Peer-assisted On-demand Streaming of Stored Media using BitTorrent-like Protocols*

Niklas Carlsson and Derek L. Eager

Department of Computer Science, University of Saskatchewan,
Saskatoon, SK S7N 5C9, Canada
{carlsson, eager}@cs.usask.ca

Abstract. With BitTorrent-like protocols a client may download a file from a large and changing set of peers, using connections of heterogeneous and time-varying bandwidths. This flexibility is achieved by breaking the file into many small pieces, each of which may be downloaded from different peers.

This paper considers an approach to peer-assisted on-demand delivery of stored media that is based on the relatively simple and flexible BitTorrent-like approach, but which is able to achieve a form of "streaming" delivery, in the sense that playback can begin well before the entire media file is received. Achieving this goal requires: (1) a piece selection strategy that effectively mediates the conflict between the goals of high piece diversity, and the in-order requirements of media file playback, and (2) an on-line rule for deciding when playback can safely commence. We present and evaluate using simulation candidate protocols including both of these components.

Keywords: BitTorrent-like systems, peer-assisted streaming, probabilistic piece selection.

1 Introduction

Scalable on-demand streaming of stored media can be achieved using scalable server protocols such as patching [1] and Hierarchical Stream Merging [2], server replication as with CDNs, and/or peer-to-peer techniques. This paper concerns peer-to-peer approaches.

A number of prior P2P protocols for scalable on-demand streaming have used a cache-and-relay approach [3-6]. With these techniques, each peer receives content from one or more parents and stores it in a local cache, from which it can later be forwarded to clients that are at an earlier play point of the file. Some work of this type concerns the problem of determining the set of servers (or peers) that should serve each peer, and at what rate each server should operate [7, 8]. Related ideas, based on application-level multicast architectures, have been used in protocols for live streaming [9, 10].

* To appear in *Proc. IFIP/TC6 Networking '07*, Atlanta, GA, May 2007. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada.

The above approaches work best when peer connections are relatively stable. Motivated by highly dynamic environments where peer connections are heterogeneous with highly time-varying bandwidths and peers may join and/or leave the system frequently, recent work by Annapureddy *et al*. [11] has considered the use of BitTorrent-like protocols [12] for scalable on-demand streaming. (Other recent work has considered use of such protocols for live streaming [13-15].)

In BitTorrent-like protocols, a file is split into smaller pieces which can be downloaded (in parallel) from any other peer that has at least one piece that the peer does not have itself. In the approach proposed by Annapureddy *et al*. for on-demand streaming of stored media files, each file is split into sub-files, each encoded using distributed network coding [16]. Each sub-file is downloaded using a BitTorrent-like approach. By downloading sub-files sequentially, playback can begin after the first sub-file(s) have been retrieved, thus allowing a form of "streaming" delivery.

Note that use of large sub-files results in large startup delays, while using very small sub-files results in close to sequential piece retrieval, which can lead to poor performance as will be shown in Section 3.3. The best choice of sub-file sizes would be workload (and possibly also client) dependent, although the method requires these sizes to be statically determined. The authors do not elaborate on how the sizes can be chosen, or how startup delays can be dynamically determined.

Rather than statically splitting each file into sequentially retrieved sub-files or using a small window of pieces that may be exchanged (as in BitTorrent-like protocols that have been proposed for live streaming [13]), in this paper we propose an approach in which any of the pieces needed by a peer may be retrieved any time they are available. As in BitTorrent, selection of which piece to retrieve when a choice must be made is controlled by a piece selection policy. For the purpose of ensuring high piece diversity, which is an important objective in *download* systems (where the file is not considered usable until fully downloaded) [17], BitTorrent uses a *rarest-first* policy, giving strict preference to pieces that are the rarest among the set of pieces owned by all the peers from which it is downloading. On the other hand, in the context of *streaming* it is most natural to download pieces *in-order*. The piece selection policy proposed in this paper attempts to achieve a good compromise between the goals of high piece diversity, and in-order retrieval of pieces. We also address the problem of devising a simple on-line policy for deciding when playback can safely commence.

The remainder of the paper is organized as follows. Section 2 provides a brief overview of BitTorrent. Section 3 defines and evaluates candidate piece selection policies. Section 4 addresses the problem of dynamically determining the startup delay. Finally, conclusions are presented in Section 5.

