On Scheduling of Peer-to-Peer Video Services

Ying Cai, *IEEE member,* Ashwin Natarajan, *Student IEEE member,* Johnny Wong, *IEEE member.*

Abstract— Peer-to-peer (P2P) video systems provide a cost-effective way for a large number of hosts to colloaborate for video sharing. Two features characterize such a system: 1) a video is usually available on many participating hosts, and 2) different hosts typically have different sets 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 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? In this paper, 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 this paper, 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.

Index Terms— P2P Systems, Video Services, Service Scheduling, Service Latecy.

I. INTRODUCTION

Consider a decentralized *Peer-to-Peer* (P2P) overlay network that consists of a large number of hosts that collaborate for the purpose of providing video services to each other. The hosts participating in such a system, which we will refer to as *P2P video system*, are *peer* to each other in the sense that they can all behave as clients and servers. While a host can be a client in requesting video services from any other hosts, it can also be a server by caching a number of videos to serve the entire community. Without causing ambiguity, we will simply use *client* to refer to a host requesting a video and *server* a host supplying a video. Since a client can download a video for its own playback and then cache it to serve future requests, a highly demanded video can be quickly spread out in the system. When a video can be supplied by many hosts from different network domains, the bottleneck formed at the original server is effectively alleviated.

A P2P video system can be built on top of a structured or unstructured P2P file sharing systems. Since many techniques have been proposed for efficient file lookup in such systems (e.g., *K-random walk* [1], *CAN* [2], *Chord* [3], *Pastry* [4], just to name a few), we will not concern ourselves on how a client can locate the servers that have the video it requests. In this paper, we focus on *service scheduling*, another component that

is critical to a P2P video system. Given a client requesting a video, which is typically available on a number of servers, which server should be used to serve the client? The problem of service scheduling is complicated because of several factors, the first one being obvious while the rest less apparent:

- From a client's perspective, it should be served by the one with minimal *service latency*, which is defined to be the period starting from the time a client submits a video request to the time it can start to download for playback. 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.
- Even if each client can find all available server candidates, different match 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.
- 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.

In this paper, we address the challenges of scheduling peer-to-peer video services. Although our discussion is in the context of video services, the proposed solution is generic and can be applied in any P2P file sharing systems. In P2P systems, two clients looking for a same file may locate two different sets of server candidates. This phenomenon is especially popular when a file is cached by many hosts in the system. Based on this observation, we develop a novel scheduling technique, called *Shaking* [5], which is characterized by two features. First, a client can be now be served by a server that is beyond the client's own search scope. Second, the matches between servers and their pending clients can now be dynamically adjusted as new requests arrive in the system. As we will show, these two features can dynamically balance server workload in the system and minimize client service latency.

II. SYSTEM MODEL AND ASSUMPTION

We assume a decentralized P2P system, where no central server is used for file management. Since efficient file lookup has been studied intensively and many techniques have been proposed [1], [2], [2], [3], [4], 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.

At the server side, each server organizes its communication bandwidth into a number of *channels*, each of which can stream a cached video at its playback rate to a remote client. Many P2P users connect to the Internet through some form of broadband residential connections, such as ADSL or cable modem, where the communication bandwidth are asymmetric with uploading bandwidth significantly less than download-ing bandwidth. For such network access, the communication bottleneck is in its connection to the ISP, instead of in the Internet backbone [6], [7]. In these cases, a number of hosts can work as a single server by aggregating their uploading bandwidth to serve a client [8]. Thus, in our discussion, we simply assume each server can have at least one channel. We also assume that each 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.

III. SHAKING

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 by clients from
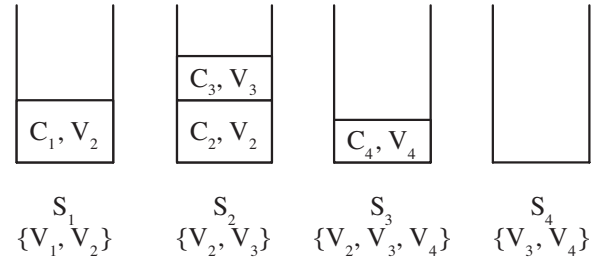

Fig. 1.   Example

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 1, 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*.

A. 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 video V can use Algorithm 1 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 a 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.

