

3.3 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 matrixes. 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 [40, 17], which works as follows. Before a host enters a critical section, it needs to inform 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 and cause the cell 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 a very small amount of 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. That is, our algorithm simply ensures that each new cell can supply a complete set of video segments. Such split is suitable when users do not require to play a video while downloading it. It is possible, however, to configure our cell to support play-while-downloading when each client is assumed to have sufficient downloading bandwidth. Today, many P2P users connect to the Internet through some form of broadband residential connections, such as cable modem and ADSL, which are usually asymmetric with uploading bandwidth significantly less than downloading bandwidth. For such network access, the

transmission rate of a file delivery is limited by the file sender's connection to its ISP, instead of by the Internet backbone [2, 16]. In this case, we can support play-while-downloading by choosing 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 [25]. 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.

4 Cache Allocation

Cell addresses the problem that given a video and a fixed amount of disk space, what part of the video should be cached. It does not answer the problems of cache replacement, i.e., whether or not a newly downloaded video should be cached, if yes, how much cache space should be allocated and to make the space available, which data should be expunged. In this section, we consider these problems.

Traditional algorithms such as least recently used (LRU) and least frequently used (LFU) work well in the environment where there is a central server. By retaining popular videos in cache, these schemes minimize the cost of serving the requests for popular videos. Unpopular videos may be quickly deleted, but this is not a major problem, considering that clients can always request a video directly from the central server. However, in a decentralized P2P system, there is no such central server and an unpopular video may become extinct, since a host seeding the first copy of a video may delete it soon or quit the system.

In [10], Cohen et al. investigate two types of cache allocations for distributed P2P systems: *Uniform* and *Proportional*. Uniform allocation tries to maintain an equal number of copies for all videos in the system regardless of their popularity. This approach guarantees that all videos have equal chance of being cached, regardless of their popularity, but suffers the problem of load balancing. As popular videos are more demanded, their hosts will be overloaded. Proportional allocation, on the other hand, tries to balance server workload by caching videos based on their popularity. The effect of using this approach is, a host tends to always retain a newly downloaded video and deletes the oldest video. As popular videos have the better chance to occupy the cache space, unpopular videos may become extinct

quickly. This problem is worse in some P2P systems such as Freenet [9], where a video is replicated along its query path from its provider to its requester.

The two approaches represent two extremes of cache allocations: Uniform emphasizes equal availability but ignores load balancing, while Proportional emphasizes load balancing but may cause unpopular videos extinct. In this section, we consider how to provide good load balancing while ensuring the existence of all videos and propose two novel techniques called *Controlled Inverse Proportional* (CIP) cache allocation, *CIP-H* and *CIP-α*. Both schemes cache unpopular videos in a higher priority, but CIP-H is a heuristic approach while CIP-α allows one to tune an α parameter to adjust the priority.

4.1 Solution I: CIP-H

This scheme works as follows. Assume a host has a cache space for N segments and there are n videos in its cache. When it downloads a new video, say L segments, the host tries to cache $Min(L, N/(n+1))$ segments of a newly retrieved video. In other words, CIP-H tries to allocate cache space equally among all cached videos. If the host does not have enough empty space for the new segments, it deletes some existing segments. With Cell technique, we can first delete the segments that are most redundant in their cells (these segments may belong to different cells if they are from different videos). If all redundant segments are deleted, we choose to break a cell and this will reduce the number of copies of the corresponding video at system wide. To determine which cell to break, the host compares the popularity of the new video with the popularity of all cached videos and marks a video if its popularity is higher than that of the new video. From all marked videos, the host chooses the one that is least popular to replace. If no video is marked, the newly downloaded video is not cached. This heuristic approach avoids deleting those videos that are less popular than a new video. Meanwhile, it is able to retain those highly popular videos.

4.2 Solution II: CIP-α

Similar to CIP-H, CIP-α is designed to provide a balance between equal availability and load balancing, but such balance can be fine-tuned with an α parameter. We introduce a new concept called *Caching Probability* which is defined to be the probability that this video will be cached after it is retrieved.