2 Overview of BitTorrent

With BitTorrent files are split into *pieces*, which themselves are split into smaller sub-pieces. Multiple sub-pieces, potentially of the same piece, can be downloaded in parallel from different peers. A peer is said to *have* a piece whenever the entire piece is downloaded. A peer is considered *interested* in all peers that have at least one piece

that it currently does not have itself. BitTorrent distinguishes between peers that have the entire file (called *seeds*), and peers currently downloading the file (called *leechers*).

In addition to the *rarest-first* piece selection policy, BitTorrent uses a number of additional policies that determine which peers to upload to. While each peer establishes persistent connections with a large set of peers (e.g., 80 [17]), at each time instance, each peer only uploads to a limited number of peers. Only peers that are *unchoked* may be sent data. Generally, clients re-evaluate the set of unchoked peers relatively frequently (e.g., every 10 seconds, each time a peer becomes interest/uninterested, and/or each time a new connection is established/broken).

To discourage free-riding, BitTorrent uses a *tit-for-tat* policy in which leechers give upload preference to the leechers that provide the highest download rates to them. Without any measure of the upload rates from other peers, it has been found beneficial if seeds give preference to recently unchoked peers [17]. Periodically (typically every third time the set of unchoked peer is re-evaluated), each client uses an *optimistic unchoke* policy to probe for better pairings (or in the case of a seed, allow a new peer to download pieces).

3 Piece Selection

Section 3.1 defines candidate policies, Section 3.2 describes our simulation model, and Section 3.3 evaluates the performance of the piece selection policies defined in Section 3.1.

3.1 Candidate Policies

To allow playback to begin well before the entire media file is retrieved, pieces must be selected in a way that effectively mediates the conflict between the goals of high piece diversity and the *in-order* requirements of media file playback. Assuming that peer j is about to request a piece from peer i, we define two baseline policies:

- **Rarest:** Among the set of pieces that peer i has and j does not have, peer j requests the rarest piece among the set of all pieces held by peers that j is connected to. Ties are broken randomly.
- **In-order:** Among the set of pieces that peer i has, peer j requests the first piece that it does not have itself.

We propose using simple probabilistic policies. Perhaps the simplest such technique is to request an in-order piece with some probability and the rarest piece otherwise. Other techniques may use some probability distribution to bias towards earlier pieces. We have found that the Zipf distribution works well for this purpose. The specific probabilistic policies considered here are as follows:

- **Portion (p):** For each new piece request, client j uses the in-order policy with a probability p and the rarest policy with a probability $(1-p)$.
- **Zipf (θ):** For each new piece request, client j probabilistically selects a piece from the set of pieces that i has, but that j does not have. The probability of selecting each of these pieces is chosen to be proportional to $1/(k+1-k_0)^{\theta}$, where k is the index of the piece, and k_0 the index of its first missing piece.

Note that parameters p and θ can be tuned so that the policies are more or less aggressive with respect to their preference for earlier pieces. For the results presented here the parameters are fixed at the following values: p =50%, p = 90%, and θ = 1.25.

3.2 Simulation Model

A similar approach is used as in prior simulation studies of BitTorrent-like protocols [18, 16]; however, rather than restricting peers to a small number of connections, it is assumed that peers are connected to all other peers in the system. It is further assumed that pieces are split into sufficiently many sub-pieces that use of parallel download is always possible when multiple peers have a desired piece.

It is assumed that a peer i can at most have n_i concurrent upload connections and that no connections are choked in the middle of an upload. The set of peers that a peer i is uploading to may change when (i) it completes the upload of a piece, *or* (ii) some other peer becomes interested and peer i is not utilizing all its upload connections. The new set of upload connections consists of (i) any peer currently in the middle of an upload, *and* (ii) additional peers up to the maximum limit n_i. Additional peers are selected from the set of interested peers. To simulate optimistic unchoking, with a probability $1/n_i$ a random peer is selected, and with a probability of $(n_i–1)/n_i$ the peer which is uploading to peer i at the highest rate is selected. Random selection is used to break ties. This ensures that seeds only use random peer selection.

For simulating the rate at which pieces are exchanged, it is assumed that connection bottlenecks are located at the end points (i.e., either by the upload bandwidth U at the sender or by the download rate D at the receiver) *and* the network operates using max-min fair bandwidth sharing (using TCP, for example). Under these assumptions each flow operates at the highest possible rate that ensures that (i) no bottleneck operates above its capacity, and (ii) the rate of no flow can be increased without decreasing the rate of some other flow operating at the same or lower rate.

