Realistic Media Streaming over BitTorrent

Charilaos STAIS, George XYLOMENOS
Mobile Multimedia Laboratory, Department of Informatics
Athens University of Economics and Business, Athens, Greece
Email: {stais, xgeorge}@aueb.gr

***Abstract*—While BitTorrent was originally conceived as a Peer-to-Peer file exchange protocol, it has proved extremely successful for asynchronous content distribution, allowing content sources to support huge numbers of users with a modest amount of bandwidth. This has prompted many researchers to study the possibility of using BitTorrent to support real-time media streaming. In this paper we present a comparison of three proposed adaptations to BitTorrent for media streaming, using our detailed packet-level BitTorrent simulator. Unlike previous evaluations which assumed that the streaming media player would drop data that did not arrive on time, in this paper we use a more realistic model where the player stalls when data are not available, thus placing emphasis on delays rather than losses. Our experiments indicate that under this, more realistic, evaluation model, user level performance can be quite reasonable.**

***Index Terms*—Peer-to-peer, BitTorrent, Streaming, Simulation, Buffer**

I. Introduction

The original communication model of TCP/IP networks was packet exchange between pairs of communicating end hosts, esssentially a packet-switched version of telephony. The commercialization of the Internet however and the success of the World Wide Web has caused Internet use to shift towards information-centric services and applications, giving rise to content delivery networks, cloud computing services and peer-to-peer (P2P) [1] file sharing applications. These services and applications have to be implemented as overlays on top of the information-agnostic Internet [2], as they focus on the information exchanged rather than on the end hosts providing it.

The success of P2P protocols and applications made it apparent that they could be used for much more than simple file exchange. For example, BitTorrent is widely used to distribute large pieces of content, such as entire Linux distributions, as it allows the content provider to serve huge amounts of users without investing in corresponding amounts of transmission capacity. The success of BitTorrent in this role led many researchers to propose modifications to it to also handle media streaming [3], [4], [5]. The simulation based performance evaluations of these proposals however are lacking: simulation setups are not fully documented, simulators do not take into account BitTorrent and TCP/IP protocol details and each evaluation uses different metrics.

In previous work [6], we addressed all these problems by presenting a performance evaluation of three proposed multimedia streaming extensions to BitTorrent, using the same assumptions and metrics for each scheme and a detailed packet-level BitTorrent simulator. That study assumed a player model where data that was not available on time was dropped, therefore the main metric of interest was the loss rate of each approach, similar to [3], [4]. Most realistic media players however, including the ubiquitous YouTube player, do not work like that: when data is not available on time, they *stall* until it arrives, similar to [5]. In this paper we compare the three proposed approaches using this type of media player, focusing on the number and duration of stall periods, which are a much better reflection of the quality of experience perceived by actual users.

The rest of the paper is organized as follows. In Section 2 we briefly present the three streaming extensions to BitTorrent examined in our work. Section 3 details our experimental setup, including the simulator used, while in Section 4 we discuss the results of our experiments. We conclude in Section 5.

II. Streaming Extensions to BitTorrent

A. General

Due to space limitations, we will summarize here the main BitTorrent concepts relevant to the paper; see [7] for a detailed explanation of BitTorrent protocols and algorithms. BitTorrent splits the content to be exchanged into fixed-size *pieces*, which can be individually requested; each piece is split into fixed-size *blocks* which can be downloaded in a pipelined mode. A server called the *tracker* facilitates the BitTorrent exchange by responding to queries from peers with the contact details of other peers; the tracker gathers this information from the queries themselves.

All the peers participating in a file exchange are collectively called a *swarm*. Each peer establishes contact with some of the swarm members returned by the tracker and monitors which pieces of the file are available at each peer. Peers generally exchange pieces in a tit-for-tat fashion, i.e. a peer only sends pieces to other peers that have also sent pieces to it [8]. Occassionaly, a peer will send some pieces to someone that has not yet reciprocated; this is how new peers manage to get their first pieces in order to participate in the tit-for-tat exchange. Out of the pieces that each peer is missing, it tries to download the rarest in the swarm for two reasons: first, rare pieces can disappear if a few peers leave the swarm; second, rare pieces are in demand by other peers, therefore they facilitate the tit-for-tat exchange.

Streaming applications generally download data in sequence, so as to be able to start reproducing the initial data

before the entire content has been downloaded, something against the *rarest-first* piece selection scheme of BitTorrent. As a result, all the proposed streaming extensions to BitTorrent start by modifying the piece selection mechanism, aiming to balance the needs of streaming, where orderly data arrival is required to maintain playback quality, and the needs of the rarest-first selection policy, which makes tit-for-tat work in BitTorrent. We describe below three such approaches.

