Improving QoS in BitTorrent-like VoD Systems

Yan Yang
Univ. of Southern California

Alix L.H. Chow
Univ. of Southern California

Leana Golubchik
Univ. of Southern California

Danielle Bragg
Harvard University

Abstract—**In recent years a number of research efforts have focused on effective use of P2P-based systems in providing large scale video streaming services. In particular, live streaming and Video-on-Demand (VoD) systems have attracted much interest. While previous efforts mainly focused on the common challenges faced by both types of applications, there are still a number of fundamental open questions in designing P2P-based VoD systems, which is the focus of our effort. Specifically, in this paper, we consider a BitTorrent (BT)-like P2P VoD system and focus on the following questions: (1) how the lack of load balance, which typically exists in a P2P-based VoD system, affects the system performance and what steps can be taken to remedy that, and (2) is a FCFS approach to serving requests at a peer sufficient or whether a Deadline-Aware Scheduling (DAS) approach can lead to performance improvements. Given the deadline considerations that exist in VoD systems, we also investigate approaches to avoid unnecessary requests and queuing time. For each of these questions, we first illustrate deficiencies of current approaches in adequately meeting streaming Quality of Service (QoS) requirements. Motivated by this, we propose several practical schemes aimed at addressing these questions. To illustrate the benefits of our approach, we present an extensive simulation-based performance study of our techniques.**

I. Introduction

In recent years peer-to-peer (P2P) systems have become an effective approach to video content distribution. In particular, many P2P *live* streaming systems, such as PPLive [1] and CoolStreaming [2], have been deployed and gained wide popularity. Because of their scalability characteristics, significant research effort (e.g., [3], [4], [5], [6], to name a few) has been focused on the design of P2P systems for live streaming. These and other works provide insight into performance, scalability, dynamics (and other characteristics) of P2P live streaming systems (see Section VI for a more detailed overview and relationship to our work).

More recently, interest has also turned toward another important and technically challenging application, specifically Video-on-Demand (VoD) service for high-quality, full-length movies. Examples of such systems include Joost, Hulu, Netflix, iTunes Store, and so on. Naturally, P2P-based designs have been considered in the context of VoD applications as well. For instance, the study in [7] shows that the MSN video server load can be reduced by $\approx 95\%$ through the use of a P2P-based approach. A P2P-based design of VoD applications is also the focus of our work.

Before describing our contributions, we first note that there are a number of fundamental differences between VoD and live streaming. VoD applications have a greater data diversity than live streaming applications. In live streaming systems, nodes typically request data around a particular playback point - that is, users are watching the stream at around the same time. In contrast, in VoD systems nodes request videos at different times, and thus their playback points differ greatly. One implication of this is that nodes in live streaming applications only need a playback buffer of several minutes (as peers are clustered around the same playback point), while nodes in VoD applications may need to hold the entire movie (as each node tends to have a different playback point). Another implication is that playback deadlines of file pieces in VoD have a larger variance than those in live streaming. This is partly due to (a) the playback points of nodes being dependent on their arrival time (which is diverse) and (b) nodes in VoD systems potentially requesting data relatively far into the future (from their playback point), as VoD data is pre-recorded (in contrast to data being generated live).

In this paper, we consider a BitTorrent (BT)-like P2P design, as BT is one of the more popular P2P systems which (according to [8]) accounts for an astounding 35% of all the traffic on the Internet (more than all other P2P applications combined). BT is also a highly efficient system, e.g., the study in [9] illustrates that it makes nearly optimal use of peers' upload bandwidth. Briefly, in BT, nodes join the system (after receiving "start-up" information from the tracker) and begin requesting pieces of data from their neighbors. The tracker maintains a list of nodes which are currently participating in the corresponding torrent. It is responsible for assisting in peer discovery and is not involved in any data transfer or data scheduling. A node, i, which does not have a complete copy of the file picks a number of peers to whose requests it will respond with an upload of an appropriate data piece. A subset of these nodes are picked based on the tit-for-tat (TFT) mechanism, i.e., those neighbors which have provided the best service (in terms of download rate) to node i recently. And a subset is picked randomly, to explore better neighbors. Nodes which have a complete copy of the file may stay in the system for some time and continue to upload data to their peers. All these choices are re-evaluated periodically.

Because of its success, a number of previous efforts, including [10], [11], [4], [12], have focused on adapting BT (originally designed for file downloads) to VoD applications. To adapt BT for streaming, a number of fundamental issues need to be addressed. To illustrate the difference between using BT for file downloads and for VoD streaming, we briefly summarize two such issues, which are studied, e.g., in [10], [11], [4], [12]. Firstly, the default piece selection strategy used in BT is not well suited for VoD applications, as BT uses a rarest first strategy to determine which piece a peer should request

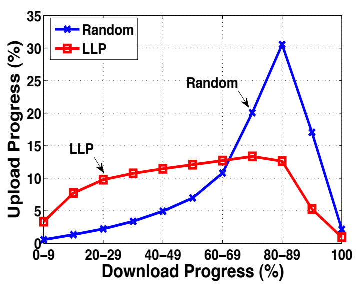

Fig. 1. Upload vs. Download

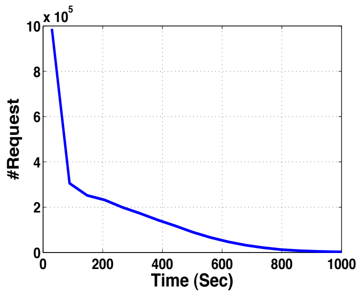

Fig. 2. Request Deadline Distribution

next. As such a strategy does not consider realtime playback constraints of video, it is unlikely to lead to good quality of service (QoS), as shown in [12], [3]. Secondly, BT's built-in incentive mechanism, TFT (as described above), is not well suited for VoD systems due to the asymmetric peer relationship in VoD applications, i.e., young nodes can download from old but not vice versa, as shown in [10]. Approaches to address these and other issues in the context of both live streaming and VoD systems are discussed in a number of works, including [7], [13], [14], and [15]. However, several open fundamental questions still remain, which are particular to P2P-based VoD systems, and (to the best of our knowledge) previous efforts either do not address them (at least not for BT-like systems) or do not propose solutions that lead to real implementations. We describe two such fundamental issues next, and focus on exploring practical solutions to the corresponding problems in the remainder of the paper.