3.3 Performance Comparisons

Throughout this paper it is conservatively assumed that there is a single persistent seed and that all other peers leave the system as soon as they have received the entire file (i.e., act only as leechers). In a real system peers are likely to continue serving other peers as long as they are still playing out the media file, and some peers may choose to serve as seeds beyond that time. With a larger aggregate available download bandwidth and higher availability of pieces the benefits of more aggressive piece selection techniques are likely to be even greater than presented here.

Without loss of generality, downloading of a single file is considered, with size and play rate both equal to one using normalized units. With these normalized units, the volume of data transferred is measured in units of the file size and time is measured in units of the time it takes to play the file data. Hence, all rates are expressed relative to the play rate, and all startup delays are expressed relative to the playback time. The file is split into 512 pieces, and unless stated otherwise, peers are assumed to have three times higher download capacity than upload capacity. Each peer is assumed to upload to at most four peers simultaneously. Throughout this section, policies are evaluated with regard to the lowest possible startup delay.

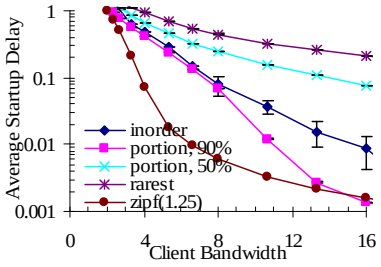


Fig. 1. Average achievable startup delay under constant rate Poisson arrival process. ($D/U = 3$, $\lambda = 64$, and $\varphi = 0$).

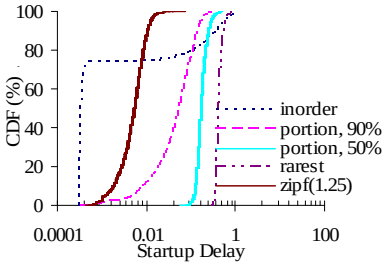


Fig. 2. Cumulative distribution function of the best achievable startup delay. ($U = 2$, $D = 6$, $\lambda = 64$, and $\varphi = 0$).

This section initially considers a simple scenario in which peers do not leave the system until having fully received the file, requesting peers arrive according to a Poisson process, and peers are homogenous (i.e., have the same upload and download bandwidth). Alternative scenarios and workload assumptions are subsequently considered.

To capture the steady state behavior of the system, the system is simulated for at least 4000 requests. Further, measurements are only done for requests which do not occur near the beginning or the end of each simulation. Typically, statistics for the first 1000 and the last 200 requests are not included in the measurements; however, to better capture the steady state behavior of the in-order policy a warmup period longer than 1000 requests is sometimes required.[1] Each data point represents the average of 10 simulations. Unless stated otherwise, this methodology is used throughout this paper. To illustrate the statistical accuracy of the results presented here, Fig. 1 includes confidence intervals capturing the true average with a confidence of 95%. Note that the confidence intervals are only visible for the in-order policy. Subsequent results have similar accuracy and confidence intervals are therefore omitted.

Fig. 1 shows the average startup delay as a function of the total client bandwidth ($U + D$). The peer arrival rate is $\lambda = 64$ and the seed has an upload bandwidth equal to that of the leechers. The most significant observation is that the Zipf(1.25) policy consistently outperforms the other candidate policies. In systems with an upload capacity at least twice the play rate (i.e., $U \geq 2$) peers are able to achieve startup delays two orders of magnitude smaller than the file playback time and much shorter than with the rarest-first policy.

Fig. 2 presents the cumulative distribution of achievable startup delays for this initial scenario. Note that Zipf(1.25) achieves low and relatively uniform startup delays. The high variability in startup delays using the in-order policy are due to groups of peers becoming synchronized, all requiring the same remaining pieces, which only the seed has. Being limited by the upload rate of the seed these peers will, at this point, see poor download rates. With many peers completing their downloads at roughly the same time, the system will become close to empty, before a new group

[1] The in-order policy was typically simulated using at least 20,000 requests.

of peers repeats this process. This service behavior causes the number of peers in the system using the in-order policy to follow a saw-tooth pattern. In contrast, the number of concurrent leechers, using the other policies, is relatively stable.

Fig. 3(a) shows that, as expected, in-order and portion(90%) do well in systems with low arrival rates; however, Zipf(1.25) outperforms these policies at moderate and higher arrival rates. The performance of Zipf(1.25) is relatively insensitive to the arrival rate. Note also that the decrease in average delay observed for high arrival rates for the in-order policy may be somewhat misleading as the achievable startup delay in this region is highly variable, as illustrated in Fig. 2.

