2012

Novel Techniques for Large-Scale and Cost-Effective Video Services

Ashwin S. Natarajan
Iowa State University

Novel techniques for large-scale and cost-effective video services

by

Ashwin S Natarajan

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Johnny S Wong, Co-Major Professor

Ying Cai, Co-Major Professor

Leslie Miller

Manimaran Govindarasu

Yong Guan

Iowa State University

Ames, Iowa

2012

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Despite the advance of network technologies in the past decade, providing video services to a large number of users remains a major technical challenge. This is especially true when it comes to serving high-definition videos. This thesis makes two contributions towards providing large-scale and cost-effective video services. 1) We consider the problem of periodic broadcast of popular videos in client/server video systems and present two novel techniques. Our research advances the state of the art with a segmentation rule that can generate a series of broadcast designs, among which we can choose the one that results in the smallest broadcast latency. We show that this rule allows us to design the broadcast technique that is the fastest up to date. 2) We then look at the problem of service scheduling in fully distributed peer-to-peer video systems, where a large number of hosts collaborate for the purpose of video sharing. Our proposed technique allows a client to be served by a server that is beyond its own file lookup scope and can dynamically adjust client and server matches as new video requests arrive in the system. Our performance evaluation shows that these features dramatically improve the system performance to a large extent in terms of reducing service latency under a range of simulation settings.

CHAPTER 1. Introduction

The advance of network technologies has made video services possible over the Internet. Indeed, the last decade has seen a booming of video-hosting websites such as www.youtube.com. According to (4), over 4 billion videos are viewed every day and over 3 billion hours of video are watched every month on Youtube. Traditional television broadcast companies (e.g., Fox (2), ABC (1), and NBC (3)) are now also hosting video clips on the Internet.

However, the quality of videos available on today's Internet is usually low. Unlike regular files, videos are large in size; streaming a video to a remote client takes a substantial amount of server and network bandwidth. A typical video server can sustain only a very limited number of concurrent video streams. This is especially true when it comes to serving high-definition videos. This problem, known as server bottleneck, has been the driving factor that limits the scalability of video services.

Since early 1990, much research has been done towards mitigating the problem of server bottleneck. The proposed techniques can be classified into two categories depending on the targeted system architecture:

- *Client/Server (C/S)*: Videos are stored in one or more dedicated servers, and all client requests are directed to these servers. Representative techniques in this category include *on-demand multicast* (16)(24)(19) and *periodic broadcast* (49)(26)(23). These schemes leverage the facility of IP multicast to improve the scalability of video services by allowing many clients to share a single server stream.

- *Peer-to-Peer (P2P)*: Here a number of hosts collaborate to provide video services. These hosts are peers to each other in the sense that a host can be a client that receives a video and also functions as a server by providing the video to another client. Without relying

on IP multicast, this strategy achieves the effectiveness of IP multicast by allowing a single source stream to be buffered and forwarded to serve many clients. Representative techniques in this category include (42), (13), (40), (15), (47), and (9).

This thesis makes two contributions towards providing large-scale and cost-effective video services:

- We present two novel techniques namely CCA+ and CCB, for efficient periodic broadcast of popular videos under the C/S architecture. Although a number of periodic broadcast techniques have been developed, these schemes are designed in an *ad hoc* manner – it just happens that they work; it is not clear why they are designed in that way. Moreover, most of them are designed with a specific requirement on client bandwidth. If client bandwidth is less than required, they do not work. On the other hand, if client bandwidth is more than required, they cannot take advantage of the extra bandwidth. Our proposed techniques provide an answer to these problems. An important contribution of this work is, we discover a segmentation rule that allows us to find a series of broadcast designs for a particular setting of client bandwidth. More specifically, given a client bandwidth of c times video playback rate, we can have $c! = c \times (c-1) \times \cdots \times 1$ different ways to partition a video. Among these $c!$ segmentation approaches, we can then choose the one that results in the smallest broadcast latency to broadcast a video. We prove the correctness of our techniques and compare their performance with that of several representative approaches.

- We look at the problem of service scheduling in peer-to-peer video systems. Such systems have two features in general: 1) a video is usually available on many participating hosts, and 2) different hosts typically have different sets of videos, though some may partially overlap. From a client's perspective, it can be served by any host having the video it requests. From a server's perspective, it can be used to serve any client requesting the videos it has. Thus, an important question is which servers should be used to serve which clients in the system? We refer to this problem as *service scheduling* and show that different matches between clients and servers can result in significantly different system performance. Finding a right server for each client is challenging not only because a client

can choose only the servers that are within its limited search scope, but also because clients arrive at different times, which are not known a priori. In our work, we address these challenges with a novel technique called *Shaking*. While the proposed technique makes it possible for a client to be served by a server that is beyond the client's own search scope, it is able to dynamically adjust the match between the servers and their pending requests as new requests arrive. Our performance study shows that our new technique can dynamically balance the system workload and significantly improve the overall system performance.

The rest of this thesis is organized as follows. We present new broadcasting techniques in Chapter 2 and our service scheduling approach for P2P video services in Chapter 3. Finally we conclude our thesis in Chapter 4.

CHAPTER 2. Two Novel Techniques for Periodic Video Broadcast

A straightforward way to implement a video-on-demand system is using client/server architecture. Here a central server stores all videos and serves all client requests. The server organizes its resources into a number of *channels*, each capable of sustaining one video stream. Due to the nature of video data, the number of channels available to a video server is usually very limited. If the server simply allocates a channel to serve each client request, the channels will be exhausted quickly, in which case future requests need to wait until some channel becomes available again.

The challenges of addressing the above server bottleneck problem are twofold. First, a channel should be able to serve as many clients as possible. This is to maximize server throughput. Second, a client request should be served as soon as possible. This is to minimize service latency. Towards these two seemly conflicting goals, much work has been done in the past decades to leverage IP multicast so that a single video stream can be shared by multiple clients. The proposed techniques can be classified into two categories:

- *On-demand Multicast* (e.g., (16), (24), (19)): The server maintains a service queue and all client requests are first placed in the queue. When a channel becomes available, the server selects a video and all clients requesting the video are served using one multicast.
- *Periodic Broadcast* (e.g., (49), (26), (23)): Instead of waiting for client requests, the server broadcasts a video repeatedly using one or more channels. All clients requesting for the video tune to the channel(s) to download the video.

In general, on-demand multicast is more suitable for less popular videos. For very popular videos, periodic broadcast is a better choice. To achieve the best performance, the server can employ a hybrid of these techniques. Specifically, it can reserve a fraction of its channels for on-

demand multicast and the rest for periodic broadcast. Since most of the demand (e.g., 80%) is typically for a few (e.g., 10% to 20%) very popular videos (16)(17), the effectiveness of periodic broadcast techniques is very crucial to the overall system performance. In this part of thesis, we focus on periodic broadcast. We will review existing techniques, review their limitations, and then present two new approaches.

2.1 Existing Techniques and Their Limitations

The earliest periodic broadcast technique is *staggered broadcasting* studied in (16)(17). Given a broadcast bandwidth n times of a video's playback rate, this scheme broadcasts the video every $\frac{|v|}{n}$ time units, where $|v|$ is the video length. Since it requires each client to have only one-channel receiving bandwidth and does not need buffer at receiving ends, this scheme is low-cost in terms of client implementation. However, it can reduce the broadcast period only linearly with respect to the increase of broadcast bandwidth.

It was first observed in (49) that more efficient broadcast can be achieved when clients have a higher receiving bandwidth. This observation has inspired a series of advanced techniques, including *Skyscraper Broadcasting* (SB) (26), *Harmonic/Pagoda Broadcasting* (28)(36)(35)(34) (HPB), *Client-Centric Approach* (CCA) (23)(25), *Greedy Disk-Conserving Broadcasting* (GDB) (20), just to name a few. The design of these techniques shares the following characteristics:

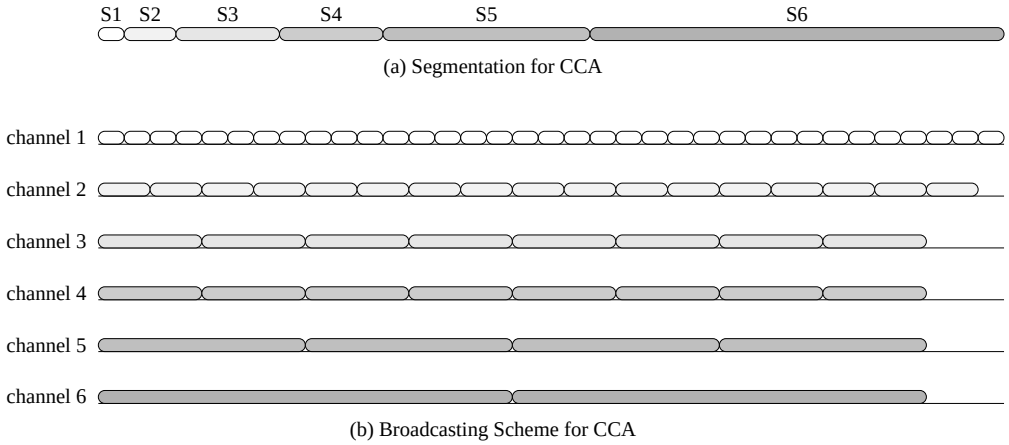
- **Channel design**: The bandwidth of the server is divided into a number of logical channels, each of which can deliver video data at some specific rate.
- **Video segmentation**: The video file is divided into a number of segments.
- **Broadcast schedule**: Each video segment is repeatedly broadcast on a specific channel at some frequency.
- **Playback strategy**: Each client has a number of data loaders, each of which tunes into its pre-assigned channels at different times to download the corresponding video segments. As the segments are downloaded and stored in a disk buffer, they are rendered onto the screen in order for playback.

- **Continuity Principle**: The size of each segment and its broadcast frequency are carefully arranged in such a way that the following *continuity principle* is guaranteed: Once a client starts to download the first segment, it can always have access to the next segment before finishing consuming the current segment.

We explain the above concept using Client-Centric Approach (CCA) (23) as an example. In this scheme, each channel has the same bandwidth, which can sustain one video stream at its regular playback rate. To broadcast a video over k channels, CCA partitions the video into k segments, S_1, S_2, ..., and S_k, each being broadcast repeatedly on one channel. Assuming each client has c data loaders, each of which can download data from one channel at its streaming rate, the size of S_i, denoted as $|S_i|$, is determined using the following formula:

$$|S_i| = \begin{cases} 1, & \text{if } i = 1, \\ 2 \cdot |S_{i-1}| & \text{if } i \ mod \ c \neq 1, \\ |S_{i-1}| & \text{if } i \ mod \ c = 1. \end{cases}$$

As an example, consider $k = 6$ and $c = 3$ (i.e., six channels are used to broadcast and clients can receive data from three channels simultaneously). In this case, the video is partitioned into six segments: S_1, S_2, S_3, S_4, S_5, and S_6. The sizes of them are 1, 2, 4, 4, 8, and 16, respectively. We will refer to $[1, 2, 4, 4, 8, 16]$ as a *broadcast series* corresponding to $k = 6$ and $c = 3$. Figure 2.1 shows a broadcast of this video.