One such question is *to which peer should a node send a request for a data piece*, among all neighbors which have that piece; we will refer to this as the "*peer request problem*". For instance, simply picking such a neighbor at random has the disadvantage that older peers (nodes which arrived earlier and have a larger fraction of the content) receive more requests from many younger peers (nodes which arrived later). Figure 1 depicts percentage of upload performed by a node as a function of percentage of download completed in a typical P2P VoD system, obtained by simulation (refer to Section II). Compared to an "ideal" load balancing scheme (referred to as "LLP" and explained in Section III), randomly picking the neighbor to which a request is sent results in unbalanced workload over the age of a node, as $\approx 50\%$ of uploads occur after 80% of the download is completed. Consequently, the data request load is unevenly distributed among the peers, (intuitively) leading to losses in QoS because: (1) requests sent to overloaded older peers suffer from long waiting times, and (2) the wasted bandwidth of young peers reduces the overall system capacity and further slows down the streaming process. Such a load balancing problem is briefly discussed in [10], with possible solutions based on an approach proposed in [16] but in the context of file downloading rather than VoD streaming (see Section VI for details). We study the peer request problem in Section III (where we also compare our approaches to other approaches, e.g., those suggested in [10], [17], [18]).

Another fundamental question is *which request for data in its queue should a peer serve next*, among all the requests for data made to that peer; we will refer to this as the "*service scheduling problem*". Typically, in a BT-like VoD system, as in [10], each node can serve up to a certain number of requests concurrently, with the remaining requests waiting in a queue until a service slot becomes available. As noted earlier, the playback deadlines of the requested data pieces in a VoD system are quite diverse, some having urgent deadlines and others being more relaxed. This diversity can be observed in Figure 2, which depicts a request deadline distribution obtained from simulations (refer to Section II). We can see that although many requests have a very short deadline, e.g., $\approx 33\%$ of requests have deadlines within 50 seconds, there are some requests that have very long deadlines, where the longest deadline for a request is ≈ 1000 seconds. Intuitively, a service discipline that would reduce the probability of a data piece missing its playback deadline should result in better QoS. In [10], FCFS scheduling is suggested. We study this scheme in Section IV and show that better solutions (which take advantage of deadline diversity) are possible.

Another question within the service scheduling context is *whether all requests for data pieces should be served*, and if not, which ones should be rejected. Existing VoD designs, including [19], [11], [10], [12], accept and serve every request. A natural question is – if a request will (likely) miss its deadline, why waste resources by serving it? The disadvantages of not identifying and rejecting such requests are that: (a) receiving such a request wastes upload resources (as this request will likely miss its deadline and not be played) and (b) the request in question might be able to receive timely service from another (perhaps less loaded) peer. Intuitively, in either case, it might be best to reject the request in question which should result in better QoS overall. We study this service rejection problem in Section IV.

In summary, here we focus on the two questions stated above. We first motivate these questions through examples. We then propose practical approaches to address them. We also show that solely addressing one of these questions is not sufficient for achieving high QoS, and thus a good P2P VoD streaming system should consider both. For ease of exposition, we first present our schemes in a homogeneous environment (i.e., where nodes have equal upload capacities). We then show how simply our schemes can be adapted to heterogeneous environments (refer to Section V).

Thus, the contributions of this work are as follows:

- We explore practical solutions to the "peer request problem" (described above), that can be easily implemented through fairly small modifications to the current BT protocol - these approaches result in better QoS in the VoD system and at the same time are scalable, efficient, and easily deployable today (see Section III).
- We propose the use of Deadline-Aware Scheduling (DAS) which includes an earliest deadline first (EDF) scheduling approach and an early drop (EDP) based approach to address the "service scheduling problem". We show that DAS results in better QoS in a VoD system. To the best of our knowledge, this work is the first to explore the use

of earliest deadline first and early drop approaches in the context of BT-like VoD systems (see Section IV).

- We show that addressing the "peer request problem" or "service scheduling problem" alone is not sufficient to achieve high QoS, i.e., that an appropriate combination of good solutions to each question is needed in a P2P VoD streaming system to provide high QoS with low overhead. To support this claim, we present an extensive evaluation study on the use of these approaches under a variety of environments (see Section IV).

II. Performance Metrics and Experimental Setup

We explore and evaluate solutions to the above stated questions through simulations, using the BT simulator provided by [9] (also used by other groups for BT related research). This is an event-based simulator, originally developed to simulate the piece exchange mechanism in the BT protocol. To explore our proposed approaches for BT-like VoD systems, we modify the simulator in [9] as follows:

- We remove BT's default piece exchange mechanism (to adapt it to VoD streaming), and implement the data piece request and service mechanisms of Sections III and IV.
- We let each node send up to D requests to peers concurrently[1]; each peer serves U of the the incoming requests, with the remainder placed in a queue.
- Nodes start their playback after a startup delay, s. After that, playback proceeds at the rate of r without interruption. If a piece is not received before its playback time, it is marked as missing[2].
- Each node serves requests until it finishes playback. Once it finishes the playback, the remaining requests in the queue are discarded and need to be reissued by the requesting nodes. This emulates a user quitting the system in the real world.
- We allow node arrivals; in what follows we use a Poisson arrival process with rate λ.
- There is one initial server in the system and it stays in the system for the duration of the simulation. Each node can request a data piece from this server, if that piece cannot be found among the peers.

Unless otherwise stated, the results that follow correspond to simulation settings given in Table I. All experiments simulate a BT-like VoD system for 30 hours. To isolate effects of our proposed approaches, we first consider the in-order piece selection strategy, i.e., nodes requesting pieces according to their playback order. We then study our proposed approaches under mixed selection (i.e., nodes requesting some pieces according to playback order and some based on their rarity) in Section IV. By default, each node serves its incoming request queue using FCFS policy.

[1]Increasing D allows a node to request data pieces further into the future at the cost of causing longer queues at peers, thereby increasing waiting time; a detailed exploration of this parameter is outside the scope of this paper.

[2]A small s result in lower startup delay but also in poorer video quality; a large s improves continuity but also increases startup delay (and both are aspects of QoS). A detailed exploration of s is outside the scope of this paper.