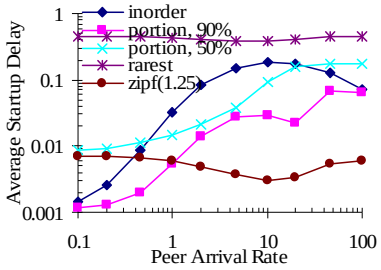
Fig. 3(b) shows that the results are relatively insensitive to the download/upload bandwidth ratio for ratios larger than 2. In this experiment the upload rate U is fixed at 2 and the download rate D varied. Note that typical Internet connections generally have ratios between 2 and 8 [19]. The increasing startup delays using the in-order policy are caused by a larger share of seed bandwidth being spent on serving recently arrived peers (which can be served by almost every other peer).

Fig. 3(c) illustrates that higher seed bandwidth allows the more aggressive (with respect to fetching pieces in order) portion and in-order policies to achieve better performance. For these results it is assumed that the maximum number of upload connections of the seed is proportional to its capacity.

In the second scenario that we consider, peers arrive according to a Poisson process, but each peer may leave the system prematurely. The rate at which each peer departs, prior to its complete reception of the file, is denoted by φ. Fig. 4 illustrates that the results are insensitive to the rate peers depart the system. This insensitivity to early departures is a characteristic of peers not relying on retrieving pieces from any particular peer and has been verified by reproducing very similar graphs to those presented in Fig. 1, 2, and 3.

In the third scenario that we consider, as motivated by measurement studies done on real file sharing torrents [20], peers are assumed to arrive at an exponentially decaying rate $\lambda(t) = \lambda_0 e^{-\gamma t}$, where λ_0 is the initial arrival rate at time zero and γ is a decay factor. By varying γ between 0 and ∞ both a pure Poisson arrival process and a flash crowd in which all peers arrive instantaneously (to an empty system) can be captured. Fig. 5 shows the impact of γ on performance. λ_0 is determined such that the expected number of arrivals within the first 2 time units is always 128. We note that with a decay factor $\gamma = 1$, 63.2% of all peer arrivals occur within the first time unit. With a decay factor $\gamma = 6.9$, the corresponding percentage is 99.9%. For these experiments no warmup period was used and simulations were run until the system emptied. Note that the performance of in-order and portion(90%) quickly becomes very poor, as the arrival pattern becomes burstier (i.e., for large γ and λ_0 values).

The fourth and final scenario that we consider assumes Poisson arrivals, as in the first scenario, but with two classes of peers: low bandwidth peers ($U_L = 0.4$, $D_L = 1.2$) and high bandwidth peers ($U_H = 2$, $D_H = 6$). Fig. 6 shows that the average startup delay for the high bandwidth peers significantly increases as the fraction of low bandwidth peers increases. The figure for low bandwidth peers looks very similar, with the exception that startup delays are higher (e.g., the minimum startup delay using Zipf(1.25) is roughly 0.08). Similar results have also been observed in a scenario where all peers are assumed to have a total bandwidth of 8 ($U = 2$ and $D = 6$),



(a) Impact of the peer arrival rate λ ($U = 2$, $D = 6$, $\gamma = 0$, $\varphi = 0$).

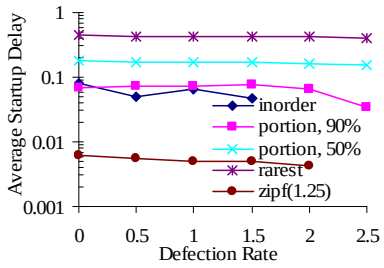
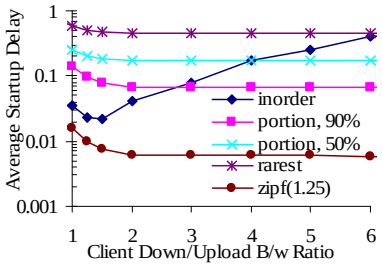


Fig. 4. Average achievable startup delay under a constant rate Poisson arrival process with early departures ($U = 2$, $D = 6$, $\lambda = 64$).



(b) Impact of the ratio between the client download and upload bandwidth D/U ($U = 2$, $\lambda = 64$, $\varphi = 0$).