Formally, given a video i, its caching probability is defined as

$$P_i = (\frac{p_{min}}{p_i})^{\alpha}, \tag{1}$$

where p_{min} is the popularity of the least popular video, p_i is the popularity of video i, and α is an adjustable parameter. By adjusting the value of α, CIP-α can have various caching effect, ranging from using Uniform and Proportional. Specifically, when $\alpha = 0$, the value of P_i remains 1 for all videos. In this setting, a newly downloaded video is always cached. Since the number of requests for a video in the whole network is proportional to its popularity, the number of copies of a video is proportional to its popularity. As a result, CIP-α achieves the same effect as the Proportional allocation. When $\alpha = 1$, $P_i = \frac{p_{min}}{p_i}$. In this setting, the caching probability of a video is inversely proportional to its popularity. Thus, the number of its copies will be the same as using the Uniform allocation. Since we can vary α value in between 0 and 1, we can achieve other caching effects that can not achieved by using Uniform and Proportional allocation. In practice, we can adjust α value based on the network conditions, such as the number of hosts, the disk space they contribute, the videos they access, and so on, to achieve a desirable tradeoff between equal availability and load balancing.

5 Performance Study

To evaluate the performance of the proposed techniques, we design two performance metrics: *accessibility* and *availability*. The accessibility of a file is defined to be the minimum number of hops that a query message needs to be delivered in order to find a complete file. This parameter measures the cost of locating a complete set of file segments – a higher accessibility means a smaller search scope. In an unstructured P2P system, a one-hop difference in accessibility can result in significant different search cost, especially when flooding techniques like breadth-first search are used. While the accessibility reflects the cost of file lookup, the availability measures the effectiveness of a caching solution: the availability of a file is defined to be the number of distinct copies of the file in the whole system.

5.1 Simulation Setup

In unstructured P2P systems, a peer either caches a file in its whole or not at all. For performance comparison with the cell technique, we design a naive scheme that can cache part of a file as a baseline approach. It is obvious that if all hosts cache from a certain part of files (e.g., starting from the beginning of a file), the cached data segments can be seriously unbalanced. To avoid this problem, a host can randomly choose some segments to cache after downloading a file. We will call this approach *Random*. To evaluate our CIP schemes, we implement two prevailing cache allocation solutions: *Uniform* and *Proportional*. In Uniform, a host decides whether to cache a newly retrieved file based on its popularity. The probability that the file will be cached is inverse proportional to its popularity. In this way although unpopular files are requested less, they have a higher chance to be cached once requested, and vice versa for popular files. Thus all files are allocated equal cache space in the whole system. In Proportional a host always caches a newly retrieved file. The total cache space in the system allocated to a file is proportional to its popularity. In both schemes first-in-first-out (FIFO) cache replacement algorithm is used to recycle the cache space of a host.

We have implemented a Gnutella-like P2P network simulator as our testbed. The network in our simulation consists of 5,000 hosts. The average distance between two hosts in the network is about 7 hops. There are 100 files being shared in the system. The file popularity, which also determines the number of requests generated for each file, follows Zipf-like distribution with the skew set to 1.0. The file with a higher ID value has a lower popularity. For simplicity, we assume each file has the same storage size and partition each file into 10 segments. The host caching capacity also follows Zipf-like distribution with the skew set to 1.0. The largest cache space of a host is 50 segments and the smallest cache space is 1 segment. In the beginning of our simulation, each file is assigned with one seeding host. Then hosts begin to request files and cache the retrieved files. This process continues until the storage capacity in the network has been sufficiently utilized (95% of the total caching space among all hosts is occupied) and finally we collect the performance data under various techniques.

5.2 Simulation Results

Our performance evaluation consists of three parts. In the first part, we compare the accessibility in Cell and Random using Uniform and Proportional respectively. This is to see how the cell technique affects the file accessibility with existing cache allocation schemes. In the second part, we compare the availability in Random using Uniform and Proportional to our proposed CIP schemes. This is to show how the proposed CIP allocations affect file availability under the naive caching scheme. In the last part, we combine Cell and CIP schemes and evaluate their performance in terms of both accessibility and availability. In the following discussion, we use *caching scheme/allocation scheme*, such as Cell/Uniform, to represent the combination of a caching technique and a cache allocation technique.