TABLE I
Simulation Settings

Simulation Time	30 hours
Avg node inter-arrival time ($\frac{1}{\lambda}$)	60 sec
Movie Encoding Rate	500 Kbps
Startup Delay (s)	10 sec
Piece Size	256 KB
File Size	400 MB (1600 pieces)
Peer Set Size	40
Node Max #Upload Connection (U)	5
Node Max #Concurrent Request (D)	10
Node Capacity (Down / Up)	5000 Kbps / 512 Kbps
Server Max #Upload Connection	5
Server Upload Capacity	5000 Kbps

For a fair comparison between approaches, we use the same node arrival sequence for each simulation with a given arrival rate. In experiments where nodes randomly select to which peer to send a request, we also use the same selection sequence. In what follows, unless otherwise stated, we focus on the continuity index (CI), defined in [5], as our main metric for video viewing quality, where:

$$CI = \frac{\#\text{total pieces} - \#\text{total missing pieces}}{\#\text{total pieces}}.$$

A higher CI implies better video playback quality.

III. Peer Request Problem

We begin with a simple example which illustrates the poor viewing quality that can occur in an unbalanced system (i.e., where requests are not evenly distributed among nodes and only a subset of the nodes serve most requests). Motivated by this, we explore approaches to balancing the request load, to improve playback quality.

Motivating Example: for ease of exposition, we perform an experiment using a homogeneous set of nodes, where we use the following peer request policies.

Random (Rand): Each node sends the request to a randomly chosen neighbor which has the needed data piece. This is a typical approach used in other works, e.g., [10], and we use it as a default/baseline case.

Least Loaded Peer (LLP): Each node sends a request to the neighbor with the shortest queue size, among all those that have the needed data piece, randomly breaking ties. We use this as an ideal case, to mimic perfect load balancing.

The resulting CDF of corresponding CIs is depicted in Figure 3, where we observe that the viewing quality (as indicated by CI) under LLP is significantly better than that of Random. Specifically, the average LLP CI is ≈ 0.97 while it is only ≈ 0.68 for Random. The standard deviation for LLP and Random is ≈ 0.07 and ≈ 0.12, respectively, which is also an indication that peers are likely to get more "stable" QoS under LLP than under Random. This is due to the more balanced distribution of requests under LLP. Intuitively, in an unbalanced system, some nodes will have a long incoming request queue. Requests sent to these nodes will experience long waiting times, which will increase the probability that (1) pieces miss their playback deadlines, (2) waiting for service of delayed pieces prevents timely requesting of other pieces, and (3) upload bandwidth of lightly loaded nodes is wasted, thereby reducing the overall system capacity. We sample nodes in our experiments and observe that under LLP, the node queue size is significantly more stable than under Random and tends to be smaller (e.g., the queue size of node #900 under LLP is always below 10 requests while that of node #900 under Random goes beyond 35 requests).

To understand why the load is unbalanced under Random, we observe percentage of upload performed as a function of the percentage of download completed[3]. The results are given in Figure 1, where we observe:

- Under Random, most of the uploads occur in the later stages of the download process, e.g., $\approx 50\%$ of uploads occur after 80% of download is completed. This is due to old nodes having more pieces, resulting in a higher probability of older nodes receiving requests. This also verifies a point made in [10] that older nodes are often overloaded.
- Under LLP, the uploads are more evenly distributed during the downloading process, e.g., $\approx 10\%$ of the uploads occur between 20% and 30% of download completion (in contrast to only $\approx 3\%$ in the Random case). This is due to nodes always sending requests to a neighbor with the shortest queue (under LLP) which helps spread the load among peers more evenly.

Motivated by this, we now focus on load balancing techniques. Conceptually, LLP would be a simple approach to load balancing; however, it is difficult to implement, as it requires exact knowledge of *instantaneous* node queue lengths. We could approximate it, by obtaining information about peers' queue sizes; however, that results in a tradeoff between message overhead (for updating such information) and resulting system performance. We explore this tradeoff in detail below. But, first, we experiment with a straightforward approach, which approximates LLP without the need for updates, to understand whether high QoS can be achieved without overhead. Specifically, we use:

Least Requested Peer (LRP): For each neighbor, each node counts how many requests it has sent to that peer and picks the one with the smallest count, randomly breaking ties.

As noted in the above example, it is the older nodes that tend to get overloaded. An approach to load balancing, based on the notion of peer age but in the context of file downloads rather than streaming, is proposed in [16] using "*stratification*". Conceptually, stratification attempts load balancing by insuring that peers of age t only download from peers of age $t + \Delta$. Adapting the notion of stratification to VoD systems is suggested in [10][4], and a similar idea to stratification is also proposed in [17], using tracker support. (Details of how [17] and [10] differ from our work can be found in Section VI.)

[3]Note that, with 100% of download completed, upload continues if a node has not completed the video playback.

[4]In [10], the suggestion of using stratification-type approaches is only made at a high level, without evaluation or sufficient scheme details for implementation. We tried experimenting with their suggested schemes, using reasonably straightforward implementation, and their performance was not as good as the schemes explored here. Due to lack of space, we do not present detailed results here.

TABLE II
DIFFERENT LOAD BALANCING SCHEMES

	Rand	YNP	CNP	LRP	Tracker	LLP
Average CI	0.678	0.785	0.786	0.716	0.785	0.966
Std. Deviation	0.104	0.155	0.157	0.143	0.097	0.066
CI Improvement (%)	-	15.82	15.91	5.59	15.77	42.56

For comparison purposes, we include an approach similar to that in [17] in our experiments below which can be described as follows.

Tracker Assistant (Tracker): The tracker sorts peers according to their arrival times. Whenever a node requests the list of available nodes from the tracker (e.g., upon arrival or when lacking peers due to peer departures), it will receive a list of nodes which have the closest arrival times to its own.

To explore the notion of load balancing by not overloading older peers, we also propose the following schemes.

Youngest-N Peers (YNP): Each node sorts its neighbors according to their age, where a peer's age can be determined from its join time (available at the tracker). YNP then randomly picks a peer among the $N > 1$ youngest peers which have the piece of interest and requests that piece from that neighbor. This approach tries to send requests to younger peers as they are less likely to be overloaded. We choose randomly among a subset of youngest peers, rather than the youngest one, as choosing the youngest one may lead to many nodes sending their requests to the same youngest peer (thus potentially overloading it).