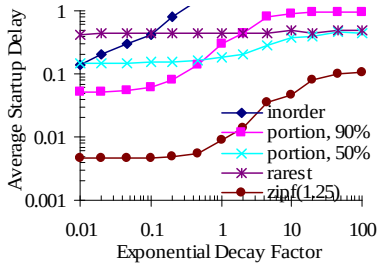
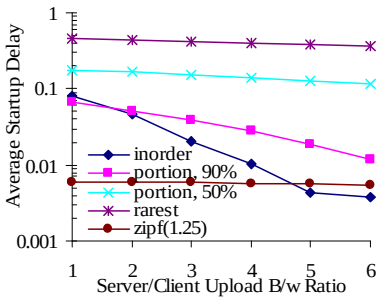


Fig. 5. Average achievable startup delay with an exponentially decaying arrival rate ($U = 2$, $D = 6$, $\lambda(t) = \lambda_0 e^{-\gamma t}$, $\lambda_0 = 128\gamma / (1 - e^{-2\gamma})$).



(c) Impact of the seed bandwidth ($U = 2$, $D = 6$, $\lambda = 64$, $\varphi = 0$).

Fig. 3. Impact of system parameters on the achievable startup delay.

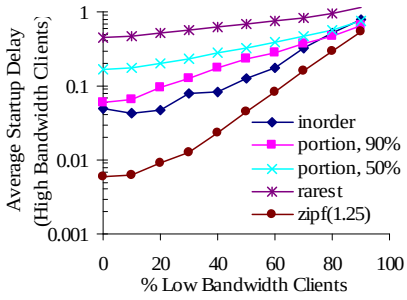


Fig. 6. Average achievable startup delay under a constant rate Poisson arrival process with both low and high bandwidth clients ($\lambda = 64$, $\varphi = 0$, $U_L = 0.4$, $D_L = 1.2$, $U_H = 2$, $D_H = 6$).

but a specified fraction of the peers make only 20% of their upload bandwidth available (i.e., $U_L = 0.4$ and $U_H = 2$).

4. Dynamically Determining Startup Delay

In highly unpredictable environments, with large and changing sets of peers, it is difficult to predict future system conditions. Therefore, one cannot expect any on-line strategy for selecting a startup delay to give close to minimal startup delays, without the potential of frequent playback interruption owing to pieces that have not been received by their playout point. To deal with such missing pieces, existing error concealment techniques can be applied by the media player, but at some cost in media playback quality. In this section we present a number of simple policies for determining when to start playback and evaluate how they perform when used in conjunction with the Zipf(1.25) piece selection policy.

4.1 Simple Policies

Possibly the simplest policy is to start playback once some minimum number of pieces have been received.

- **At-least (*b*):** Start playback when *b* pieces have been received, and one of those pieces is the first piece of the file.

Somewhat more complex policies may attempt to measure the rate at which in-order pieces are retrieved. We define an "in-order buffer" that contains all pieces up to the first missing piece, and denote by d_{seq}, the rate at which the occupancy of this buffer increases. Note that d_{seq} will initially be smaller than the download rate (as some pieces are retrieved out-of-order), but can exceed the download rate as holes are filled. The rate d_{seq} can be expected to increase over time, as holes are filled more and more frequently. Therefore, it may be safe to start playback once the estimated current value of d_{seq} allows the in-order buffer to be filled within the time it takes to play the entire file (if that rate was to be maintained). With k pieces in the in-order buffer d_{seq} must therefore be at least $(K–k) / K$ times as large as the play rate r. Using this "rate condition" two rate-based policies can be defined.

- **LTA (*b*):** The current value of d_{seq} is conservatively estimated by $(Lk/K)/T$, where T is the time since the peer arrived to the system. With the LTA(b) policy a client starts playback when at least b pieces have been retrieved *and* $(Lk/K)/T \geq r(K–k)/K$. (See Fig. 7.)
- **EWMA (*b*, α):** The current value of d_{seq} is estimated by $(L/K)/\tau_{seq}$, where τ_{seq} denotes an exponentially weighted moving average of the time between additions of a piece to the in-order buffer. With the EWMA(b, α) policy a client starts playback when at least b pieces have been retrieved *and* $(L/K)/\tau_{seq} \geq r(K–k)/K$.

τ_{seq} is initialized at the time the first piece of the file is retrieved to the time since the peer's arrival to the system. When multiple pieces are inserted into the in-order buffer at once, they are considered to have been added at times equally spaced over the time period since the previous addition. For example, if the 10^{th}, 11^{th} and 12^{th} pieces of the file are added to the in-order buffer together (implying that at the time the 10^{th} pieces was received pieces 11 and 12 had previously been received), then τ_{seq} is updated three times, with each inter-arrival time being one third of the time since the 9^{th} piece was added to the in-order buffer.