B. 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 $[C, V]$

Algorithm 1 BuildingClosureSet(V)

1: $ClosureSet = \emptyset$;
2: $sList = \emptyset$;
3: $vList = \emptyset$;
4: Launch $Search(V)$ to find $SPool(V)$;
5: **for all** $\{S \mid S \in SPool(V)\}$ **do**
6:   $sList = sList \bigcup \{S\}$;
7: **end for**
8: **while** $sList \neq \emptyset$ **do**
9:   **for all** $\{S \mid S \in sList\}$ **do**
10:     $ClosureSet = ClosureSet \bigcup \{S\}$;
11:     $sList = sList - \{S\}$;
12:     Contact S for its service queue Q;
13:     **for all** $\{[C_i, V_i] \mid [C_i, V_i] \in Q \text{ and } V_i \notin vList\}$ **do**
14:       Launch $Search(V_i)$ to find $SPool(V_i)$;
15:       **for all** $\{S' \mid S' \in SPool(V_i) \text{ and } S' \notin ClosureSet\}$ **do**
16:         $sList = sList \bigcup \{S'\}$;
17:         $vList = vList \bigcup \{V_i\}$;
18:       **end for**
19:     **end for**
20:   **end for**
21: **end while**

Algorithm 2 Shake($[C, V]$, S)

1: Get $SPool(V)$
2: $S' \Leftarrow \{s \in SPool(V) \text{ and } latency([C, V], s) \leq latency([C, V], S) \text{ and } latency([C, V], s) \text{ is the least among } SPool(V)\}$
3: **if** $S' \neq \emptyset$ **then**
4:   Append $\{T([C,V],S,S')\}$ to $SPlan$
5:   *return* S'
6: **else**
7:   $ShakingSet([C, V]) = \{s \mid s \in SPool(V) \text{ and } s \notin ShakingPool\}$
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:       $Destination([C, V]) = Shake([C_x, V_x], s)$;
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**

---

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 2 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 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 S_4 to $ShakingPool$. As S_4 cannot serve $[C_3, V_3]$ earlier
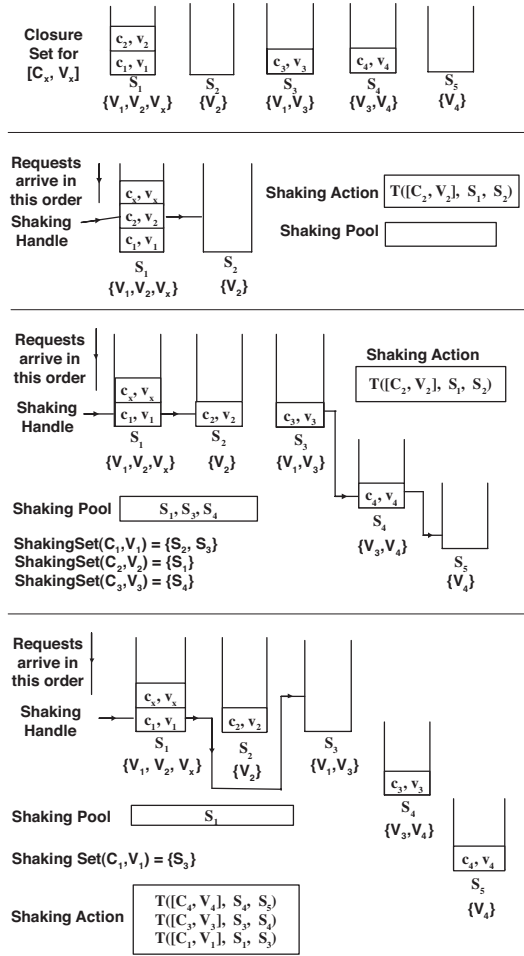
Fig. 2. Generating Shaking Plan.

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 $ShakingPlan$.

C. 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.

IV. DISCUSSIONS

A. Implementation Issues

The file lookup techniques used in many structured P2P systems [3], [4], [2], [9], [10] 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 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.

B. Multi-source Shaking

So far we have assumed a client downloads its video from a single source. In reality, a 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. It is possible to extend our Shaking algorithm for a client to find a set of servers that can provide it with the fastest download. Given a client C that requests video V, the client first constructs $SPool(V)$ and selects a set of servers, denoted as $ServerSet$, from this pool. For each server in $ServerSet$, the client can then try to shake out its spending requests using our early algorithm. The servers selected in $ServerSet$ should be able to allow the client to playback the video at a minimal latency. To construct such a $ServerSet$, the client needs to compute the service latency given a set of servers. This can be done as follows.