Closest-N Peers (CNP): This approach more closely emulates the stratification behavior described above. It is similar to YNP but instead sorts the neighbors based on how close they are to a node's own age, and then randomly picks from the N closest age peers that have the needed piece.

Table II reports the average CI, standard deviation (STD), and the improvement in the average CI as compared to Random for all the approaches discussed above. Figure 4 depicts the average CI as a function of N for YNP (the results for CNP are very similar to YNP and are thus omitted). We make the following observations:

- YNP and CNP give significant performance improvements as compared to Random, for good choices of N. Similar performance is also achieved by Tracker. (Although Tracker does not require picking a good N, it performs worse than YNP and CNP when we apply our scheduling improvements in Section IV.)
- YNP (and CNP) can be quite sensitive to the choice of N as seen in Figure 4. If N is too small, YNP and CNP risk overloading a few peers, but with very large values of N (approaching a node's neighbor set size), YNP and CNP degenerate to Random (as also observed in our experiments). In our later experiments, we fix $N = 15$ when we use YNP/CNP. We reduce the sensitivity to choice of N through approaches presented in Section IV.
- LRP does not perform as well as the other schemes, which indicates that this straightforward approach to approximating LLP is not sufficient. Adding randomization (as in YNP and CNP) might help but is outside the scope of this paper.

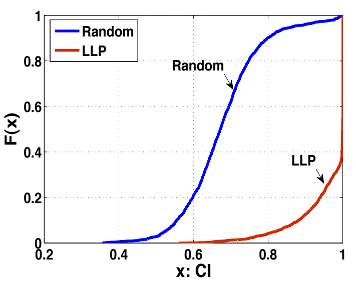

Fig. 3. CI (Random, LLP)

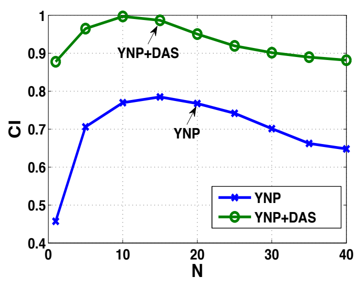

Fig. 4. Different N

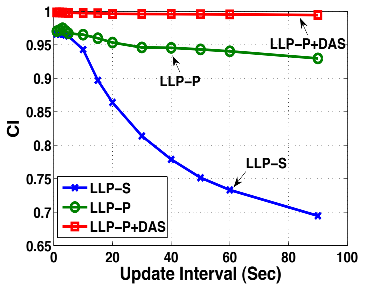

Fig. 5. CI (LLP-S, LLP-P)

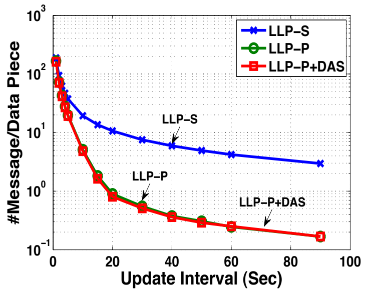

Fig. 6. LLP Update Overhead

- LLP gives the best performance among all schemes. However, since it is difficult to implement in practice (as noted above), we next consider its implementable approximations that perform better than LRP (but at the cost of update overhead).

LLP with Stale Information (LLP-S): One possible implementation of LLP is to let each node report its queue length to its neighbors periodically, which we term *LLP-S*[5]. Not surprisingly[6] this results in a tradeoff between information freshness and update overhead. Figure 5 shows LLP-S performance and Figure 6 shows the corresponding message overhead, *per data piece*, plotted on a *log* scale. With a small update interval (e.g., 5 seconds), LLP-S performs well but at the cost of high message overhead (e.g., ≈ 38 messages per data piece in the case of a 5 second update interval). Under longer update intervals, LLP-S's performance drops quickly, e.g., it performs similarly to Random when the update interval is increased to 90 seconds (with a corresponding message overhead of ≈ 3 messages per data piece).

LLP Piggyback (LLP-P): because of a relatively high overhead of using LLP-S (even with larger update intervals), we propose *LLP Piggyback* (LLP-P) which is suitable for *BT-like* VoD system. In a BT-like system, when a node receives a data piece, it sends out a *Have* message to all its neighbors. We *piggyback* our LLP update messages on these *Have* messages, thus reducing the additional message overhead. Since it is possible that no *Have* message is sent out by a node for a long period of time (e.g., a node experiencing slow download or one that downloaded all pieces), we still include explicit update messages in LLP-P, when no update message has been sent (either explicitly or through piggybacking) for T_l time units. Due to lack of space, we give a formal description of LLP-P in [20].

Figures 5 and 6 depict LLP-P's performance and corresponding message overhead, respectively, where we observe:

- With piggybacking, the update message overhead is significantly reduced, e.g., message overhead corresponding to a 30 second update interval is only ≈ 0.6, as compared to ≈ 7.6 without piggybacking.
- With piggybacking, the average CI is *less* sensitive to the update interval - it only drops $\approx 4\%$ when going from a 5 second to a 90 second interval. This indicates that we can use a larger update interval without significant performance degradation, which is due to the already frequent updates achieved through piggybacking on *Have* messages (these were measured to be sent out, on the average, every ≈ 4 seconds in the simulation).

[5]Studying malicious behavior in queue length reporting is outside the scope of this paper.

[6]This is also noted in, e.g., [18], with differences between [18] and our work explained in Section VI.

Rather than relying solely on the "Have" messages, one could also piggyback updates on other messages, e.g., piece requests, streaming data, etc. We expect this can further reduce the overhead while improving CI.

Although, as demonstrated in this section, a good load balancing scheme is important, significant room for CI improvement remains. Next, we study how service scheduling affects system performance and propose approaches to further improve CI.

IV. Service Scheduling Problem

In [10], the authors show, using their model, how to bound delay in a BT-like VoD system, under the FCFS queuing policy, which is related to the second question we posed in Section I, i.e., what service scheduling policies are better suited for VoD systems and perhaps under what environments. It includes two sub-problems: (i) in what order should requests be served and (ii) whether some requests should be rejected. To address these, we propose *Deadline-Aware Scheduling* (DAS) which considers the requests' deadline constraints.