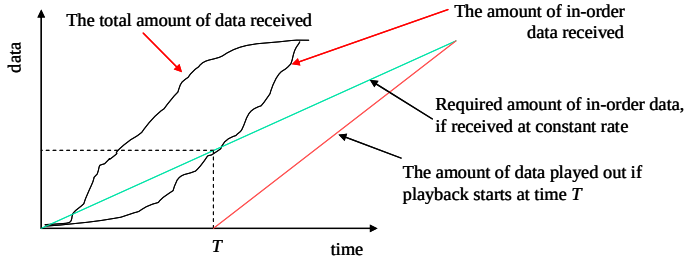


Fig. 7. Startup condition of the LTA policy, using the amount of in-order data received by each time *T*.

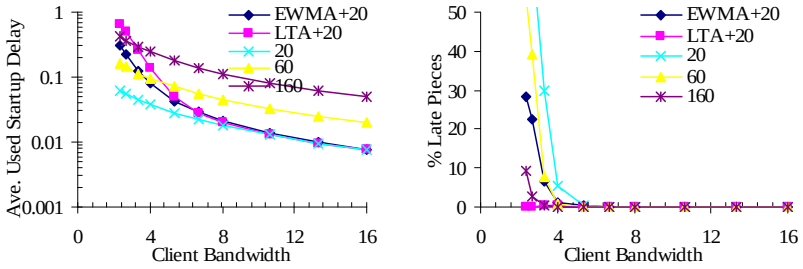
4.2 Performance Comparisons

Making the same workload assumptions as in Section 3.3, the above startup policies are evaluated together with the Zipf(1.25) piece selection policy. While policies may be tuned for the conditions under which they are expected to operate, for highly dynamic environments, it is important for policies to adapt as the network condition changes.

To evaluate the above policies over a wide range of network conditions the four scenarios from Section 3.3 are used. Most comparisons are of: (i) at-least(20), (ii) at-least(60), (iii) at-least(160), (iv) LTA(20), and (v) EWMA(20, 0.1). Fig. 8 through 11 present the average startup delay and the percentage of pieces that are not retrieved in time for playback. Again, note that such pieces could be handled by the media player using various existing techniques, although with some degradation in quality.

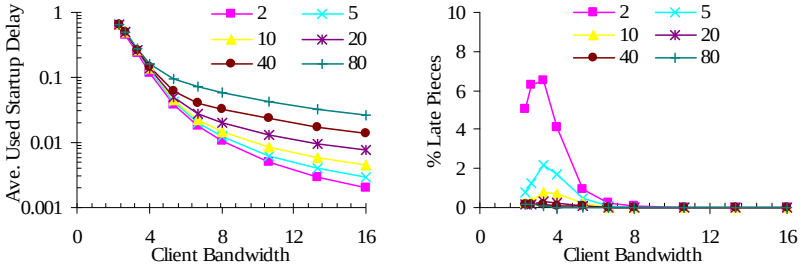
Fig. 8 and 9 present results for the first scenario. Fig. 8 shows that the d_{seq} estimate, used by LTA(20) and EWMA(20,0.1), allows these policies to adjust their startup delay based on the current network conditions. These policies increase their startup delay enough so as to ensure a small percentage of late pieces, for reduced client bandwidths. This is in contrast to the at-least policy which always requires the same number of in-order pieces to be received before starting playback (independent of current conditions). Fig. 9 shows the impact of using different *b*-values with the LTA policy. As can be observed, most benefits can be achieved using *b* equal to 10 or 20. These values allow for relatively small startup delays to be achieved without any significant increase in the percentage of late pieces. While omitted, results for the second scenario suggest that the results are relatively insensitive to departure rates.

Fig. 10 presents the results for the third scenario. Here, the exponential decay factor (measuring the burstiness with which peers arrive) is varied four orders of magnitude. Fig. 11 presents the results for scenario four, in which arriving peers belong to one of two classes, high and low bandwidth clients. For this scenario, the portion of low bandwidth peers is varied such that the network conditions change from good (where most peers are high bandwidth clients) to poor (where the majority of peers are low bandwidth clients). As in previous scenarios, we note that both LTA(20) and EWMA(20,0.1) adjust well to the changing network conditions, while the at-least policy is non-responsive and do not adjust its startup delays. This is best illustrated by the relatively straight lines and/or high loss rates observed by this policy.