(a) Segmentation for CCA

(b) Broadcasting Scheme for CCA

Figure 2.1 Client-Centric Approach (k=6, c=3)

In CCA, the k segments are organized into $\lceil \frac{k}{c} \rceil$ groups: the first c segments form the first group, the next c segments form the second group, ..., and the last $k - \lfloor \frac{k}{c} \rfloor \cdot c$ segments form the last group. These segments are downloaded group by group. To download segments in one group, a client listens to all corresponding channels (i.e., one data loader for one channel) and downloads a segment as soon as a new broadcast of the segment starts. After the client finishes downloading the last segment in a group, it continues to download the segments in the next group. As soon as the client has access to S_1, it can start to playback the video.

The first property ensures the playback continuity for the segments within one group. It is possible that a client may have to download all segments within one group simultaneously, in which case the client uses all its receiving bandwidth. On the other hand, the second property guarantees that after downloading one group of segments, the client can always shift smoothly to download the first segment in the next group. Since the last segment, say S_i, in one group is made to be the same size as the first segment S_{i+1} in the next group, a new broadcast of S_{i+1} always starts right after the current broadcast of S_i finishes.

The research problem of periodic broadcast design is about video segmentation, i.e., what size should be chosen for each segment. As the size of the first segment determines the broadcast latency, it should be made as small as possible, under the condition that the continuity principle is not violated. To segment a video, two factors needs to be considered: broadcast bandwidth and client bandwidth. In all existing techniques, when more bandwidth is used to broadcast a video, the first segment becomes smaller, because more video segments are created. However, it is not true that they can all make the first segment smaller with a higher client bandwidth. In fact, most of them are designed with some rigid requirements on client bandwidth. For example, the aforementioned SB assumes the receiving bandwidth of each client is two times of video playback rate. At the other extreme, techniques like HPB and their variations (41)(51)(48) would require each client to have a bandwidth that is equal to the server broadcast bandwidth. The former cannot do any better with a higher client bandwidth, whereas the latter simply does not work when the client bandwidth is not equal to the broadcast bandwidth. There are exceptions, though, including CCA and GDB. These schemes are more flexible with client bandwidth in the sense that they provide a specific segmentation approach for a particular client

bandwidth. Given a fixed broadcast bandwidth, their segmentation can make the first segment smaller with an improved client bandwidth. As a result, the more client bandwidth, the more efficient broadcast can be achieved. However, like all others, these schemes are designed in a rather *ad-hoc* manner – it just happens that the playback continuity is guaranteed with the sizes of video segments they choose. Given a video segmentation, one can verify if it ensures the playback continuity, but it is unclear if there is any rule that can be used to guide video segmentation.

2.2 Proposed Techniques

In this section, we present two new techniques, CCA+ and CCB (Client-Centric Broadcast). The first one enhances CCA to further leverage client bandwidth for more efficient video broadcast. The new scheme reduces the broadcast latency up to 50% as compared to CCA. The second technique is a generalized broadcast scheme. At the core of this scheme is a segmentation rule that allows us to design a series of broadcast techniques for a particular setting of client bandwidth. Specifically, given c data loaders at the client side, this approach generates $c!$ different periodic broadcast designs. Among them, we can then choose the one that results in the least broadcast latency. CCB, to our knowledge, is the fastest broadcast technique up to date. We prove the correctness of these two techniques and show its performance advantage as compared with some representative existing techniques.

2.3 Technique 1: CCA+

2.3.1 Motivation Example

Since the first segment determines the broadcast latency, the key to minimize broadcast latency is to make the broadcast series grow as faster as possible, under the condition that the playback continuity is guaranteed. In CCA, the first segment in one group is made the same size as the last segment in the previous group. We will adopt the same strategy to ensure that a client can always continue to download the next group after downloading the current group. Our research focuses on how to grow the sizes of the segments within each group as fast as

possible.

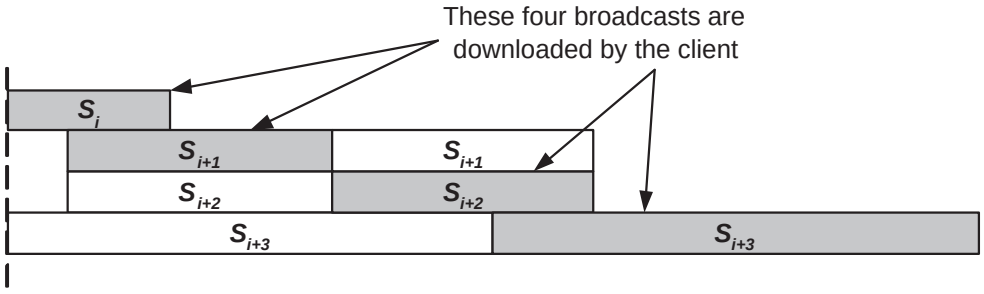
As a motivation example, let's assume client receiving bandwidth is 2 channels. Let S_i, S_{i+1}, S_{i+2}, and S_{i+3} be four consecutive segments, where S_i and S_{i+1} are in one group and S_{i+2} and S_{i+3} in the next group. Suppose the sizes of S_i and S_{i+1} are known and their playback continuity is guaranteed. Since S_{i+2} is the first segment in a group, we make it the same size as S_{i+1}, i.e., $|S_{i+2}| = |S_{i+1}|$. The question now is, what should be the size of S_{i+3}? Our key observation is that, as long as $|S_{i+3}|$ satisfies the following two conditions: 1) it is a multiple of $|S_i|$, and 2) it is no larger than $|S_i| + |S_{i+1}| + |S_{i+2}|$, a client can always play back the four segments continuously. If $|S_{i+3}|$ is a multiple of $|S_i|$, then there are only three possible broadcast alignments of these two segments: 1) the broadcast of S_i and S_{i+3} starts at the same time, 2) a new broadcast of S_{i+3} follows right after a broadcast of S_i finishes, and 3) a broadcast of S_i starts and finishes in between a broadcast of S_{i+3}. The three alignments are illustrated in Figure 2.2. We now show that a client can always start to download S_{i+3} before finishing playing back S_i, S_{i+1}, and S_{i+2}:

- *Alignment 1*: Since S_i and S_{i+3} start at the same time, and $|S_{i+3}|$ is no larger than $|S_i| + |S_{i+1}| + |S_{i+2}|$, a new broadcast of S_{i+3} must have started before a client finishes consuming S_i, S_{i+1}, and S_{i+2}. In this case, after the client finishes downloading the current broadcast of S_{i+1}, it proceeds to download S_{i+2} and S_{i+3} as soon as their broadcast starts.

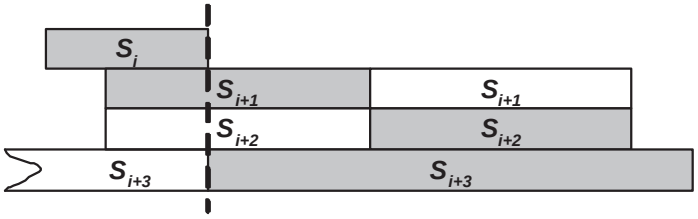
- *Alignment 2*: Since a new broadcast of S_{i+3} starts right after the current broadcast of S_i finishes, a client can use the same data loader to download S_{i+3} after downloading S_i.

- *Alignment 3*: Since the current broadcast of S_{i+3} starts earlier than that of S_i and $|S_{i+3}| \leq |S_i| + |S_{i+1}| + |S_{i+2}|$, a new broadcast of S_{i+3} will start within the next $|S_i| + |S_{i+1}| + |S_{i+2}|$ time units. Thus, after downloading S_i, a client can use the same data loader to download S_{i+3}.

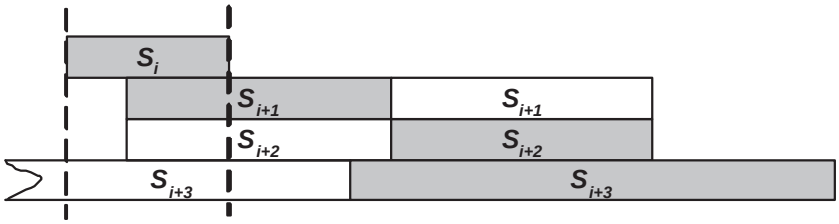
According to the above observation, we can make the size of S_{i+3} equal to the largest number that is a multiple of $|S_i|$ but no larger than $|S_i| + |S_{i+1}| + |S_{i+2}|$. We now consider the broadcast series. To start with, we set the sizes of the first two segments, S_1 and S_2, to be 1 and 2, respectively. Then we have $|S_3| = 2$, since S_3 is the first segment in the next group.



(a) Alignment 1: A broadcast of S_i and S_{i+3} starts at the same time



(b) Alignment 2: A broadcast of S_{i+3} starts right after a broadcast of S_i finishes



(c) Alignment 3: A broadcast of S_i starts and finishes in between a broadcast of S_{i+3}

Figure 2.2 All Possible Broadcast Alignments of S_i and S_{i+3}

As for $|S_4|$, we can make it 5, which is the largest number that is a multiple of $|S_1|$ and no larger than $|S_1| + |S_2| + |S_3|$. We can then have $|S_5| = 5$. Knowing $|S_3|$, $|S_4|$, and $|S_5|$, we can make $|S_6|$ equal to 12, the largest number that is a multiple of $|S_3|$ and no larger than $|S_3| + |S_4| + |S_5|$. Similarly, we can compute the sizes of other segments, $|S_7|$, $|S_8|$, $\cdots$, and derive the following broadcast series:

$[1, 2, 2, 5, 5, 12, 12, 25, 25, 60, 60, 125, 125, 300, 300, \dots]$

The above broadcast series is faster than CCA's broadcast series for a client with 2 channels of receiving bandwidth. The broadcast series of CCA for such a case is given by:

$[1, 2, 2, 4, 4, 8, 8, 16, 16, 32, 32, 64, 64, 128, \dots]$

It is worth mentioning that the new broadcast series grows faster than that from Skyscraper Broadcasting (26), which was designed specifically for 2-channel receiving bandwidth and until now is the fastest one in this setting:

$[1, 2, 2, 5, 5, 12, 12, 25, 25, 52, 52, 105, 105, 212, 212, \dots]$

2.3.2 Design of CCA+

The above motivation example allows us to develop a more efficient broadcast technique. We will call this scheme $CCA+$, alluding to the fact that it is an improvement of the original CCA. Given k channels of broadcasting bandwidth and c channels of receiving bandwidth, CCA+ partitions a video into k segments and organizes them into $\lceil \frac{k}{c} \rceil$ groups. Also similar to CCA, the new scheme sets the sizes for the first c segments to be $1, 2, 2^2, ..., 2^{c-1}$, and makes the first segment in each group the same size as the last segment in the previous group. However, for each of other segments S_i, its size is determined by the sizes of its previous $c+1$ segments, S_{i-c-1}, S_{i-c}, S_{i-c+1}, $\cdots$, and S_{i-1}. Specifically, the size of S_i is set to be the largest multiple of $|S_{i-c-1}|$ that is not larger than $\sum_{j=i-c-1}^{i-1} |S_j|$. More formally, CCA+ uses the following formula to generate the broadcast series:

$$|S_i| = \begin{cases} 2^{i-1}, & \text{if } 1 \le i \le c, \\ |S_{i-1}|, & \text{if } (i) \ mod \ c = 1 \text{ and } i > c, \\ \lfloor \frac{\sum_{j=i-c-1}^{i-1} |S_j|}{|S_{i-c-1}|} \rfloor \cdot |S_{i-c-1}| & \text{otherwise.} \end{cases}$$

At receiving ends, clients also download video segments group by group. However, the data loader used to download the ith segment in one group will be used to download the $(i+1 \ mod \ c)$th segment in the next group. We now prove that a client can always play back the video continuously. Clearly, the playback continuity for the first c segments is guaranteed, and after the client finishes downloading the cth segment, it can always proceed to download the next segment. So what we need to prove is, if a client can playback $c+1$ consecutive segments continuously, then it can always start to download the next segment before it finishes playing back the $c+1$ segments.