As argued in Section I, the requested pieces' deadlines (in a VoD system) are quite diverse (refer to Figure 2). Hence, a node's request queue contains a mix of requests, those with urgent deadlines and those with less urgent ones. In such situations a FCFS policy may not work well as illustrated by the following example.

Peer Service Example: we are given two requests A and B which arrive in order (see Figure 7), where each request takes 1 time unit to serve. A's deadline expires in $t+2$ time units and B's deadline in $t+1$ time units. Using FCFS, A is served first and B is served 1 time unit later. Unfortunately, then B misses its deadline as by the time A's service completes, B's deadline passes. However, if the service policy is deadline aware, B could receive service before A, and A would still make its deadline. As a result, both A and B make their deadlines.

Earliest Deadline First (EDF): the above simple example illustrates the benefits of a deadline-aware policy. Motivated by this, we use *earliest deadline first* (EDF) policy. Under EDF, each node maintains a queue sorted by the request deadline

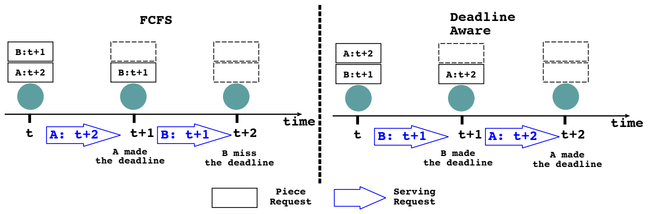

Fig. 7. Piece Service Example

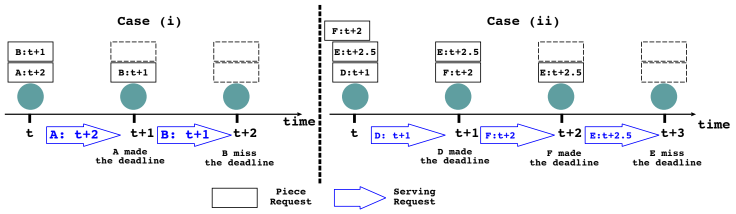

Fig. 8. Service Rejection Example

and picks the request with the most urgent deadline to serve first. In our experiments, each node stamps a request with information of the remaining time until playback point for that piece[7]. Upon receipt of the request, the serving peer extracts this information and uses it as the request deadline.

Service Rejection Example: we return to the above example and extend it with two additional cases illustrated in Figure 8. In Case (i), we have another request, C, which has a deadline in $t + 2.5$ and arrives after B is served. Since it is less urgent than A, according to EDF, C will be served after A, which would be at $t + 3$, resulting in C missing its deadline. In this case, we should not serve C so that it can try other peers. In Case (ii), at time t, there are two requests, D and E, in the queue, with respective deadlines of $t + 1$ and $t + 2.5$. Then, request F arrives, and its deadline is $t + 2$. Since D is the most urgent one, according to EDF, it will be served first, after which (at time $t + 1$), F will be served, since it is more urgent than E. After F is served, which is at time $t + 2$, we find that there is no way for E to make its deadline (as its service takes unit time). In this situation, we should have accepted service of F but dropped E, as E is less urgent and thus has more time to look for service from other peers.

Early Drop (EDP): The question before us now is how to avoid wasting a request's time, waiting time in the queue, if (given the current load on that peer) it cannot be served on time. In particular, (1) if a request cannot be served on time, inserting it into the queue wastes resources/time that can be used by other requests, and (2) inserting a new request into the queue may change the waiting time of existing requests (when EDF is used), suggesting that we should re-evaluate existing requests to see if they can still be served on time.

To address these issues, we propose the *Early Drop* (EDP) policy which works as follows. We first estimate the waiting time of a newly arrived request, using currently available bandwidth and the request load already in the queue that can affect the newly arrived request (i.e., this is based on the request's deadline and the service policy used, e.g., FCFS vs. EDF). If it is determined that the newly arrived request can make its deadline, it is inserted into the queue (according to the service policy). At that point, we estimate (in a similar manner) the waiting time of all requests that were already in the queue before the new arrival and ended up being queued behind the new arrival (i.e., after it was inserted into the queue). If some of these requests will now miss their deadlines, these requests are dropped from the queue, and the peers that made the original requests can try to obtain the corresponding pieces from other peers. Thus, our approach tries to drop requests from the queue as early as possible, i.e., as soon as it is determined that they will miss their deadline. We give a more formal description of EDP in [20].

[7]Malicious nodes can attempt to forge deadlines, thus making their requests more urgent. Studying of malicious behaviors and corresponding prevention schemes is part of future efforts. We note that a simple detection can reduce the effectiveness of such exploits, e.g., a node can estimate deadlines for peers' requests based on their request history or join time.

TABLE III
DIFFERENT LOAD BALANCING SCHEMES WITH DAS

	Rand	YNP	LRP	Tracker	LLP-P	LLP
Average CI	0.929	0.982	0.887	0.975	0.996	0.998
Std. Deviation	0.104	0.155	0.143	0.097	0.005	0.066
CI Improvement (%)	37.02	25.10	23.88	24.20	5.29	3.31
Msg. Overhead	1.959	0.549	6.904	1.830	0.430	0.053

Deadline-Aware Scheduling (DAS): Given our deadline considerations, it is (intuitively) useful to combine EDF and EDP, and we term the combined scheme *Deadline-Aware Scheduling* (DAS). Here, we study the effect of DAS under different load balancing schemes discussed in Section III[8]. In Table III we report the average CI, standard deviation, and the improvement in the average CI as compared to the case without DAS. We set $N = 15$ in YNP; LLP-P's update threshold is set to 40 second (based on our earlier experiments). We omit CNP results as they are quite similar to those of YNP. We observe the following:

- All load balancing schemes show significant improvement with DAS applied. When DAS is used in LLP or LLP-P, we can achieve CI of nearly 1, which indicates the importance of including DAS, even when good load balancing schemes are used.
- Figure 4 depicts the average CI of YNP as a function of N. We found that YNP is less sensitive to N when DAS is used, which is a highly desirable property. The reason is that, even if peers decide to send requests to a small subset of neighbors, under DAS, these requests still have a high chance of either being served before their deadlines or being reissued to a "better" peer. Even as N grows large, we no longer see the big drop in CI that occurs when DAS is not used. This makes this policy more practical to implement than YNP alone (e.g., we can safely use reasonably small values of N).
- Figure 5 depicts the average CI of LLP-P as a function of the update interval threshold. We found that LLP-P with DAS is less sensitive to the update interval threshold and can achieve a CI close to 1 even with a large interval. The

[8]The performance of EDF or EDP acting alone can be found in [20].