(a) Average startup delay. (b) Percentage of late pieces.

Fig. 8. Performance with constant rate Poisson arrival process (D/U= 3, λ= 64, and φ= 0).



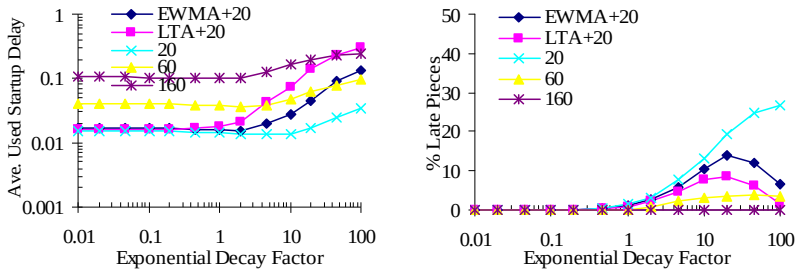
(a) Average startup delay. (b) Percentage of late pieces.

Fig. 9. Impact of parameter b in the LTA policy (D/U = 3, λ = 64, and φ = 0).

Designed for highly dynamic environments we find the LTA(b) policy promising. It is relatively simple, uses a single parameter, and is somewhat more conservative than EWMA(b,α), which may potentially give too much weight to temporary changes in the rate at which the in-order buffer is being filled.

5 Conclusion

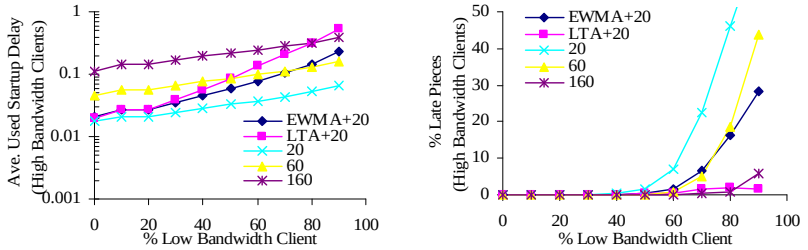
This paper considers adaptations of the BitTorrent-like approach to peer-assisted download that provide a form of streaming delivery, allowing playback to begin well before the entire file is received. A simple probabilistic piece selection policy is shown to achieve an effective compromise between the goal of high piece diversity, and in-order piece retrieval. Whereas one cannot expect any on-line strategy for selecting startup delays to give close to minimal startup delays, we find that a simple rule based on the average rate at which in-order pieces are retrieved to give promising results.



(a) Average startup delay.

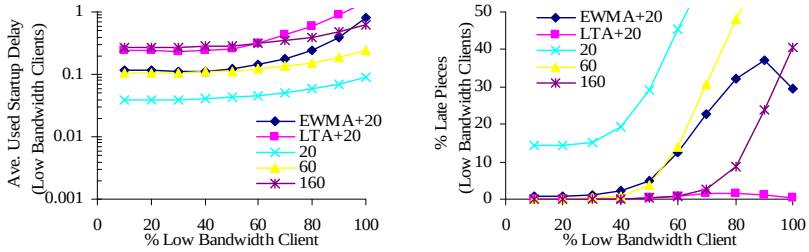
(b) Percentage of late pieces.

Fig. 10. Performance with an exponentially decaying arrival rate (U=2, D=6, $\lambda(t) = \lambda_0 e^{-\gamma t}$, $\lambda_0 = 128\gamma / (1 - e^{-2\gamma})$).



(a) Average startup delay for high bandwidth clients.

(b) Percentage of late pieces for high bandwidth clients.



(c) Average startup delay for low bandwidth clients.

(d) Percentage of late pieces for low bandwidth clients.

Fig. 11. Performance with heterogeneous clients (λ = 64, φ = 0, U_L = 0.4, D_L = 1.2, U_H = 2, D_H = 6).

References

1. Hua, K. A., Cai, Y., Sheu, S.: Patching: A Multicast Technique for True Video-on-Demand Services. In: Proc. ACM MULIMEDIA '98, Bristol, U.K., Sept. 1998, pp. 191--200.