Let S_i, S_{i+1}, ..., S_{i+c}, and S_{i+c+1} be any $c+2$ consecutive segments and the playback continuity of the first $c+1$ segments are known to be guaranteed. If $|S_{i+c+1}| = |S_{i+c}|$, then a client can download the two segments sequentially using the same data loader, because a new broadcast of S_{i+c+1} always starts right after a broadcast of S_{i+c} finishes. If $|S_{i+c+1}| \neq |S_{i+c}|$, then according to the above formula, $|S_{i+c+1}|$ must be a multiple of $|S_i|$. As such, there are only three possible broadcast alignments for S_i and S_{i+c+1}, as showed in Figure 2.3. We analyze them as follows:

- Alignment 1: S_i and S_{i+c+1} start at the same time. Because the client can access only c channels simultaneously, it cannot download the current broadcast of S_{i+c+1}. From the time when the client starts to download S_i, it will take $\sum_{j=1}^{i+c} |S_j|$ time units to play back all the first $c+1$ segments. Since $|S_{i+c+1}| \leq \sum_{j=i-c-1}^{i-1} |S_j|$, a new broadcast of S_{i+c+1} will start in the next $\sum_{i=1}^{c+1} |S_i|$ time units. It is also true that, because $|S_i| \leq |S_{i+c+1}|$, the new broadcast of S_{i+c+1} will start after the current broadcast of S_i. As such, the client can download S_i and then S_{i+c+1} using the same data loader.

- Alignment 2: A new broadcast of S_{i+c+1} starts right after a broadcast of S_i finishes. In this case, after the client finishes downloading S_i, it can use the same data loader to download S_{i+c+1}. Since the client has access to S_{i+c+1} before consuming the previous $c+1$ segments, the playback continuity is guaranteed.

- Alignment 3: A broadcast of S_i starts and finishes in between a broadcast of S_{i+c+1}. Since $|S_{i+c+1}|$ is no larger than $\sum_{j=1}^{i+c} |S_j|$ and is a multiple of $|S_i|$, the current broadcast

of $|S_{i+c+1}|$ must finish in the next $\sum_{j=2}^{i+c} |S_j|$ time units. Thus, before the first $c+1$ segments are consumed completely, a new broadcast of S_{i+c+1} must have already started. Again, in this case, the data loader used to download the current broadcast of S_i can be used to download the next broadcast S_{i+c+1}.

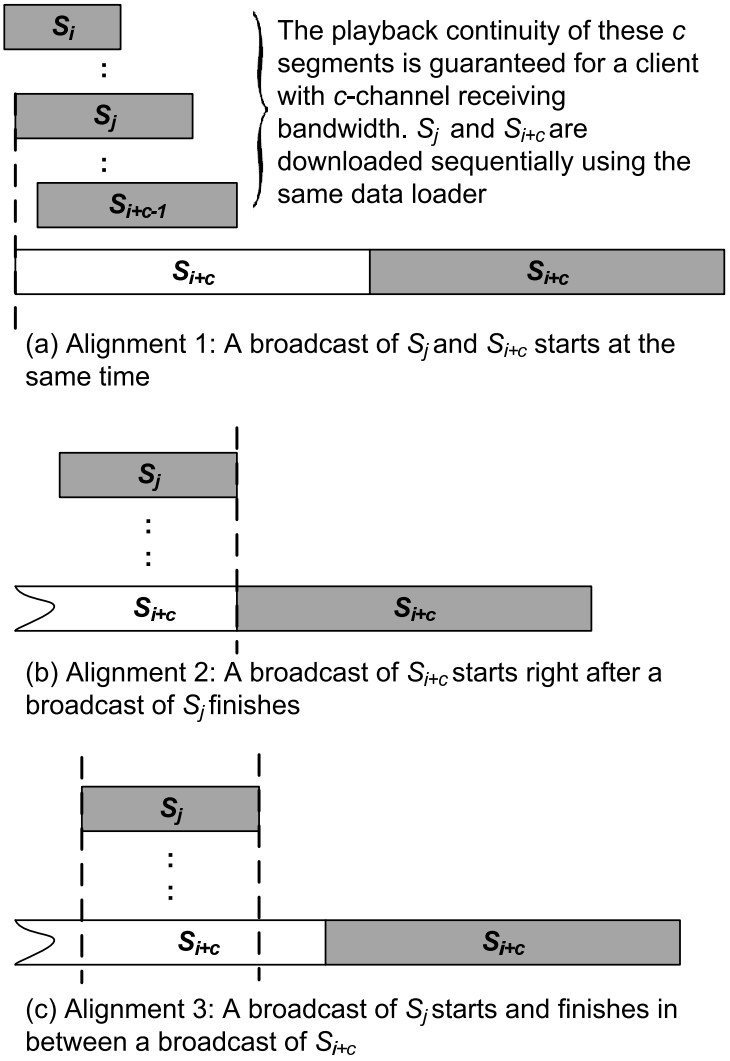


Figure 2.3 Three possible alignments of S_i and S_{i+c+1}

2.4 Technique 2: CCB

2.4.1 Segmentation Rule

As mentioned earlier, the key to minimize broadcast latency is to make the broadcast series grow as fast as possible, but under the condition that the playback continuity is guaranteed. Let S_i, S_{i+1}, ..., and S_{i+c-1} be a group of c consecutive segments and a client with c data loaders can start to play back as soon as it has access to a new broadcast of S_i. The question is, what is the maximum size of the next segment, S_{i+c}? Here we introduce a rule that can help answer this question. Our key observation is, if a client uses the data loader which downloads segment S_j $(i \leq j \leq i+c-1)$ to download S_{i+c}, then the playback continuity of the $c+1$ segments is guaranteed as long as the size of S_{i+c} satisfies the following two conditions: 1) it is a multiple of $|S_j|$, and 2) it is no larger than $\sum_{m=j}^{i+c-1} |S_m|$.

We refer to the above rule as a *Segmentation Rule*. If $|S_{i+c}|$ is a multiple of $|S_j|$, then there are only three possible broadcast alignments for these two segments: 1) the broadcast of S_j and S_{i+c} starts at the same time, 2) a new broadcast of S_{i+c} follows right after a broadcast of S_j finishes, and 3) a broadcast of S_j starts and finishes during a broadcast of S_{i+c}. The three alignments are illustrated in Figure 2.3. We now show that a client can always start to download S_{i+c} before finishing the playback of S_{i+c-1}:

- Alignment 1: S_j and S_{i+c} start at the same time. Because the same data loader is used to download the two segments, a client cannot download the current broadcast of S_{i+c}. From the time when the client starts to download S_j, it will take $\sum_{m=j}^{i+c-1} |S_m|$ time units to finish the playback of segment S_{i+c-1}. Since $|S_{i+c}| \leq \sum_{m=j}^{i+c-1} |S_m|$, a new broadcast of S_{i+c} will start no later than $\sum_{m=j}^{i+c-1} |S_m|$ time units. On the other hand, because $|S_j| \leq |S_{i+c}|$, the new broadcast of S_{i+c} can start only after the current broadcast of S_j. As such, in this case, the client can download S_j and then S_{i+c} sequentially.

- Alignment 2: A new broadcast of S_{i+c} starts right after a broadcast of S_j finishes. In this case, after the client finishes downloading S_j, it can use the same data loader to download S_{i+c}. Since the client has access to S_{i+c} before consuming S_{i+c-1}, the playback

continuity is guaranteed.

- Alignment 3: A broadcast of S_j starts and finishes in between a broadcast of S_{i+c}. Since $|S_{i+c}|$ is no larger than $\sum_{m=j}^{i+c-1} |S_m|$ and is a multiple of $|S_j|$, the current broadcast of $|S_{i+c}|$ must finish in the next $\sum_{m=j+1}^{i+c-1} |S_j|$ time units. Thus, before the $i+c-j$ segments are consumed completely, a new broadcast of S_{i+c} must have already started. Again, in this case, the same data loader used to download the current broadcast of S_j can be used to download the next broadcast of S_{i+c}.

2.4.2 Design of CCB

The segmentation rule allows us to design a series of segmentation approaches, given a particular setting of client bandwidth. As an illustration example, let's assume each client has two data loaders (i.e., $c = 2$). In this case, every two segments form a group. For the two segments in the first group, we can make $|S_1| = 1$ and $|S_2| = 2$. After a data loader finishes downloading one segment in a group, it will proceed to download some segment in the next group. We will use the terms *download schedule* to refer to an assignment of data loaders to download segments in two consecutive groups. When $c = 2$, there are two download schedules, and for each schedule, we can have one particular broadcast series:

- *Schedule 1*: The data loader which downloads the first segment in one group is assigned to download the first segment in the next group, and the data loader which downloads the second segment in one group will download the second segment in the next group. Since S_3 is downloaded by the data loader that downloads S_1, we can make its size equal to 3, the largest number that is a multiple of $|S_1|$ but not larger than $|S_1| + |S_2|$. Since S_4 is downloaded by the data loader that downloads S_2, we can make its size equal to the largest number that is a multiple of $|S_2|$ but not larger than $|S_2| + |S_3|$. As such, we have $|S_4| = 4$. Since the playback continuity of S_3 and S_4 is guaranteed, we can then determine the size of the segments in the next group, S_5 and S_6, and so on so forth. As such, we have the following broadcast series: [1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, ...].

- *Schedule 2*: The data loader which downloads the first segment in one group is assigned to download the second segment in the next group, and the data loader which downloads the second segment in one group will download the first segment in the next group. Since S_3 is downloaded by the data loader that downloads S_2, we can make its size equal to that of S_2. As for S_4, since it is downloaded by the data loader that downloads S_1, we can make its size equal the largest number that is a multiple of $|S_1|$ but not larger than $|S_1|+|S_2|+|S_3|$. So we have $|S_4| = 5$. Likewise, we can determine the size of the segments group by group and have the following broadcast series: [1, 2, 2, 5, 5, 12, 12, 25, 25, 60, 60, 125, 125, 300, 300, ...].