Suppose there are n servers in a $ServerSet$. Each server S in $SPool(V)$ has a bandwidth b and a service time t, which is the time when S can start to serve C. Since the client knows the service time and bandwidth of each server in the $ServerSet$, 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 them until the video download is completed. Thus, all servers $ServerSet$ 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 showed in Figure 3. Given a server S_i, we denote its bandwidth and service time as b_i and t_i, respectively. 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

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

After t_n time units, all servers start transmission 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

$$T = \frac{b * L - D_{t_n}}{\sum_{i=0}^{n} b_i}. \tag{2}$$

From Equation (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 $ServerSet$. Then, L_t is given by,

$$L_t = d_0 + (t_n - t_0) + T. \tag{3}$$

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

$$Latency = L_t - L. \tag{4}$$

V. PERFORMANCE STUDY

Until now, a client requesting for a video first locates a set of servers and then simple chooses the one that can provides the least service latency. We call this approach *Naive* and use
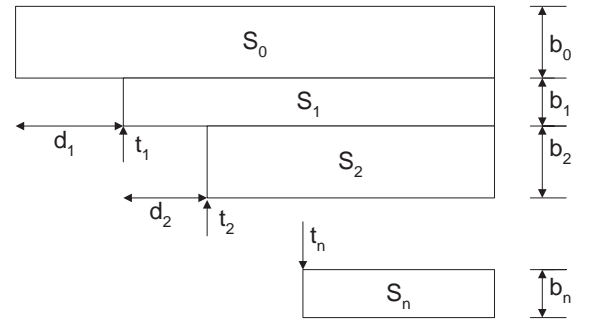

Fig. 3. The bandwidth and service time of each server in ServerSet

it as a baseline to compare with $Shaking$ in our performance study. We simulate a decentralized P2P 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.

A. 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 is generated proportionally to its popularity. Note that this is the perfect status pursued by many file replication techniques. Figure 5 shows the performance results. It shows that under all scenarios, Shaking outperforms Naive to a great extend. For example, when the request interval is 100 seconds/request, Shaking achieves 0 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.

B. 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
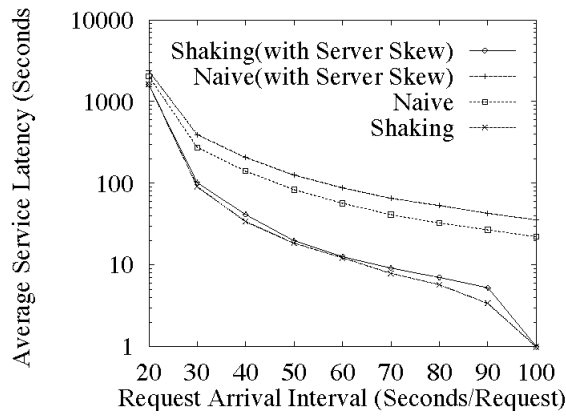
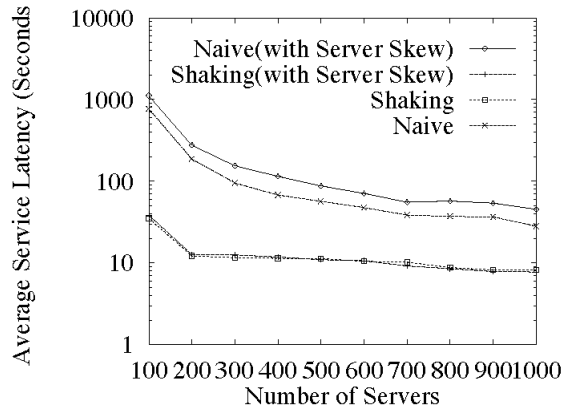Fig. 4.   Effect of Client Arrival Interval

Fig. 5.   Effect of Number of Servers

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.

VI. Concluding Remarks

In this paper, we investigate the problem of service scheduling in P2P video systems and propose a novel technique, called *Shaking*. The proposed 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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] Ying Cai, Ashwin Natarajan, and Johnny Wong. Shaking Service Requests in P2P Video Systems. In *Proc. IEEE GLOBECOM 2005*, St.Louis, MO., October 2005.

[6] 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'5*.

[7] S. Ganguly, A. Saxena, S. Bhatnagar, S. Banerjee, and R. Izmailov. Fast Replication in Content Distribution Overlays. In *Proc. of INFOCOM'5*.

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

[9] 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.

[10] 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.

Ying Cai received his Ph.D. in computer science from the University of Central Florida in 2002. Currently, Dr. Cai is an assistant professor in the Department of Computer Science at the Iowa State University. His research interests include wireless networks, mobile computing, and multimedia systems.

Ashwin Natarajan received his B.E in Computer Science and Engineering from National Institute of Technology, Trichy, India. He received his M.S in Computer Science from Iowa State University in 2004 and is currently pursuing his Ph.D. His research interests include Media Streaming, P2P Systems and Wireless networks.

Johnny Wong is a Professor & Associate Chair of the Computer Science Department at Iowa State University. He received his Ph.D. in Computer Science from University of Sydney, Australia. His research interests include Software Systems & Networking, Security & Multimedia Systems. Most of his research projects are funded by government agencies and industries. He has served as a member of program committee of various international conferences on intelligent systems and computer networking. He co-chaired the COMPSAC 2006 conference and has published over 100 papers in peer reviewed journals and conferences, and is a member of IEEE Computer Society and ACM.