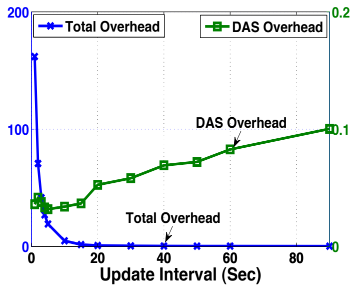

Fig. 9. Overhead (LLP-P+DAS)

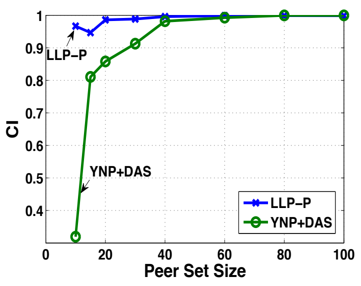

Fig. 10. Peet Set Size (CI)

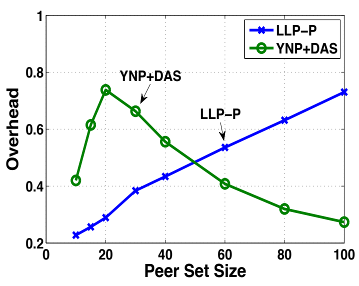

Fig. 11. Peer Set Size (Overhead)

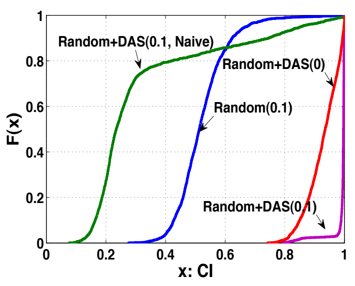

Fig. 12. CI (Mixed Selection)

update message overhead of LLP-P with DAS is similar to LLP-P without DAS as shown in Figure 6.

DAS Overhead: use of EDP introduces additional message overhead, i.e., piece requests have to be reissued, when requests are dropped by a peer. Such request dropping may cause a chain effect in request reissuing, as a reissued request may cause additional drops when it arrives at another peer, etc. To evaluate the overhead due to DAS, in our experiments, we measure the number of requests sent per data piece under different load balancing schemes and report[9] them in Table III, where we observe:

- In general, the better load balanced is the scheme, the lower is the DAS overhead, since a more balanced scheme can "direct" the requests to "better" peers which cuts down on reissueing of requests.
- Under LLP-P, increasing the update interval threshold increases the DAS overhead as shown in Figure 9, which is due to the reduced effectiveness of LLP-P under larger update intervals. However, the total message overhead of LLP-P is still dominated by LLP update messages. Thus, the total message overhead increases when we use a smaller update threshold as depicted in Figure 9. Our measurements indicate that the total message overhead of LLP-P with DAS is similar to that without DAS.

Is LLP-P Always Preferred?: above LLP-P shows good performance under small message overhead; thus, a natural question is whether LLP-P should be the scheme of choice. In BT, each node has a list of neighboring peers and the overhead of LLP-P depends on (1) how often it sends explicit update messages, and (2) to how many neighbors it needs to send these messages (i.e., its peer set size). To study how peer set size affects LLP-P overhead, we vary each node's peer set size from 10 to 100 and compare LLP-P to YNP. Figure 10 depicts the resulting average CI, and Figure 11 shows the corresponding message overhead, where we observe:

- With a larger peer set size, the average CI of both LLP-P and YNP is improved, as a larger number of peers increases piece availability among neighbors which helps load balancing schemes.
- With a larger peer set size, message overhead of LLP-P increases. This is due to LLP update messages being sent to more peers. By contrast, message overhead of YNP reduces because better load balancing through larger peer size helps reduce reissuing of requests. Therefore, under larger peer set sizes, YNP can outperform LLP-P in terms of message overhead while both YNP and LLP-P can achieve similar CI.

Mixed Piece Selection: for ease of exposition, above we evaluated our approaches using in-order piece selection for determining which data piece to request. Mixed piece selection is studied in the literature, e.g., [12], [3], [11], where most techniques can be summarized as a combination of rarest-first selection (mainly for piece diversity) when deadlines are not urgent and in-order selection (mainly for making deadlines) when they become urgent. It has been shown that mixed piece selection improves performance, when done properly. Thus, we also evaluated our load balancing approaches under mixed piece selection; since the results were qualitatively similar to those presented earlier, we omit them here. Instead, we focus on combing mixed piece selection with our DAS approaches.

Under EDP, a node keeps searching for a peer to serve a request, and eventually obtains the piece on-time, unless no peer has that piece or those peers who do, are too overloaded to make the deadline. Using a mixed strategy can help reduce the probability of not being able to obtain a piece on-time. However, a naive implementation of mixed selection under DAS does not work well. Specifically, the rarest-first selection part of mixed piece selection conflicts with EDF, as rarest-first selection typically requests pieces far away from the current playback point that have more slack time than the normal in-order request. Consequently, such requests end up at the back of the queue and end up waiting a long time to be served.

To make DAS work well with mixed piece selection, we use two request queues (per node), one for in-order requests and one for rarest-first requests. In-order requests are served using EDF, and rarest-first requests are served using FCFS[10]. When a service slot becomes available, we consider the first request in the rarest-first queue - if serving that request does not result in a missed deadline for a request in the in-order queue, then we go ahead and serve it; otherwise, we pick a request from the in-order queue (using EDF). We give the details of adapting DAS for mixed piece selection in [20].

Let p be the probability of selecting a rarest piece and $1-p$ be the probability of doing in-order selection, in the

[9]For LLP-P, we show the total message overhead which includes both, LLP update messages and DAS request reissue messages.

[10]This is done to respect the motivation of requesting pieces that are rare at request time. For the same reason we limit the rarest-first queue.