5.2.1 Part I: Effect of Cell on File Accessibility

In this study, we compare the file accessibility in Cell and Random, using both Uniform and Proportional allocations. Figure 2 shows that the results of Cell and Random share some similarity: with Uniform, the accessibility of different files remains quite constant; with Proportional the accessibility of a file drops quickly when its popularity decreases. As showed in the figure, files with IDs from 97 to 99 have become unaccessible because they have disappeared from the system. This study shows that in all cases the file accessibility in Cell is much better than that in Random. In other words, the file lookup cost in Cell is significantly reduced. This result is not a surprise because Cell allows one to locate a complete set of file segments by finding any one of these segments. Thus, the search scope for a file is determined by the distance between the requesting peer to the nearest peer that caches any segment of the file. In contrast, Random requires one to search the segments one by one individually. As a result, the search scope of a file is determined by the furthest peer that caches the last missing segment. From the results we can see although Uniform benefits unpopular files in search cost, the accessibility of unpopular files in Cell/Proportional is still no worse than that in Random/Uniform.

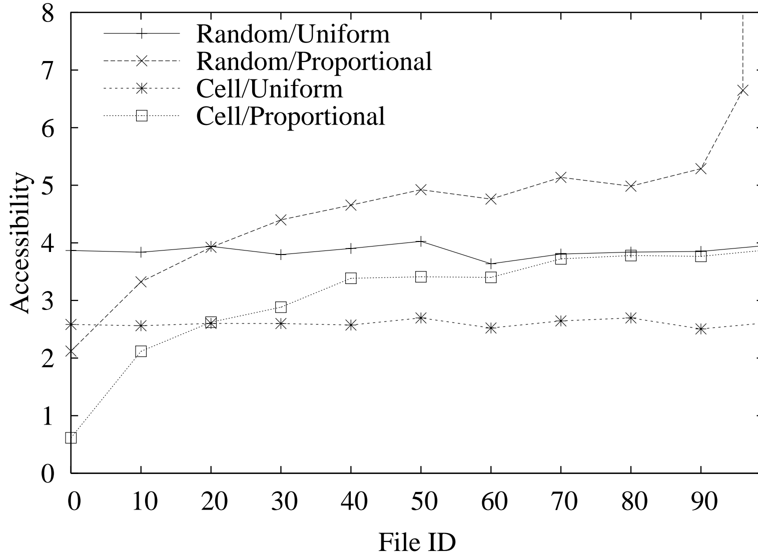

Figure 2: Effect of Cell on Accessibility

5.2.2 Part II: Effect of CIP on File Availability

In this simulation, we study how the proposed CIP allocation schemes affect the availability of files compared to Random with Uniform and Proportional. In CIP-α scheme, the value of α is set to 0.3, 0.5 and 0.7 respectively. Figure 3 shows that for a few most popular files, the availability in CIP-α is a little lower than that in Proportional but much higher than that in Uniform. For remaining files the availability in CIP is quite close to that in Uniform and higher than that in Proportional. It is worth mentioning that in Proportional, 3 least popular files become extinct. In contrast, all CIP schemes maintain at least 10 copies of each file in the system. This demonstrates that CIP schemes are very reliable in retaining unpopular files. While CIP-H and CIP-α schemes have similar performance, it can be seen that CIP-α provides more fine tuned control over the file availability. Increasing α value brings CIP-α closer to Uniform and results in more copies for unpopular files. On the other hand, decreasing α value makes CIP-α behave more similarly to Proportional, i.e., the availability of popular files becomes higher.