Note that both of the above broadcast series are faster than the one from CCA under the same setting: [1, 2, 2, 4, 4, 8, 8, 16, 16, 32, 32, 64, 64, 128, 128, 256, ...]. The series under schedule 2 is same as the broadcast series of CCA+ under the same setting. This series grows faster than that from Skyscraper Broadcasting, [1, 2, 2, 5, 5, 12, 12, 25, 25, 52, 52, 105, 105, 212, 212, ...], which was designed specifically for 2-channel client bandwidth and until now is the fastest one in this setting. Since both new broadcast series guarantee playback continuity, we can choose the faster one for video segmentation and let clients download segments using the corresponding download schedule.

In general, given a set of c data loaders, there are a total of $c! = c \times (c-1) \times \cdots \times 1$ different download schedules. This is due to the fact that after a data loader downloads one segment in a group, it can be used to download any segment in the next group. For example, when $c = 3$, there are $3 \times 2 \times 1$ download schedules, as illustrated in Figure 2.4. We will use an array of $l[1..c]$ to denote a schedule that indicates which segment in the previous group was downloaded by the data loader used to download the j^{th} segment in the current group. As mentioned earlier, any set of c segments among the k segments of the video, form one group. The next set of c segments form the next group. If the data loader that downloads the ith segment in the current group is the one that downloads the jth segment in the previous group, we have $l[i] = j$. For example, for schedule 1 in Figure 2.4, we have $l[1] = 1$, $l[2] = 2$, and $l[3] = 3$; similarly, for schedule 6, we have $l[1] = 2$, $l[2] = 3$, and $l[3] = 1$. Given a schedule

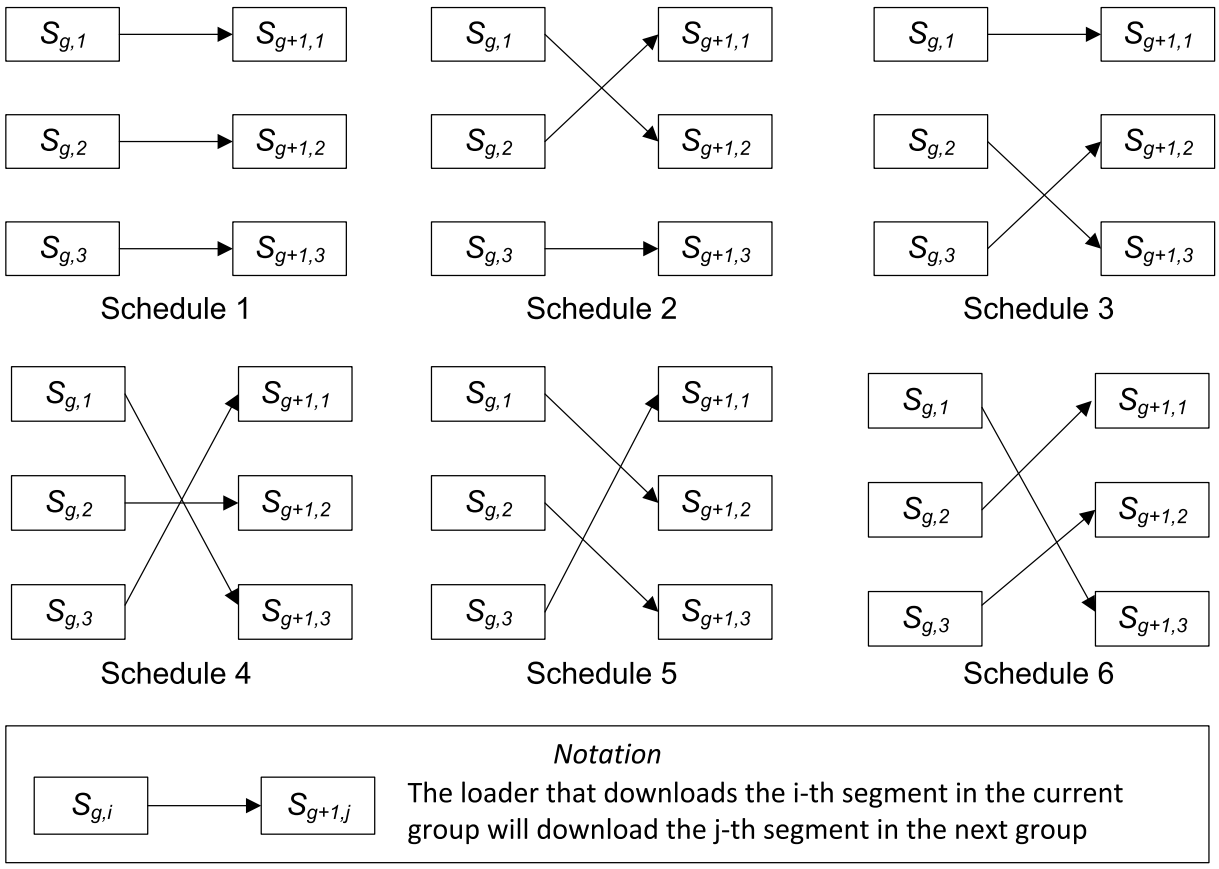


Figure 2.4 When $c = 3$, there are six different download schedules

$l[1..c]$, we can compute the size of each video segment using the following algorithm, where k is the number of broadcast channel:

According to the segmentation rule mentioned earlier, two conditions must be satisfied in order to ensure continuity: 1) $|S_i|$ should be a multiple of $|S_{i-c+l[i]}|$. 2) It should be no larger than the sum of the sizes of the segments from $S_{i-c+l[i]}$ till S_i, which is denoted by the variable *sum* in algorithm 1. The value of $|S_i|$ is chosen such that $|S_i|$ is the largest multiple of $|S_{i-c+l[i]}|$ that is lesser than *sum*. Given a client bandwidth of c channels, we can enumerate all $c!$ schedules. For each of these schedules, we apply the above algorithm to generate the corresponding broadcast series. The fastest one is then chosen for video segmentation. We refer to this technique as *Client-Centric Broadcast* (CCB). To play back the video, a client first retrieves the download schedule from the server, and then applies its data loaders to receive the segments accordingly.

Algorithm 1 *Segmentation*$(k, c, l[1..c])$

Let $S_1, S_2, .., S_k$ be the set of k segments
for all $i = 1$ to $c - 1$ **do**
  $|S_i| = 2^{i-1}$
end for
for all $i = c$ to k **do**
  $sum = 0$
  for all $m = i - c + l[i]$ to $i - 1$ **do**
    $sum = sum + |S_m|$
  end for
  $|S_i| = \left\lfloor \frac{sum}{|S_{i-c+l[i]}|} \right\rfloor * |S_{i-c+l[i]}|$
end for

2.5 Performance Study

For performance evaluation, we compare CCB, CCA, CCA+, and GDB. To our knowledge these are the only ones up to date that can leverage client bandwidth for more efficient broadcast. Unlike others, GDB requires client receiving bandwidth to be at least 3 channels. It uses the following integer-valued partition function $f(n)$ to generate broadcast series, where i denotes the number of video streams that a client can receive simultaneously:

$$f_{GDB(i)}(n) = \begin{cases} 2^{n-1}, & \text{if } n \leq i, \\ \lfloor \frac{(\sum_{j=n-i+1}^{n-1} f_{GDB(i)}(j)}{f_{GDB(i)}(n-i+1)} \rfloor * f_{GDB(i)}(n-i+1), & \text{if } n > i \end{cases}$$

When the client receiving bandwidth is 2 channels, we also compare the performance of CCB, CCA+ and CCA to that of SB. Since we are primarily interested in the relative performance of these techniques, we assume the system has only one video. If the system has n videos, we can divide the server bandwidth into n parts and create n virtual servers, each serving one video. Thus, the results reported in this section are also valid for systems with many videos. We choose *broadcast latency* as our performance metric and will focus on how this is affected by client bandwidth and server bandwidth. Our study uses a 120-minute video and varies the server bandwidth from 2 to 10 channels, which we believe is large enough to show the performance trend.

2.5.0.1 Effect of Server Bandwidth

When the client bandwidth is 2 channels, the performance results of SB, CCA, CCA+ and CCB are plotted in Table 2.1. We do not include GDB in this study since it does not work in this setting of client bandwidth. The figure shows that as the server bandwidth increases, the broadcast latency under all four techniques reduces sharply. In all settings, however, CCB outperforms CCA to a large extent. For example, when the server bandwidth is 5 channels, CCB reduces the broadcast latency to 450 seconds. In contrast, CCA incurs 553 seconds. This study also shows that CCB performs no worse than SB and CCA+ in all settings. It is worth mentioning that up to date, SB and CCA+ are the two fastest broadcasting techniques when client bandwidth is 2 channels.

Server Bandwidth	Broadcast Latency (seconds)			
	SB	CCA	CCA+	**CCB**
2	2400	2400	2400	**2400**
3	1440	1440	1440	**1440**
4	720	800	720	**720**
5	480	553.84	480	**450**
6	266.66	342.85	266.66	**266.66**
7	184.61	248.27	184.61	**184.61**
8	112.5	160	112.5	**112.5**
9	80.89	118.03	80.89	**80.89**
10	51.06	77.41	48.32	**48.32**

Table 2.1 Client bandwidth = 2 channels

Table 2.2 shows the performance results of GDB, CCA, CCA+ and CCB when the client bandwidth is 3 channels. It shows that CCB performs significantly better than CCA. For example, when the server bandwidth is 10 channels, the broadcast latency under CCB and CCA is 17 seconds and 34 seconds, respectively. The performance gap between the two techniques actually increases as the server bandwidth increases. CCA also outperforms CCA+ and GDB in all settings. When the client bandwidth is fixed at 4 channels, the performance results of the four techniques are plotted in Table 2.3. It shows that when the server bandwidth is 10 channels, CCA, GDB, CCA+ and CCB guarantee a broadcast latency of 22, 14.9, 14.4 and 9.6

seconds, respectively. In other words, CCB outperforms CCA by 140% and GDB and CCA+ by about 60% respectively.

Server Bandwidth	Broadcast Latency (seconds)			
	CCA	CCA+	GDB	**CCB**
2	2400	2400	2400	**2400**
3	1029	1029	1029	**1029**
4	654.54	654.54	654.54	**514.28**
5	378.94	327.27	342.85	**276.92**
6	205.71	171.42	232.25	**156.52**
7	141.17	116.12	130.90	**88.88**
8	86.74	63.16	91.13	**51.06**
9	48.97	33.8	55.81	**29.87**
10	34.12	23.07	40.22	**17.30**

Table 2.2 Client bandwidth = 3 channels

Server Bandwidth	Broadcast Latency (seconds)			
	CCA	CCA+	GDB	**CCB**
2	2400	2400	2400	**2400**
3	1029	1029	1029	**1029**
4	480	480	480	**480**
5	313.04	313.04	248.27	**240**
6	184.61	156.52	135.84	**124.13**
7	101.40	80	77.41	**65.45**
8	53.33	41.37	44.17	**34.95**
9	36.18	27.9	25.44	**18.65**
10	22.01	14.45	14.90	**9.67**

Table 2.3 Client bandwidth = 4 channels

2.5.0.2 Effect of Client Bandwidth

This study investigates the effect of client bandwidth on the broadcast latency. Again, the video duration is set to be 120 minutes. We fixed the server bandwidth at 10 channels, and varied the client receiving bandwidth from 2 to 10 channels. The results of the four techniques, CCA, CCA+, CCB, and GDB are plotted in Table 2.4. As mentioned before, GDB does not

work when the client receiving bandwidth is less than 3 channels. It shows that the CCB remains the best performer, until the client receiving bandwidth becomes equal to the server bandwidth. In that case, all these techniques have the same broadcast series $(1, 2, 2^2, \cdots, 2^{k-1})$.