TABLE IV
MESSAGE OVERHEAD UNDER DAS AND MIXED PIECE SELECTION

	Rand	YNP	LRP	Tracker	LLP-P
Msg. Overhead	3.669	0.837	7.125	3.600	0.758
Overhead Inc. (%)	87.29	52.46	3.20	96.72	76.28

mixed strategy. To evaluate our approach, we perform the following experiments: (1) Random using mixed selection with $p = 0.1$[11]; (2) Random using mixed selection with DAS, and $p = 0.1$ using naive (one queue per node) implementation; (3) Random using in-order selection with DAS; (4) Random using mixed selection with DAS, with $p = 0.1$ and the two-queues per node adaptation described above. Figure 12 depicts the corresponding results, where we observe:

- Without the separate request queues, the performance of DAS is quite poor under mixed piece selection - the average CI is only ≈ 0.33 as compared to ≈ 0.68 without DAS. In contrast, the proposed two-queues per node adaptation gives significant improvements - the average CI is ≈ 0.99 as compared to ≈ 0.68 without DAS.
- DAS performs better under mixed selection than under in-order selection; the average CI is ≈ 0.99 as compared to ≈ 0.93, which is due to better piece diversity under mixed selection, with later pieces having higher availability under mixed selection than under in-order piece selection.

We experimented with other load balancing schemes using DAS with mixed selection. All schemes showed significant improvements, with the average CI of LRP being ≈ 0.97 and that of other schemes being ≈ 0.99. Thus, even schemes with relatively poor performance before, such as Random and LRP, using mixed selection with DAS can achieve a CI similar to more load balanced schemes. However, this comes at the cost of higher message overhead (when compared to using in-order selection) as depicted in Table IV. Under mixed piece selection, part of the system resources are shifted to serving rare pieces, which reduces the service rate of the in-order pieces and increases the corresponding queue length at nodes. This, in turn, increases the chance of a request reissue.

V. HETEROGENEOUS ENVIRONMENT

So far, we focused on a homogeneous environment, which enabled a simpler exposition and clearer evaluation of our schemes. However, nodes in the real world have different capabilities (e.g., upload capacity). The different upload capacities affect load balancing characteristics, e.g., faster nodes can finish servicing requests in their queues before the slower nodes. Thus, the load balancing schemes need to be adjusted, to account for the heterogeneity in node capacities. We show how to adjust the LLP and YNP schemes; other load balancing schemes can be modified similarly[12].

LLP-HLB: LLP related schemes consider nodes' queue length as a way to balance load and thus reduce response times. In a

[11]We pick $p = 0.1$ as it is a typical value evaluated in the literature. Exploring other values of p or other mixed selection schemes, e.g., as in [12], is outside the scope of this paper

[12]Studying how to provide upload incentives in a heterogeneous environment is outside the scope of this paper.

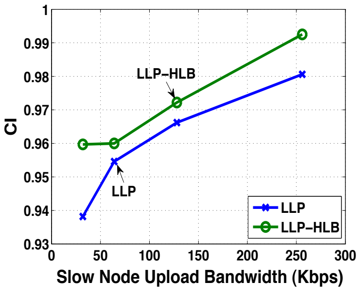

Fig. 13. LLP (Heterogeneous)

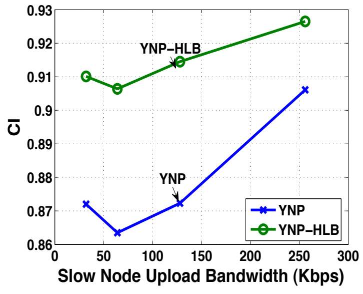

Fig. 14. YNP (Heterogeneous)

TABLE V
HETEROGENEOUS SETTINGS

Slow Node Upload BW (Kbps)	32	64	128	256
Fast Node Upload BW (Kbps)	992	960	896	768

heterogeneous environment, the response times of nodes also depend on their upload bandwidth. Thus, a natural way to adapt LLP is to consider the amount of time it would take to respond to all requests at a node (rather than just its queue length), which is proportional to $\frac{\text{node queue length}}{\text{upload bandwidth}}$; we term this scheme *LLP-HLB*.

We evaluate LLP-HLB under a variety of heterogeneous settings given in Table V, with arrival probabilities of fast and slow nodes being the same (hence the average system capacity remains roughly the same across the settings). Figure 13 shows the average CI comparison between LLP and LLP-HLB, where CI drops when nodes have a larger disparity in upload bandwidth and LLP-HLB shows improvements over LLP. However, these improvements are not large, which indicates that LLP adapts to heterogeneous environments fairly well; this is due to fast nodes clearing their request queues faster (i.e., having a shorter queue), which results in more requests being directed to them. We also observed similar results for LLP-S and LLP-P, with details found in [20].

YNP-HLB: in the case of YNP, instead of *randomly* choosing among N youngest peers, we adapt it to make this choice based on weighted probabilities, where the weights are proportional to the corresponding nodes' upload capacities. We term this YNP-HLB. Figure 14 demonstrates the improved performance of YNP-HLB over YNP in various heterogeneous settings. As expected, the HLB adaptation has a significantly greater affect on YNP, as YNP has no information about loads at different peers, and hence does not naturally account for heterogeneity like LLP.

Since our HLB schemes depend on knowledge of neighbors' upload capacities, the above experiments assume the nodes have perfect knowledge of peers' upload bandwidth. In a real systems, errors can occur in bandwidth estimation. We examined the sensitivity of our schemes by introducing 20% and 40% errors in peer's bandwidth information. In both cases, we found that the impact on the system performance was negligible (refer to [20]).

VI. RELATED WORK

Design of VoD systems has received attention from systems, networking, and signal processing communities for a number of years. Early efforts mainly used traditional client-server architectures and focused on how to efficiently disseminate video content from the server to clients; this includes efforts such as patching [21], periodic broadcastings [22], stream merging [23], and so on. Recently, much of the research focus in this area shifted to P2P-based designs, e.g., [7], [19], [15], [14]. In [7] authors conduct measurements and a simulation study, using data traces from the MSN video service, and show that a P2P based approach can greatly reduce server cost. In [19] authors discuss challenges and design issues of PPLive, a popular P2P streaming system, with millions of real world users, and [15] studies a P2P VoD system consisting of set-top boxes in a homogeneous DSL network. The work in [14] proposes a tree based P2P VoD system which organizes nodes by arrival time and tries to combine advantages of tree-based and mesh-based overlays. While these works provide interesting insight into P2P-based VoD design, to the best of our knowledge, only a few of them consider the fundamental questions posed in this paper, which we discuss below.