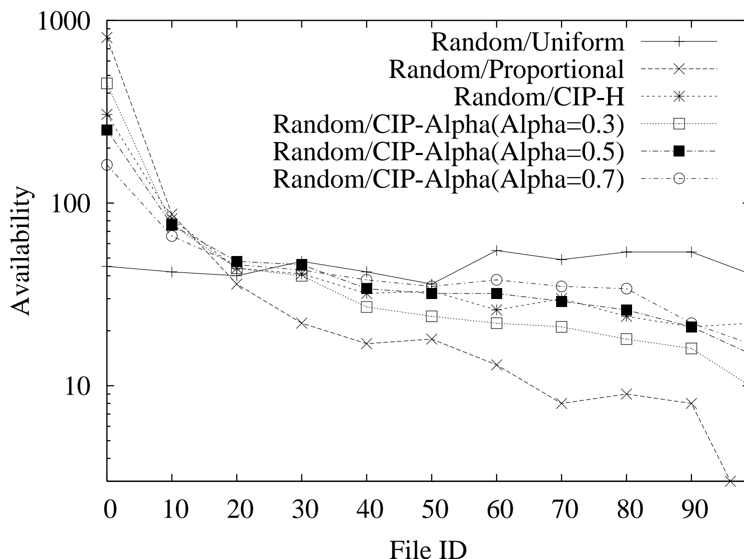

Figure 3: Effect of CIP on Availability

5.2.3 Part III: Effect of Cell/CIP on File Accessibility and Availability

Figure 4 shows the performance of Cell with CIP schemes. With different α values, the accessibility curves in Cell/CIP-α have some minor differences. A smaller α value brings higher accessibility for popular files while a larger α value results in a higher accessibility for unpopular files. We note that under all scenarios, all the accessibility of various files under Cell/CIP schemes is always higher than that of Random/Uniform and Random/Proportional. In Figure 5, the curves of Cell/CIP schemes fall in between the curves of Random/Uniform and Random/Proportional. CIP schemes allow the cell technique to provide tradeoff between achieving better load balance and providing higher availability of unpopular files. In CIP-$alpha$ schemes, the curves with higher α values are more similar to that of Proportional while the curves with lower α values are closer to that of Uniform. This simulation again demonstrates that the combination of Cell and CIP can provide highly flexible control over the availability of files with heterogeneous popularity.