Client Bandwidth	Broadcast Latency (seconds)			
	CCA	CCA+	GDB	**CCB**
2	77.41	48.32	N/A	**48.32**
3	34.12	23.07	47.05	**17.3**
4	22.01	14.45	14.9	**9.60**
5	13.66	10.02	9.31	**8.05**
6	13.25	9.60	7.78	**7.39**
7	12.52	9.44	7.28	**7.15**
8	11.26	9.39	7.10	**7.07**
9	9.38	9.38	7.05	**7.04**
10	7.03	7.03	7.03	**7.03**

Table 2.4 Server bandwidth = 10 channels

CHAPTER 3. Shaking: A Scheduling Technique for P2P Video Services

An alternative way to build a VOD system is to apply the concept of peer-to-peer computing. A host receiving a video stream can forward the stream to serve others. After it finishes playing back, it can also cache the video and supply to future requests. As such, unlike in the client/server architecture, hosts participating in P2P video services are peers to each other in the sense that they can all contribute their resources to the whole community. This feature effectively alleviates the problem of server bottleneck and can provide highly scalable video services at a minimal cost.

The idea of P2P video services was first explored in (42) and (13). Since then, many advanced techniques have been developed towards efficient and robust streaming (.e.g., tree-based (27)(14)(46), mesh-based (18)(21)(7)(50), topology-awareness (30)(9)). In contrast to existing research, we investigate the problem of service scheduling in fully distributed P2P video systems. In the rest of this section, we explain the problem, discuss some closely related work, and then present our solution.

3.1 Problem of Service Scheduling

Consider a decentralized P2P system that consists of a large number of hosts that collaborate for the purpose of video sharing. Without causing ambiguity, we will simply use *client* and *server* to refer to a host requesting a video and a host supplying a video, respectively. A client uses some P2P file lookup technique (e.g., (32)(37)(44)(38)) to search a video. Typically, the video is available on a number of servers. So the question is which server should be used to serve the client? This problem, which we refer to as service scheduling, is complicated because of several factors.

First, from a client's perspective, it should be served by the one with minimal service latency. However, although a video may be available on many hosts in the system, a client looking for the video usually can locate only a few, typically one, of them. The servers found may not be able to provide the fastest service.

Second, even if each client can find all available server candidates, different matches between clients and servers can result in significantly different performance results. As an example, consider two servers, S_1 and S_2. S_1 caches videos v_1 and v_2 while S_2 has videos v_1 and v_3. Given two clients, C_1 and C_2, requesting for v_1 and v_2, respectively, if C_1 is served by S_1, C_2 will have to wait until S_1 finishes serving C_1. However, if C_1 is served by S_2, then C_2 can be served by S_1 immediately. This problem is attributed to this fact: while a client may have a number of hosts as its server candidates, a host can also be a server candidate to more than one client – a host caching a number of files can be a server to any client requesting these files.

Third, clients request files at different times. Thus, the match between clients and servers must be dynamically adjusted as clients arrive. This is particularly challenging in decentralized P2P systems, where each client finds and chooses its server by its own. In the previous example, when C_1 arrives, it can choose either S_1 or S_2. It is the next request that determines which server should be used to serve C_1.

3.2 Related Work

There has been a rich literature on service scheduling research, starting from CPU scheduling in operating systems (e.g., (31), (43), (12), (10)). Here we discuss the work related to video services. The early work focuses on how to choose a video to serve. Here a central server is used to serve all clients. The server maintains a queue and all service requests are first placed in the queue. When the server has a free channel, it selects one of the requested videos in the queue and sends the video to all its requestors using multicast. To decide which video to select, Dan et al proposed *First Come First Served* (FCFS) and *Maximum Queue Length First* (MQL) (16). The former selects the video with the oldest request to serve next. It treats each video equally regardless of its popularity, but will result in lower system throughput. MQL maximizes the system throughput by selecting the batch with the largest number of pending

requests to serve first. It, however, is unfair since it favors more popular videos. Aggarwal et al addressed this problem with a technique called *Maximum Factored Queue length first* (MFQ) (6). This scheme improves on the FCFS and MQL policies by taking into account both the waiting times of the requests and the popularity of the videos. When sufficient bandwidth becomes available, MFQ selects the pending batch with the largest size weighed by the *best* factor, (the associated access frequency)$^{-\frac{1}{2}}$, to serve next. This scheme can achieve throughput close to that of MQL with little compromise of its fairness. All these techniques assume all clients are served by a single server and cannot be applied for our problem, which is how to match clients and servers in order to minimize service latency.

In the context of P2P live streaming systems, there is research on how to build a transmission schedule for a client. It is assumed that a video may be segmented and stored on multiple servers. To download a video, a client may need to download its segments from multiple sources. When a client joins a live streaming session and downloads different segments from different servers, it is possible that some of the segments may miss their deadlines because of the heterogeneous upload bandwidth of servers and their workload. So the problem is, given a set of servers $S_1, S_2, ..., S_n$ with uploading bandwidths $b_1, b_2, ..., b_n$, how to build a schedule for a client to receive segments $Seg_1, Seg_2, ..., Seg_m$ within their deadlines such that the total number of segments missing their deadlines is minimized? This problem has been proved to be NP-complete (22). There are a number of heuristic solutions (e.g., (33), (5), (53), (22), (52), (11)), but they are not applicable to our problem. They cannot help a client to find a server that is outside of its search scope. Moreover, these techniques consider the download schedules of video segments that belong to a same video. In contrast, our research aims at finding a good match between clients and servers, where clients may be requesting different videos.

3.3 Proposed: Shaking

We assume a client C requesting a video V can simply call $Search(V)$ to locate a set of servers caching V and will not concern the detailed implementation of this function. The servers found through the lookup process form the client's server pool, denoted as $SPool(V)$, from which the client chooses one as its server. To request video V from server S, client C

sends a command $Request(C, V)$ to the server. For simplicity, we also assume each server can have at least one channel. The server maintains a service queue Q; and all arriving requests are first appended to this queue. When a channel becomes free, the server schedules a pending request for services in a FIFO manner, i.e., clients are served according to their arriving order. Thus, given a number of channels and a list of pending requests, we can determine the service latency of each request.

Suppose a client C_i requests a video V_i. The client can call $Search(V_i)$ to find a set of server candidates and then submit its request to the one, say S, that can provide the fastest service. Since client requests are served according to the order of their arrival, C_i needs to wait until S finishes serving all earlier requests. An important objective of our research is to reduce this wait time. This goal can be achieved by trying to move the requests that arrived earlier at S to other servers. Assume client C_j is in the service queue and the video it requests is V_j. C_i can launch a lookup for V_j and check if any server found can provide V_j to C_j no later than S. If there is such a server, say S', then S' can then be used to serve C_j. When a request is moved out of S's queue, all requests pending after this request, including C_i itself, will be served at an earlier time. Since a video may last many minutes, the reduction on their service latency can be significant.

Given a set of server candidates, a client can contact them for their pending requests and try to find each of them a new server. We call this process as *Shaking*. Shaking makes it possible for a client to be served by a server that is beyond the client's search scope. In the above example, S' located by C_i may be invisible to C_j. Given a limited search scope, each client may shake only a small number of servers. However, many small shakes, originating from clients at different locations, together can have a global effect. Since each client can try to shake its server candidates, a more demanded server may be shaken more frequently. Thus, the pending requests in an overloaded server can be migrated gradually to other less loaded servers in the system. Because each shake dynamically adjusts the match between clients and servers, the mismatch caused by limited search scope and unpredictable client arrivals is effectively addressed.

A challenge of implementing the above Shaking idea is the *chaining* effect. In Figure 3.1,

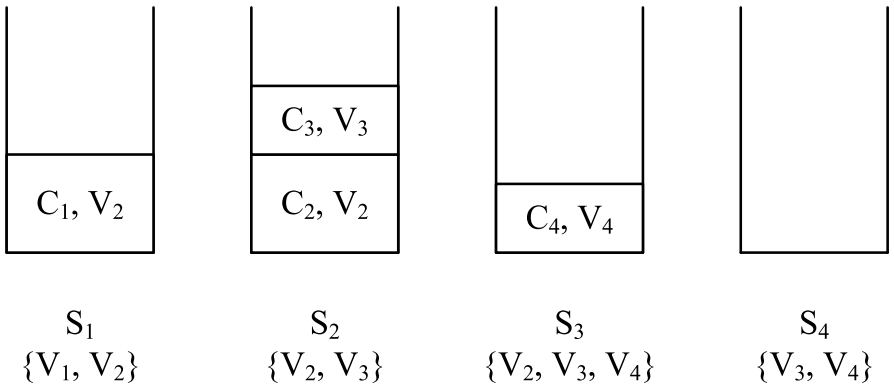


Figure 3.1 Example

a client C trying to shake request $[C_1, V_2]$ out of server S_1 may find two servers, S_2 and S_3, that have V_2. Although these two servers cannot serve $[C_1, V_2]$ earlier than S_1, the client can try to see if it can shake any pending requests out of S_2 and S_3. For example, it may launch a search for V_4 and find server S_4, which is free of workload at this moment. This example shows that in order to successfully move out a request, a client may need to shake a chain of servers. The chain may even consist of loops, which happens when multiple requests pending on different servers for a same video. In addition to the chaining issue, the order of shaking also has significant impact on the shaking results. In the above example, we can move $[C_4, V_4]$ from S_3 to S_4, and then $[C_2, V_2]$ and $[C_3, V_3]$ from S_2 to S_3, and finally $[C_1, V_2]$ from S_1 to S_2. This shaking order allows S_1 to serve client C immediately. However, if we move $[C_3, V_3]$ first from S_2 to S_4, client C will receive no benefit. We address these challenges with a 3-step solution, *building closure set*, *shaking closure set*, and *executing shaking plan*.

3.3.1 Building Closure Set

A closure set is the maximum set of servers that a client can find to shake to minimize its service latency. A client requesting a video V can use Algorithm 2 to build such a closure set. The client first calls $Search(V)$ to find $SPool(V)$, i.e., a set of servers having video V, and contacts each server S for its service queue. Then for each pending request $[C_i, V_i]$ in the queue, the client searches for $SPool(V_i)$. Note that for each video, the client needs to search its servers only once. Given the same lookup mechanism, the client can find only a fixed server pool for a

particular video. The information about the servers and their pending requests found during this process are stored locally.