Parvez et al. in [10] briefly suggest use of stratification in VoD systems, proposed in [16] in the context of downloads, which they suggest to achive by: (1) sending multiple copies of the same request to several peers so that eventually all requests are served by a subset of faster peers, (2) sending requests to the peers with faster response time, based on historical information so that after some time a node only sends requests to a subset of peers which provide fast response, and (3) having nodes maintain a limited buffer around the playback point (rather than the entire video content downloaded thus far) so that nodes only serve a subset of peers which have "nearby" playback points. These are not evaluated in [10], and as noted above, our evaluation of some of these schemes did not lead to as good performance as those studied here.

Authors in [18] propose a *DHT-based* design to balance request load. They use a scoring function, to select the peer with the lowest cost to serve a request, which takes into consideration a peer's information such as bandwidth, current load, and online time. At a high level, this is similar to our LLP-based schemes; however, there is not sufficient information provided (e.g., about how to weigh the different parameters of the scoring function), for us to be able to make quantitative comparisons to this approach. Also, they only focus on the level of load balancing as their performance metric, while we focus on the resulting QoS. Moreover, as we showed in Section IV, a good load balancing scheme is not sufficient for high QoS; hence, our proposed DAS improvement. Liang et al. in [17] use tracker assistance to improve performance. We evaluated this scheme in Section III and showed that our proposed schemes have better performance with lower overhead.

The piece selection problem in BT-like systems is studied, e.g., in [11], [4], [12], [3]. While piece selection is not the focus of our work, we evaluated how our proposed approaches are affected by the different piece selection strategies.

Another category of works related to ours is those focused on P2P live streaming, e.g., [2]. Although theoretical analysis and measurement studies of P2P live streaming systems provide insight into the design of VoD systems as well, as noted in Section I, there are fundamental differences between these applications, which give rise to some of the fundamental questions studied here.

VII. Conclusions

We posed and studied two fundamental questions in the context of BT-like VoD systems and proposed practical approaches to addressing these questions. Our extensive simulation-based study showed that our approaches can provide significant improvements in QoS in BT-like VoD system.

References

[1] *PPLive*. http://www.pplive.com.

[2] S. Xie, B. Li, G. Y. Keung, and X. Zhang, "Coolstreaming: Design, theory, and practice," *IEEE Trans. on Multimedia*, vol. 9, no. 8, December 2007.

[3] Y. Zhou, D. M. Chiu, and J. C. S. Lui, "A simple model for analyzing p2p streaming protocols," in *ICNP*, 2007.

[4] A. Vlavianos, M. Iliofotou, and M. Faloutsos, "Bitos: Enhancing bittorrent for supporting streaming applications," in *INFOCOM Workshop*, 2006.

[5] X. Zhang, J. Liu, B. Li, and T. S. P. Yum, "Coolstreaming/donet: A data-driven overlay network for efficient live media streaming," in *INFOCOM*, 2005.

[6] N. Magharei, R. Rejaie, and Y. Guo, "Mesh or multiple-tree: A comparative study of live p2p streaming approaches," in *INFOCOM*, 2007.

[7] C. Huang, J. Li, and K. Ross, "Can internet video-on-demand be profitable," in *SIGCOMM*, 2007.

[8] *Peer-to-peer in 2005*. CacheLogic.

[9] A. Bharambe, C. Herley, and V. Padmanabhan, "Analyzing and improving bittorrent performance," in *INFOCOM*, 2006.

[10] N. Parvez, C. Williamson, A. Mahanti, and N. Carlsson, "Analysis of bittorrent-like protocols for on-demand stored media streaming," in *SIGMETRICS*, 2008.

[11] Y. R. Choe, D. L. Schuff, J. M. Dyaberi, and V. S. Pai, "Improving vod server efficiency with bittorrent," in *Multimedia*, 2007.

[12] K. W. Hwang, V. Misra, and D. Rubenstein, "Stored media streaming in bittorrent-like p2p networks," *Tech Report, Columbia University, NY*, no. cucs-024-08, 2008.

[13] M. H. Hefeeda, B. K. Bhargava, and D. K. Y. Yau, "A hybrid architecture for cost-effective on-demand media streaming," *Elsevier Computer Networks*, vol. 44, 2004.

[14] M. Zhou and J. Liu, "Tree-assisted gossiping for overlay video distribution," *ACM Multimedia Tools and Applications*, vol. 29, no. 3, 2006.

[15] K. Suh, C. Diot, J. Kurose, L. Massoulie, C. Neumann, D. Towsley, and M. Varvello, "Push-to-peer video-on-demand system: design and evaluation," *IEEE JSAC*, vol. 25, no. 9, 2007.

[16] A. Gai, F. Mathieu, F. D. Montgolfier, and J. Reynier, "Stratification in p2p networks: Application to bittorrent," in *ICDCS*, 2007.

[17] C. Liang, Z. Fu, Y. Liu, and C. W. Wu, "ipass: Incentivized peer-assisted system for asynchronous streaming," in *INFOCOM Mini Conf.*, 2009.

[18] K. Graffi, S. Kaune, K. Pussep, A. Kovacevic, and R. Steinmetz, "Load balancing for multimedia streaming in heterogeneous peer-to-peer systems," in *NOSSDAV*, 2008.

[19] Y. Huang, T. Z. J. Fu, D.-M. Chiu, J. C. S. Lui, and C. Huang, "Challenges, design and analysis of a large-scale p2p-vod system," in *SIGCOMM*, 2008.

[20] Y. Yang, A. Chow, L. Golubchik, and D. Bragg, "Improving QoS in bittorrent-like vod systems," http://vista.usc.edu/pub/vod-tech.pdf, Tech. Rep.

[21] L. Gao, D. Towsley, and J. Kurose, "Efficient schemes for broadcasting popular videos," in *NOSSDAV*, 1998.

[22] A. Hu, "Video-on-demand broadcasting protocols: a comprehensive study," in *INFOCOM*, 2001.

[23] D. Eager, M. Vernon, and J. Zahorjan, "Bandwidth skimming: a technique for cost-effective video-on-demand," in *MMCN*, 2000.