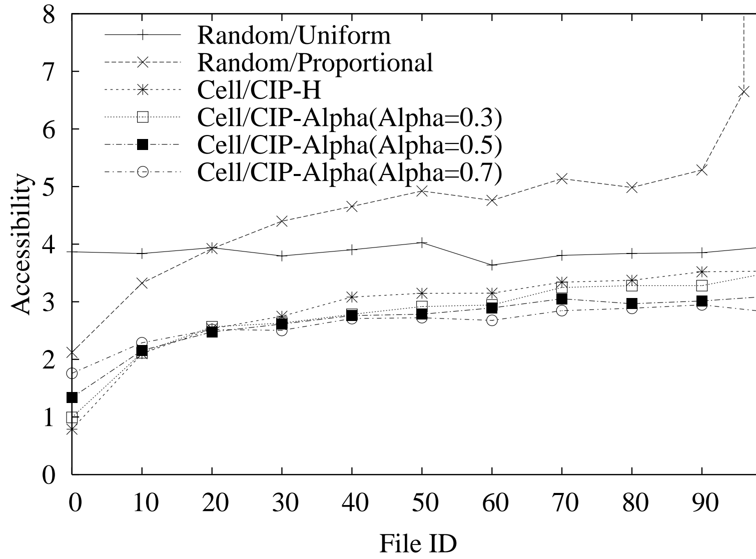

Figure 4: Effect of Cell/CIP on Accessibility

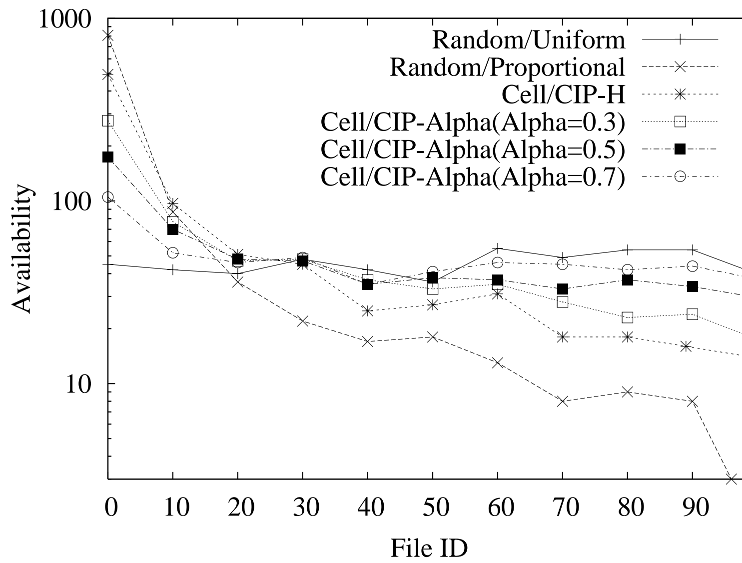

Figure 5: Effect of Cell/CIP on Availability

6 Concluding Remarks

For efficient video management in P2P systems, we have presented two novel ideas, *Cell* caching collaboration and *Controlled Inverse Proportional* (CIP) cache allocation. The Cell technique dynamically groups the participating hosts into cells. Since each cell typically consists of only a small number of hosts, cost-effective caching coordination can be achieved within each cell. This technique also brings a video closer to its requester. Specifically, looking for a complete video is now transformed into looking for a peer that caches any part of the video. Thus, it can dramatically reduce the cost of video lookup when applied in decentralized and unstructured P2P systems. The CIP solution addresses the challenges of cache allocation in P2P systems. Unlike traditional approaches, it gives higher caching priority to less popular files. While this effectively prevents less popular files from becoming extinct, it is able to maintain good load balancing: because the popular files are requested more frequently, they can still have more replications than the less popular ones. Our simulation results confirm that the proposed techniques can significantly improve the overall system performance in terms of increasing both video accessibility and availability.

References

[1] S. Acharya and B. C. Smith. MiddleMan: A Video Caching Proxy Server. In *Proc. ACM/IEEE NOSSDAV*, Chapel Hill, NC, June 2000.

[2] M. Adler, R. Kumar, K. Ross, D. Rubenstein, D. Turner, and D. Yao. Optimal Peer Selection for P2P Downloading and Streaming. In *Proc. of INFOCOM'05*, 2005.

[3] G. Aggarwal, R. Motwani, and A. Zhu. The load rebalancing problem. In *Proc. of ACM SPAA'03*, 2003.

[4] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *Proc. of IPTPS'03*, February 2003.

[5] Y. Cai, Z. Chen, and W. Tavanapong. Video Management in Peer-to-Peer Systems. In *Proc. of The Fifth IEEE International Conference on Peer-to-Peer Computing*, Gemany, September 2005.

[6] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P Systems Scalable. In *Proc. ACM SIGCOMM*, pages 407–418, Karlsruhe, Germany, 2003.

[7] Y.-H. Chu, S. G. Rao, S. Seshan, and H. Zhang. Enabling Conferencing Applications on the Internet Using an Overlay Multicast Architecture. In *Proc. ACM SIGCOMM*, pages 55–67, San Diego, CA, 2001.

[8] 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.

[9] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, HSan Diego, CA, USA, July 2000.

[10] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proc. of ACM SIGCOMM'02*, Pittsburgh, PA, USA, August 2002.

[11] Y. Cui, B. Li, and K. Nahrstedt. oStream: Asynchronous Streaming Multicast in Application-Layer Overlay Networks. *IEEE Journal on Selected Areas in Communications, Special Issue on Recent Advances in Overlay Networks*, 22(1):91–106, January 2004.

[12] 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.

[13] 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.

[14] D. L. Eager, M. C. Ferris, and M. K. Vernon. Optimized Regional Caching for On-Demand Data Delivery. In *Proc. of SPIE's Conf. on Multimedia Computing and Networking (MMCN'99)*, pages 301–316, San Jose, CA, USA, January 1999.

[15] D. L. Eager, M. K. Vernon, and J. Zahorjan. Minimizing Bandwidth Requirements for On-Demand Data Delivery. *IEEE Tras. on Knowledge and Data Engineering*, 13(5):742–757, 2001.

[16] S. Ganguly, A. Saxena, S. Bhatnagar, S. Banerjee, and R. Izmailov. Fast Replication in Content Distribution Overlays. In *Proc. of INFOCOM'05*, 2005.

[17] D. K. Gifford. Weighted Voting for Replicated Data. In *Proc. 7th Symposium on Operating Systems Principles*, pages 150–159, 1979.

[18] S. Gruber, J. Rexford, and A. Basso. Protocol considerations for a prefix-caching proxy for multimedia streams. *Computer Networks*, 33(1-6):657–668, 2000.