Algorithm 2 BuildingClosureSet(V)

```
1: ClosureSet = ∅;
2: sList = ∅;
3: vList = ∅;
4: Launch Search(V) to find SPool(V);
5: for all { S | S ∈ SPool(V) } do
6:    sList = sList ⋃ {S};
7: end for
8: while sList ≠ ∅ do
9:    for all { S | S ∈ sList } do
10:      ClosureSet = ClosureSet ⋃ {S};
11:      sList = sList − {S};
12:      Contact S for its service queue Q;
13:      for all { [C_i, V_i] | [C_i, V_i] ∈ Q and V_i ∉ vList } do
14:         Launch Search(V_i) to find SPool(V_i);
15:         for all {S' | S' ∈ SPool(V_i) and S' ∉ ClosureSet } do
16:            sList = sList ⋃ {S'};
17:            vList = vList ⋃ {V_i};
18:         end for
19:      end for
20:   end for
21: end while
```

3.3.2 Shaking Closure Set

Given a set of the servers and their service queues, the client now tries to find a shaking plan that can minimize its service latency. A shaking plan is an ordered list of action items, each denoted as $T([C,V],S_i,S_j]$, meaning that "transfer $[C,V]$ from S_i to S_j". Recall that different shaking orders can have significantly different results. Given a set of servers and their service queues, the client can try different shaking orders to generate various shaking plans and then choose the one that has the best result. Specifically, given a list of servers in $SPool(V)$, say S_1, ..., S_n, the client can try each server as the start point of shaking and generate a shaking plan. Each time it chooses a server, it first appends its request $[C,V]$ in the server's queue and then tries to transfer other requests to other servers. Trying to shake all servers allows the client

to find out which one should be used as its server. Note that all such tries are done locally without actually getting the servers involved.

In order to service a new request $[C_x, V_x]$, a server say S from $SPool(V_x)$ which can provide the fastest service, is chosen and $[C_x, V_x]$ is appended to its service queue. To provide $[C_x, V_x]$ with a faster service, all requests in S that arrived earlier should be shaken out, if possible. For each earlier request $[C, V]$, $Shake([C, V], S)$ is called. When the *Shake* algorithm is invoked on the first request, *ShakingPool* is initialized with the server in which the request is queued. *ShakingSet* for all requests is initially empty. Algorithm 3 gives a formal description on how to shake a given request $[C, V]$ which is currently queued in S. $Latency([C, V], S)$ denotes the expected service latency of $[C, V]$ if it is served by S. *ShakingPool* is the set of servers currently under shaking. *SPlan* denotes the shaking plan generated during this process.

Given a request $[C, V]$, the above algorithm finds $SPool(V)$ first and then checks if any server in $SPool(V)$ can serve V no later than S. If there exists such a server, say S', an action $T([C, V], S, S')$ is appended to the shaking plan. If there is no server, it creates a *ShakingSet* for this request, which contains all servers that are in $SPool(V)$, but not in *ShakingPool*. Note that *ShakingPool* is the set of servers that are currently under shaking. The algorithm then recursively tries to shake out each request in the service queue of the servers in $[C, V]$'s *ShakingSet*.

As an example, consider Figure 3.2. Suppose a client C_x requests video V_x and builds a closure set that contains five servers, S_1, S_2, S_3, S_4, and S_5. The videos cached by these servers and their service queues are shown in the figure. Since V_x is cached only by S_1, $[C_x, V_x]$ is added to S_1's service queue. C_x then tries to create a *ShakingPlan* so that it can be served earlier. It first tries to shake out $[C_2, V_2]$. Since $SPool(V_2)$ contains $\{S_1, S_2\}$ and S_2 can serve $[C_2, V_2]$ earlier than S_1, C_x adds an action $T([C_2, V_2], S_1, S_2)$ to *ShakingPlan*. C_x then tries to shake out $[C_1, V_1]$. Since $SPool(V_1)$ contains $\{S_1, S_2, S_3\}$ and neither one of them can serve $[C_1, V_1]$ earlier than S_1, C_x creates $ShakingSet([C_1, V_1])$, which contains $\{S_2, S_3\}$. $\{S_2, S_3\}$ are added to the *ShakingPool*. C_x starts to shake S_2. $SPool(V_2)$ contains $\{S_1, S_2\}$. However, S_1 cannot serve V_2 earlier. So C_x creates $ShakingSet([C_2, V_2])$. Since $ShakingSet([C_2, V_2]) = \emptyset$ as S_1 is in *ShakingPool*, C_x goes ahead to shake S_3. $SPool(V_3)$ contains $\{S_3, S_4\}$. C_x adds

Algorithm 3 Shake($[C, V]$, S, *ShakingPool*)

1: Get $SPool(V)$
2: $S' \Leftarrow$ {s $\in$ $SPool(V)$ and $latency([C,V],s) \leq latency([C,V],S)$ and $latency([C,V],s)$ is the least among $SPool(V)$ }
3: **if** $S' \neq \emptyset$ **then**
4: Append {T([C,V],S,S')} to *SPlan*
5: *return* S'
6: **else**
7: $ShakingSet([C,V]) =$ {s | s $\in$ $SPool(V)$ and s $\notin$ *ShakingPool* }
8: **if** $ShakingSet([C,V]) = \emptyset$ **then**
9: *return* NULL;
10: **end if**
11: **for all** $\{s \mid s \in ShakingSet([C,V])\}$ **do**
12: $ShakingPool = ShakingPool \bigcup s$
13: **for all** $[C_x, V_x] \in Q(s)$ **do**
14: $Destination([C_x,V_x]) = Shake([C_x,V_x], s,$ *ShakingPool*);
15: **if** $Destination([C_x,V_x]) \neq$ NULL **then**
16: Append $\{T([C_x,V_x], s, Destination([C_x,V_x])\}$ to *SPlan*
17: **if** $latency([C,V],s) \leq latency([C,V],S)$ **then**
18: Append {T([C,V],S,s)} to *SPlan*
19: *return* s
20: **end if**
21: **end if**
22: **end for**
23: $ShakingPool = ShakingPool$ - {s}
24: **end for**
25: *return* NULL;
26: **end if**

S_4 to $ShakingPool$. As S_4 cannot serve $[C_3, V_3]$ earlier than S_4, C_x tries to shake S_4 and adds it to $ShakingPool$. $SPool(V_4)$ contains $\{S_4, S_5\}$. Because S_5 can serve V_4 earlier than S_4, $[C_4, V_4]$ is shaken out to S_4 and an action $T([C_4, V_4], S_4, S_5)$ is appended to $ShakingPlan$. Since S_4 can accept $[C_3, V_3]$, another action $T([C_3, V_3], S_3, S_4)$ is appended to $ShakingPlan$. S_4 is then removed from the $ShakingPool$. Since S_3 can accommodate $[C_1, V_1]$ now, an action $T([C_1, V_1], S_3, S_4)$ is appended to the $ShakingPlan$ and S_3 is removed from $ShakingPool$. Since all requests have been shaken out, C_x proceeds to execute the actions listed in $ShakingPlan$.

3.3.3 Executing Shaking Plan

Given a shaking plan, client C tries to execute the listed actions one by one in order as follows. For each action $T([C, V], S, S')$ in the plan, the client sends server S a message $Transfer([C, V], S')$. Upon receiving such a request, S first checks if request $[C, V]$ is still in its service queue. If it is not, the server sends an *Abort* message to client C. Otherwise, the server sends a message $Add([C, V], L)$ message to S', where L is the expected service latency of $[C, V]$ at S. When S' receives such a message, it checks if it can serve $[C, V]$ in the next L time units. If yes, it appends $[C, V]$ to its service queue and sends to an OK message to S. Otherwise, it sends an *Abort* message to S. When S receives an OK message from S', it removes $[C, V]$ from its service queue and sends a message OK to client C. In the case that S receives an *Abort* message from S', it also sends an *Abort* message to client C. After the client receives an OK message from S, it continues to execute the next action listed in the shaking plan. When client C receives an *Abort* message, it aborts all remaining actions in the shaking plan.

It is worth mentioning that in the above process, the shaking client does not transfer the pending requests directly. Rather, the client can only recommend a list of transferring actions: for each action $T([C, V], S, S')$, the client can only submit it to S. It is S', the destination server, that has the final approval on the transferring action, and S' will not approve unless it can serve the request $[C, V]$ no later than S. There are two advantages of this simple approach. First, it avoids the potential abuse of selfish clients, which may try to generate bogus shaking plans to get earlier services. The above approach ensures that a request cannot be transferred

at the compromise of its service latency. Second, this approach does not require a shaking client to have the latest workload of the servers being shaken. When a client submits an action $T([C,V],S,S')$ to S, S can inform S' of the actual value of L, i.e., the expected service latency of $[C,V]$ at S.

3.4 Discussions

3.4.1 Implementation Issues

The file lookup techniques used in many structured P2P systems (44)(38)(37)(39)(29) allow a client to efficiently locate a server that has the file it requests. When such lookup techniques are adopted for P2P video systems, the cost of building a closure set for a video will not be a major concern. However, if some flooding-based lookup technique is used, a closure set may contain many servers and could be expensive to build. A simple way to address this problem is to apply some threshold control mechanism. For instance, when the number of servers in the closure set exceeds some threshold, the client can stop searching for new servers. Another problem is server crashes. When a server crashes, all requests in its queue are lost. With Shaking, a client request can be transferred from one server to another. Thus, a client may not be aware that its request is lost. To address this problem, each server can periodically update its current clients about their expected service latency. If a client does not receive such information for some time period, it can simply resubmit its request. Finally, a server may be shaken by several clients simultaneously. In the execution of shaking plans, each request transfer is treated as one transaction. Thus, a failed request transfer does not affect the requests that have been transferred successfully. However, when a request transfer fails, the remaining actions in a shaking plan are aborted. This scenario typically happens when a server is included by many clients in their closure sets. This problem can be largely avoided by marking a server when it is included in some closure set. A marked server will then not be included in another closure set for some time period.

3.4.2 Multi-source Shaking

We have assumed previously that a client downloads its video from a single source. In reality, a single source may not be able to upload a video at its playback rate. This is particularly true since many participating peers access the Internet through last-mile connections such as DSL and cable modem, which normally have a very limited uploading bandwidth. We extend our Shaking algorithm for a client to find a set of servers that can provide it with a faster download. Given a client C that requests video V, the client first constructs $SPool(V)$ and selects a set of servers, denoted as *server set*, from this pool. For each server in server set, the client can then try to shake out its pending requests using the shaking approach. The servers selected in server set should be able to allow the client to playback the video at a minimal latency. To construct such a server set, the client needs to compute the service latency given a set of servers and the time at which they can start serving the client. This can be done as follows.