2. Eager, D. L., Vernon, M. K., Zahorjan, J.: Optimal and Efficient Merging Schedules for Video-on-Demand Servers. In: Proc. ACM MULTIMEDIA '99, Orlando, FL, Nov. 1999, pp. 199--202.
3. Cui, Y., Li, B., Nahrstedt, K.: oStream: Asynchronous Streaming Multicast in Application-layer Overlay Networks, IEEE Journal on Selected Areas in Communications (Special Issue on Recent Advances in Service Overlays) **22** (1) (2004), pp. 91--106.
4. Bestavros, A., Jin, S.: OSMOSIS: Scalable Delivery of Real-time Streaming Media in Ad-hoc Overlay Networks. In: Proc. ICDCS Workshops '03, Providence, RI, May 2003, pp. 214--219.
5. Sharma, A., Bestavros, A., Matta, I.: dPAM: A Distributed Prefetching Protocol for Scalable Asynchronous Multicast in P2P Systems. In: Proc IEEE INFOCOM '05, Miami, FL, Mar. 2005, pp. 1139--1150.
6. Luo, J-G., Tang, Y., Yang, S-Q.: Chasing: An Efficient Streaming Mechanism for Scalable and Resilient Video-on-Demand Service over Peer-to-Peer Networks. In: Proc. IFIP NETWORKING '06, Coimbra, Portugal, May 2006, pp. 642--653.
7. Rejaie, R., Ortega, A.: PALS: Peer-to-Peer Adaptive Layered Streaming, in: Proc. NOSSDAV '03, Monterey, CA, June 2003, pp. 153--161.
8. Hefeeda, M., Habib, A., Botev, B., Xu, D. Bhargava, B.: PROMISE: Peer-to-Peer Media Streaming using CollectCast. In: Proc. ACM MULTIMEDIA '03, Berkeley, CA, Nov. 2003, pp. 45--54.
9. Castro, M., Druschel, P., Rowstron, A., Kermarrec, A-M., Singh, A., Nandi, A.: SplitStream: High-Bandwidth Multicast in Cooperative Environments. In: Proc. ACM SOSP '03, Bolton Landing, NY, Oct. 2003, pp. 298--313.
10. Kozic, D., Rodriguez, A., Albrecht, J., Vahdat, A.: Bullet: High Bandwidth Data Dissemination using an Overlay Mesh. In: Proc. ACM SOSP '03, Bolton Landing, NY, Oct. 2003, pp. 282--297.
11. Annapureddy, S., Gkantsidis, C., Rodriguez, P. R.: Providing Video-on-Demand using Peer-to-Peer Networks, Proc. Workshop on Internet Protocol TV (IPTV) '06, Edinburgh, Scotland, May 2006.
12. Cohen, B.: Incentives Build Robustness in BitTorrent. In: Proc. Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, June 2003.
13. Zhang, X., Liu, J., Li, B., Yum, T-S. P.: CoolStreaming/DONet: A Datadriven Overlay Network for Peer-to-Peer Live Media Streaming. In: Proc. IEEE INFOCOM '05, Miami, FL, March 2005, pp. 2102--2111.
14. Zhang, M., Zhao, L., Tang, Y., Luo, J-G., Yang, S-Q.: Large-scale Live Media Streaming over Peer-to-Peer Networks through Global Internet. In: Proc. Workshop on Advances in Peer-to-Peer Multimedia Streaming '05, Singapore, Nov. 2005, pp. 21--28.
15. Liao, X., Jin, H., Liu, Y., Ni, L. M., Deng, D.: AnySee: Peer-to-Peer Live Streaming. In: Proc. IEEE INFOCOM '06, Barcelona, Spain, Apr. 2006.
16. Gkantsidis, C., Rodriguez, P. R.: Network Coding for Large Scale Content Distribution. In: Proc. IEEE INFOCOM '05, Miami, FL, Mar. 2005, pp. 2235--2245.
17. Legout, A., Urvoy-Keller, G., Michiardi, P.: Rarest First and Choke Algorithms are Enough. In: Proc. ACM IMC '06, Rio de Janeiro, Brazil, Oct. 2006.
18. Bharambe, A. R., Herley, C., Padmanabhan, V. N.: Analyzing and Improving a BitTorrent Network's Performance Mechanisms. In: Proc. IEEE INFOCOM '06, Barcelona, Spain, Apr. 2006.
19. Saroiu, S., Gummadi, K. P., Gribble, S. D.: A Measurement Study of Peer-to-Peer File Sharing Systems. In: Proc. IS&T/SPIE MMCN '02, San Jose, CA, Jan. 2002, pp. 156--170.
20. Guo, L., Chen, S., Xiao, Z., Tan, E., Ding, X., Zhang, X.: Measurement, Analysis, and Modeling of BitTorrent-like Systems. In: Proc. ACM IMC '05, Berkley, CA, Oct. 2005, pp. 35--48.