[19] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. K. Bhargava. PROMISE: Peer-to-Peer Media Streaming Using CollectCast. In *Proc. ACM Multimedia'03*, pages 45–54, Berkeley, CA, 2003.

[20] 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.

[21] D. Karger and M. Ruhl. New algorithms for load balancing in peer-to-peer systems. In *Tech. Rep. MIT-LCS-TR911, MIT LCS*, July 2003.

[22] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. of ACM ASPLOS*, November 2000.

[23] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proc. of ACM International Conference on Supercomputing*, June 2002.

[24] Z. Miao and A. Ortega. Scalable Proxy Caching of Video under Storage Constraints. *IEEE Journal on Selected Areas in Communications*, 20(7):1315–1327, September 2002.

[25] A. Nakao, L. Peterson, and A. Bavier. A Routing Underlay for Overlay Networks. In *Proc. ACM SIGCOMM*, pages 11–18, Karlsruhe, Germany, 2003.

[26] 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.

[27] Y.-W. Park, K.-H. Baek, and K.-D. Chung. Reducing Network Traffic Using Two-Layered Cache Servers for Continuous Media Data on the Internet. In *Proc. of the IEEE Int'l. Conf. on Computer Software and Applications*, pages 389–374, Taipei, Taiwan, October 2000.

[28] S. Ramesh, I. Rhee, and K. Guo. Multicast with Cache (MCache): An Adaptive Zero-Delay Video-on-Demand Service. In *Proc. of IEEE INFOCOM'01*, pages 22–26, April 2001.

[29] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Proc. of IPTPS'03*, February 2003.

[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM*, pages 161–172, San Diego, CA, 2001.

[31] R. Rejaie, M. Handley, H. Yu, and D. Estrin. Proxy Caching Mechanism for Multimedia Playback Streams in the Internet. In *Proc. of Int'l Web Caching Workshop*, San Jose, CA, March 1999.

[32] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. IFIP/ACM Int. Conf. Distributed Systems Platforms (Middleware)*, pages 329– 350, Heidelberg, Germany, 2001.

[33] A. Rowstron and P. Druschel. Storage Management in Past, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proc. of 18th ACM Symposium on Operating Systems Principles (SOSP01)*, pages 188–201, Alberta, Canada, 2001.

[34] B. B. S. Banerjee and C. Kommareddy. Scalable Application Layer Multicast. In *Proc. ACM SIGCOMM*, pages 205–217, Pittsburgh, PA, 2002.

[35] 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.

[36] S. Sen, J. Rexford, and D. Towsley. Proxy Prefix Caching for Multimedia Streams. In *Proc. of IEEE INFOCOM'99*, pages 1310–1319, March 1999.

[37] 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.

[38] I. Stoica, R. Morris, D. Karger, M. Kaashock, and H. Balakrishman. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. In *Proc. ACM SIGCOMM*, pages 149–160, San Diego, CA, 2001.

[39] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. In *Proc. ACM SIGCOMM*, pages 175–186, Karlsruhe, Germany, 2003.

[40] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

[41] D. A. Tran, K. A. Hua, and T. T. Do. A Peer-to-Peer Architecture for Media Streaming. *IEEE Journal on Selected Areas in Communications, Special Issue on Recent Advances in Overlay Networks*, 22(1):91–106, January 2004.

[42] P. C. V. Padmanabhan, H. Wang and K. Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. In *Proc. ACM/IEEE NOSSDAV*, pages 177–186, Miami, FL, 2002.

[43] K.-L. Wu, P. S. Yu, and J. L. Wolf. Segment-Based Proxy Caching of Multimedia Streams. In *Proc. of World Wide Web*, pages 36–44, 2001.

[44] D. Xu, M. Hefeeda, S. Hambrusch, and B. Bhargava. On Peer-to-Peer Media Streaming. In *Proc. IEEE Conf. on Distributed Computing and Systems (ICDCS)*, pages 363–371, Wien, Austria, 2002.

[45] Z.-L. Zhang, Y. Wang, D. Du, and D. Su. Video Staging: A Proxy-Server-Based Approach to End-to-End Video Delivery Over Wide-Area Networks. *IEEE/ACM Trans. on Networking*, 8(4):429–442, 2000.