Suppose there are n servers in a server set. Each server S_i in $SPool(V)$ has a bandwidth b_i and a *service time* t_i, which is the time when S_i can start to serve C. Since the client knows the service time and bandwidth of each server in the server set, the client can create a *download schedule*. To minimize service latency, the download schedule should allow the client to download from each server as soon as possible and use all of their bandwidth until the video download is completed. Thus, all servers in server set will finish serving C at the same time. Without loss of generality, we sort these servers in the increasing order of their service times, as shown in Figure 3.3. Let $d_0 = t_0$ and $d_i = t_i - t_{i-1}$, where $i > 0$. After t_i time units, the amount of video data received by the client can be calculated as

$$D_{t_i} = \sum_{j=1}^{i} d_j \sum_{k=0}^{j-1} b_k. \tag{3.1}$$

After t_n time units, all servers are transmitting different parts of the video and they will finish at the same time. Suppose it takes T time units for server S_n to finish transmission. Let b be the play-back rate of the video and L be the length of video. Then T can be calculated as

$$T = \frac{b * L - D_{t_n}}{\sum_{i=0}^{n} b_i}. \tag{3.2}$$

From Equation (3.2), we can find the total time taken for the client to receive the entire video. Let L_t be the total transmission time for video V from all the servers in the server set. Then, L_t is given by,

$$L_t = d_0 + (t_n - t_0) + T. \tag{3.3}$$

Finally, the *Service Latency* for the client, which is the amount of time the client C has to wait before starting to play back the whole video is given by

$$Latency = L_t - L. \tag{3.4}$$

Consider a client C_i that requests for a video V_i and finds a server S that can serve V_i. C_i performs a multi-shake on the requests in the request queue of server S to reduce its latency. Let $[C, V]$ be the request present in the end of the request queue of S_i. C_i performs a search for the set of servers, $SPool(V)$ in its search scope that can serve V. Let *server set([C,V])* be the set of servers currently serving the request $[C, V]$ and *candidate set* denote the set of servers that are in $SPool(V)$ and not currently serving C. C_i continues to add servers from candidate set if the total bandwidth of all the servers in server set([C,V]) is lesser than the download bandwidth of C, b_C. This process is called *Bandwidth Augmentation*. Alternatively, if the aggregate bandwidth of the server set([C,V]) is equal to b_C, we try to improve the utilization of the client's download bandwidth. This can be done by initiating a multi-shake operation on the request queues of the servers in candidate set so that they can reduce the latency of $[C, V]$. Algorithm 4 shows how to shake a request $[C, V]$ in a multi-source model. b_x denotes the bandwidth of a peer x. $(A[C, V], s, ServerSet([C, V])$ denotes a transaction to add the server s to the server set([C,V]).

3.5 Performance Study

Until now, a client requesting for a video first locates a set of servers and then simply chooses the one that can provide the least service latency. We call this approach *Naive* and use it as a baseline to compare with *Shaking* in our performance study. We simulate a decentralized P2P

Algorithm 4 Multi-Shake($[C, V]$, S)

1: *Server set(C,V)* = current server set of the request $[C, V]$
2: $b_{serverset}$ = bandwidth of serverset of the request $[C, V]$
3: *Original Serverset* = *Server set(C,V)*
4: Find $SPool(V)$
5: *Candidate Set* = {s | s ∈ $SPool(V)$ and s ∉ *Server set(C,V)* }
6: **while** $b_{serverset} < b_C$ and *Candidate Set* ≠ ∅ **do**
7: Choose that server s that reduces the latency the most when added to $b_serverset$.
8: *Candidate Set* = *Candidate Set* - s.
9: Add s to *Server set(C,V)*
10: Append {A([C,V],s,Server Set([C,V]))} to $SPlan$
11: **if** $b_{serverset} + b_s > b_c$ **then**
12: $b_{serverset} = b_{serverset} + b_c - b_s$
13: **else**
14: $b_{serverset} = b_{serverset} + b_c$
15: **end if**
16: **end while**
17: **if** there are servers remaining in *Candidate Set* **then**
18: *Shaking Set* = {s | s ∈ *Candidate Set* and s ∉ *Server set(C,V)* }
19: **if** {$ShakingSet([C, V]) = \emptyset$ } **then**
20: return
21: **end if**
22: **for all** server $s \in$ *Original Serverset* **do**
23: **for all** server $s_s \in$ *Shaking Set* **do**
24: **if** s replaced by s_s in *Original Serverset* can reduce latency more than s **then**
25: Append { T([C,V],s,s_s) } to $SPlan$
26: **else**
27: **for all** $[C_x, V_x] \in Q(s_s)$ **do**
28: Multi-Shake($[C_x, V_x]$,$ShakingPool$)
29: **if** s replaced by s_s in *Original Serverset* can reduce latency more than s **then**
30: Append { T([C,V],s,s_s) } to $SPlan$
31: return
32: **end if**
33: **end for**
34: $ShakingPool$ = $ShakingPool$ - {s}
35: **end if**
36: **end for**
37: **end for**
38: **end if**

video system, where a number of servers together cache 100 different videos. Without loss of generality, each video lasts 60 minutes and is MPEG-I compressed with a constant playback rate of 1.5 Mbps. We also assume each server has one channel. In reality, a server may have n channels. When a client finds such a server, it can treat it as n virtual servers, each having one channel and the same set of videos. We choose *average service latency* as the performance metric and study how it is affected by these parameters: *request arrival interval*, *number of servers*, *skew of server capacity*, *skew of video replication*, and *skew of video popularity*. Because of the space constrain, we report only the performance results under various request arrival interval and number of servers.

3.5.1 Effect of Request Arrival Interval

In this study, we fixed the number of servers at 500 and varied the request arrival interval from 20 to 100 second per request. We collected two groups of performance data. In the first group, we assume the servers are uniform in their capacity, i.e., each server caches a randomized number of different videos. In the second group, the server capacity has skew of 0.5. Under both settings, the copies of a video available in the system are generated proportionally to its popularity. Note that this is the perfect status pursued by many file replication techniques. Figure 3.5 shows the performance results. It shows that under all scenarios, Shaking outperforms Naive to a great extent. For example, when the request interval is 100 seconds/request, Shaking achieves 0 seconds of service latency. In contrast, Naive still incurs about 100 seconds of latency. When the request arrival interval increases (i.e., the request rate decreases), the service latency under both techniques reduces. However, the reduction rate under Shaking is more significant. Importantly, the figure shows that Naive is quite sensitive to the server skew and performs better when there is no skew. In contrast, the performance of Shaking is not affected much by the skew of server capacity. As Shaking dynamically adjusts the match between clients and servers, the server workload is effectively balanced. Thus, it can be regarded as a dynamic load balancing technique.

3.5.2 Effect of Number of Servers

In this study, we fixed the request arrival interval at 60 seconds/request and varied the number of servers from 100 to 1,000. Similarly, we collected two groups of performance data, one without server capacity skew and the other with skew at 0.5. Given a fixed client request rate, increasing the number of servers reduces the average server workload in the system. Again, under all simulation setting, Shaking performs significantly better than Naive. When the number of servers is 100, the average service latency under Shaking is less than 50 seconds. In contrast, Naive incurs nearly 1000 seconds of latency. This study also confirms that Shaking can effectively handle the skew of server capacity. Such capability is important given the fact that the hosts participating in P2P sharing are typically heterogeneous in their caching capacity.

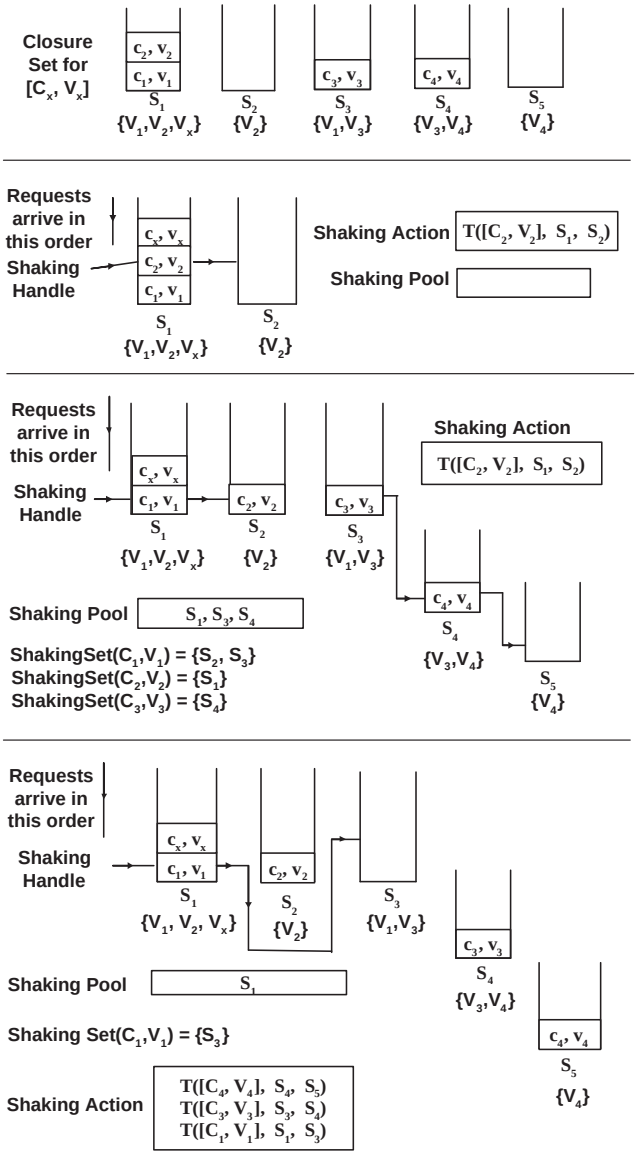


Figure 3.2 Generating Shaking Plan

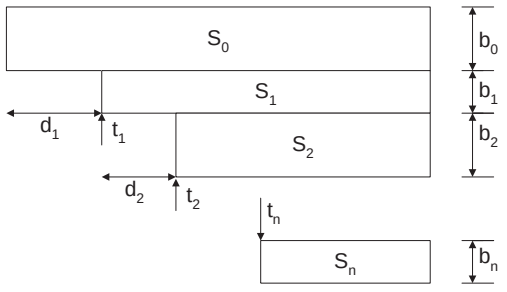


Figure 3.3 The bandwidth and service time of each server in ServerSet

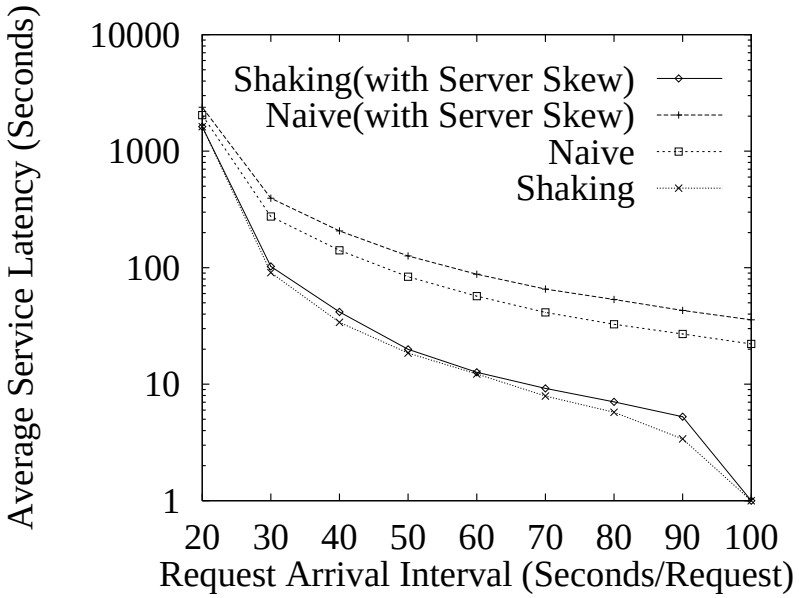


Figure 3.4 Effect of Client Arrival Interval

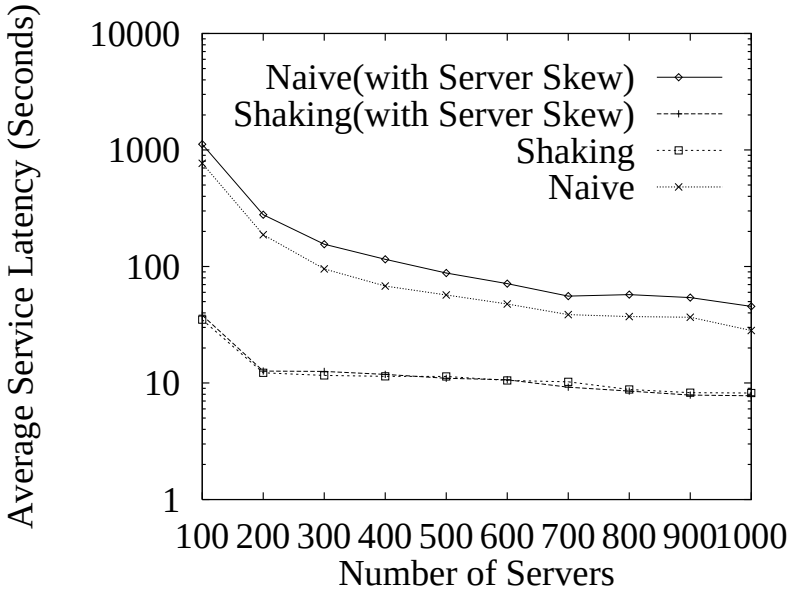


Figure 3.5 Effect of Number of Servers

CHAPTER 4. Conclusion and Future Work

This thesis makes two contributions towards large-scale and cost-effective video services.

- We have presented two novel techniques for periodic broadcast of popular videos. Our first technique, CCA+ enhances an existing periodic broadcast technique CCA and reduces the broadcast latency up to 50% as compared to CCA. Our second technique, CCB is a generalized periodic broadcast approach. At the core of this technique, we propose a segmentation rule that allows us to design a series of broadcast approaches for a particular setting of client bandwidth. CCB chooses the fastest periodic broadcast technique among these approaches. To our knowledge, it is the fastest broadcast technique up to date. We prove that as soon as a client starts to download the first segment of the video, it can start to play back continuously till the end in both these techniques.

- We investigate the problem of service scheduling in P2P video systems and propose a novel technique, called *Shaking*. This technique is characterized by a few desirable features. First, Shaking makes it possible for a client to be served by a server that is beyond the client's own search scope. Second, the match between the servers and clients can be dynamically adjusted to minimize client service latency. Furthermore, the proposed scheme is able to avoid potential abuse of selfish clients, which may try to preempt all earlier requests in a server. The proposed technique can be used in general to improve the performance of regular P2P file sharing systems, which to our knowledge do not consider service scheduling up to date. As indicated by our performance study, an effective scheduling algorithm is critical to the system load balancing and can significantly reduce the average service latency.

We envision extending our research along two directions:

- Adapt CCB for more efficient broadcasting in heterogeneous networking settings. It is worth mentioning that several techniques, including HeRo (8) and BroadCatch (45), have been developed for video broadcasting in an environment wherein clients have vastly different receiving bandwidth.

- Adapt Shaking for segment-based video streaming. The current version of Shaking assumes a client downloads a whole video from one server. In reality, a video may be split into segments and stored in multiple servers or a server may have a whole video but is willing to transmit some part of it (because of the upload bandwidth concerned).

BIBLIOGRAPHY

[1] ABC. http://abc.go.com.

[2] Fox Broadcasting Company. http://www.fox.com.

[3] NBC. http://www.nbc.com.

[4] Youtube Statistcis. http://www.youtube.com/t/press$_$statistics.

[5] V. Agarwal and R. Rejaie. Adaptive multi-source streaming in heterogeneous peer-to-peer networks. In *SPIE Conf on Multimedia Computing and Networking*, 2005.

[6] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. On Optimal Batching Policies for Video-on-Demand Storage Servers. In *Proc. of IEEE Int'l Conf. on Multimedia Systems*, Hiroshima, Japan, June 1996.

[7] S. Annapureddy, S. Guha, C. Gkantsidis, D. Gunawardena, and P. R. Rodriguez. Is high-quality vod feasible using p2p swarming? In *Proceedings of the 16th international conference on World Wide Web*, pages 903–912, New York, NY, USA, 2007. ACM.

[8] O. Bagouet, K. A. Hua, and D. Oger. A periodic broadcast protocol for heterogeneous receivers. In *Proc. of SPIE Conf. on Multimedia Computing and Networking (MMCN'03)*, pages 27–38, Berkeley, CA, January 2004.

[9] Y. Cai and J. Zhou. An Overlay Subscription Network for Live Internet TV Broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1711–1720, December 2006.

[10] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2), Feb. 1988.

[11] J. Chakareski and P. Frossard. Utility-based packet scheduling in p2p mesh-based multicast. In *in Proc. of SPIE International Conference on Visual Communication and Image Processing (VCIP09*, 2009.

[12] S. J. Chapin. Distributed and multiprocessor scheduling. *ACM Computing Surveys*, 28, 1996.

[13] 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.

[14] Y.-h. Chu, S. G. Rao, and H. Zhang. A case for end system multicast (keynote address). *SIGMETRICS Perform. Eval. Rev.*, 2000.

[15] Y. Cui, B. Li, and K. Nahrstedt. ostream: Asynchronous streaming multicast in application-layer overlay networks. *IEEE Journal on Selected Areas in Communications*, 22:91–106, 2003.

[16] 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.

[17] A. Dan, D. Sitaram, and P. Shahabuddin. Dynamic Batching Policies for an On-Demand Video Server. *Multimedia Systems*, 4(3):112–121, June 1996.

[18] C. Dana, D. Li, D. Harrison, and C. Chuah. Bass: Bittorrent assisted streaming system for video-on-demand. In *Proc. of IEEE 7th Workshop on Multimedia Signal Processing*, pages 1–4, 2005.

[19] 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.

[20] L. Gao, J. Kurose, and D. Towsley. Efficient Schemes for Broadcasting Popular Videos. *Journal of Multimedia Systems*, 8(4):284–294, 2002.

[21] Y. Guo, S. Mathur, K. Ramaswamy, S. Yu, and B. Patel. Ponder: Performance aware p2p video-on-demand service. In *Proc. of IEEE GLOBECOM 2007*, pages 225–230, 2007.

[22] C.-H. Hsu and M. Hefeeda. Quality-aware segment transmission scheduling in peer-to-peer streaming systems. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, MMSys '10, 2010.

[23] K. A. Hua, Y. Cai, and S. Sheu. Exploiting Client Bandwidth for More Efficient Video Broadcast. In *Proc. of Int'l Conf. on Computer Communication and Networking*, pages 848–856, Louisiana, U.S.A, October 1998.

[24] 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.

[25] K. A. Hua, Y. Cai, and S. Sheu. Leverage Client Bandwidth to Improve Service Latency of Distributed Multimedia Applications. *Journal of Applied Systems Studies (JASS)*, 2(3):686–704, 2001.

[26] K. A. Hua and S. Sheu. Skyscraper Broadcasting: A New Broadcasting Scheme for Metropolitan Video-On-Demand Systems. In *Proc. of ACM SIGCOMM*, Cannes, France, Sepetember 1997.

[27] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, J. W. O'Toole, Jr., M. Frans, and K. James. Overcast: Reliable multicasting with an overlay network. In *In Proceedings of Operating Systems Design and Implementation*, pages 197–212, 2000.

[28] L. Juhn and L. Tseng. Harmonic broadcasting for video-on-demand service. *IEEE Transactions on Broadcasting*, 43(3):268–271, Sept. 1997.

[29] 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.

[30] M. Kwon and S. Fahmy. Topology-aware overlay networks for group communication. In *In Proc. of ACM NOSSDAV*, pages 127–136. ACM Press, 2002.

[31] B. W. Lampson. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 11:347–360, 1968.

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

[33] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, A. E. Mohr, and E. E. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *in IPTPS*, pages 127–140, 2005.

[34] J. Paris. A Fixed-Delay Broadcasting Protocol for Video-on-Demand. In *Proc. of Int'l Conf. on Computer Communication and Networking*, pages 418–423, October 2001.

[35] J. Paris and D. Long. Limiting the Receiving Bandwidth of Broadcasting Protocols for Videos on Demand. In *Proc. of Euromedia 2000 Conference*, pages 107–111, May 2000.

[36] J. F. Paris, S. W. Carter, and D. D. E. Long. Efficient Broadcasting Protocols for Video on Demand. In *Proc. of SPIE's Conf. on Multimedia Computing and Networking (MMCN'99)*, pages 317–326, San Jose, CA, USA, January 1999.

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

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

[39] 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 (SOSP'01)*, pages 188–201, Alberta, Canada, 2001.

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

[41] J.-P. Sheu, H.-L. Wang, C.-H. Chang, and Y.-C. Tseng. A fast video-on-demand broadcasting scheme for popular videos. *IEEE Transactions on Broadcasting*, 50:120–125, June 2004.

[42] S. Sheu, K. A. Hua, and W. Tavanapong. Chaining: A Generalized Batching Technique for Video-On-Demand. In *Proc. of Int'l Conf. On Multimedia Computing and System*, pages 110–117, Ottawa, Ontario, Canada, June 1997.

[43] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

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

[45] M. A. Tantaoui, K. A. Hua, and T. T. Do. BroadCatch: A Periodic Broadcast Technique for Heterogeneous Video-on-Demand. *IEEE Transaction on Broadcasts*, to appear.

[46] D. A. Tran, K. Hua, and T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming. In *In Proc. of IEEE Infocom*, 2003.

[47] 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.

[48] Y.-C. Tseng, M.-H. Yang, and C.-H. Chang. A recursive frequency-splitting scheme for broadcasting hot videos in vod service. *IEEE Transactions on Communications*, 50(8):1348– 1355, August 2002.

[49] S. Viswanathan and T. Imielinski. Metropolitan Area Video-on-Demand Service Using Pyramid Broadcasting. *Multimedia systems*, 4(4):179–208, August 1996.

[50] A. Vlavianos, M. Iliofotou, and M. Faloutsos. Bitos: Enhancing bittorrent for supporting streaming applications. In *INFOCOM*, 2006.

[51] H.-F. Yu, H.-C. Yang, and L.-M. Tseng. Reverse Fast Broadcasting(RFB) for Video-on-Demand Applications. *IEEE Transactions on Broadcasting*, 53(1):103–111, 2007.

[52] M. Zhang, Y. Xiong, Q. Zhang, L. Sun, and S. Yang. Optimizing the throughput of data-driven peer-to-peer streaming. *IEEE Trans. Parallel Distrib. Syst.*, 20(1), Jan. 2009.

[53] X. Zhang, J. Liu, B. Li, and T. shing Peter Yum. Coolstreaming/donet: A data-driven overlay network for peer-to-peer live media streaming. In *in IEEE Infocom*, 2005.