B. Fixed-Size Window

The *Fixed-Size Window* (FSW) approach [3] modifies the BitTorrent piece selection strategy by placing a fixed-size sliding window over the pieces and only allowing pieces within that window to be selected for downloading. This window starts from the first piece that has not been downloaded yet and includes k consecutive pieces, where k is a configuration parameter. Essentially, rarest-first is limited to operate within the window, thus allowing the pieces which will be played back soon to be downloaded, without "wasting" bandwidth to download other pieces. The window slides to the right when its first piece has been downloaded, to allow more pieces to be selected for download.

C. High-Priority Set

The FSW approach does not exploit the available piece download opportunities that well. On the one hand, if most pieces in the window have completed downloading but the first one has not, the window cannot slide, hence the choice of pieces becomes very limited. On the other hand, by limiting the peer within the window it misses the opportunity to download rare pieces outside the window that may become useful later on in the tit-for-tat exchange. For this reason, in the *High-Priority Set* (HPS) or BiToS approach [4], a fixed-size set (the HPS) holds the next pieces in sequence that have not been already downloaded. While in the FSW approach the window of size k covers exactly k consecutive pieces in the sequence space, some of which may have already been downloaded, in the HPS approach the set of size k covers at least k pieces in the piece sequence space, none of which has been downloaded.

The other departure from FSW is that in this approach pieces outside the HPS can also be requested. With probability p a peer will select a piece from the HPS and with probability $1-p$ the peer will select a piece beyond the HPS, in both cases using the rarest-first policy within each set; p is a configuration parameter. Whenever a piece completes downloading, it is removed from the HPS and the next in sequence piece that has not been downloaded yet is added to the HPS.

D. Stretching Window

The HPS approach has the drawback that as pieces are downloaded, the size of the HPS remains constant, therefore the first pieces in the HPS, which are probably closer to their playback time, do not increase their priority. In the *Stretching Window* (SW) approach this problem is mitigated by combining elements of the FSW and HPS approaches [5]. The SW behaves like the HPS in that it contains (up to) k pieces that have not been downloaded, but the distance between the first and last piece in the SW in terms of the piece sequence space is bounded by a limit $l > k$; hence, the SW may contain up to k pieces, provided these pieces do not cover more than l consecutive slots in the piece sequence. In addition, pieces are only requested from within the SW.

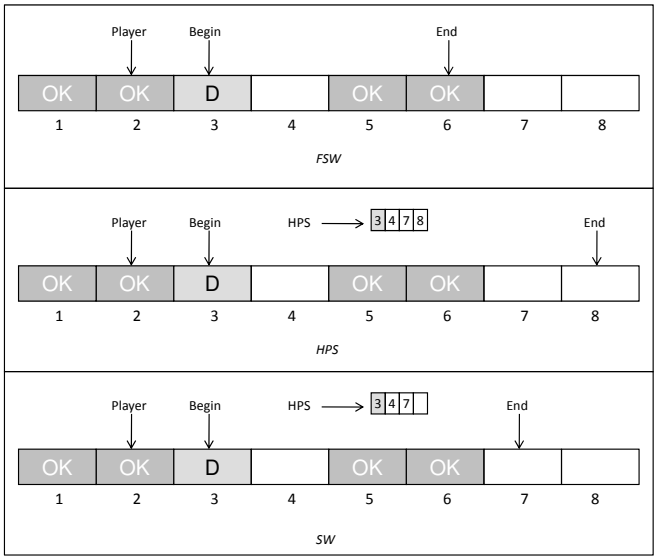


Fig. 1. Protocol extension functionality.

Figure 1 explains the differences between the three approaches. In all cases, pieces marked OK have already been downloaded, pieces marked D are currently being downloaded and unmarked pieces have not started downloading yet. Arrows indicate the current playback position, as well as the window limits (for FSW) or HPS limits (for HPS and SW). In the FSW approach the window starts from the first in piece being downloaded and covers $k = 4$ consecutive pieces, regardless of how many of these have been downloaded. In the HPS approach the HPS starts from the same point, but covers k not downloaded pieces, whether they have started downloading or not; the pieces in the HPS are also shown above the window. In addition, pieces may be requested outside the HPS. In the SW approach, the SW can also grow like the HPS to up to k not downloaded pieces, but it may not cover more than $l = 5$ slots in the piece sequence space, hence it may be smaller than with HPS, as shown in the figure. In addition, pieces are only requested from within the SW, unlike in the HPS approach.

III. Experimental Setup

A. BitTorrent Simulator

Our simulation-based comparison uses our own detailed OMNeT++ Simulator for BitTorrent, which is publicly available [7], extended with the necessary functionality for the three streaming extensions discussed above. Our simulator models not only the detailed protocol messages exchanged by

BitTorrent peers and the tracker, it also models all the TCP/IP messages exchanged, using the INET framework; for larger scale simulations, a simplified communication model can also be used [7], without any modifications to the BitTorrent code. We used transit-stub topologies generated by the GT-ITM module [9], including both core and access routers. Each scenario was executed 10 times with different random seeds, using the tools from the OverSim framework [10] to randomly place peers in the network. The overall setup is identical to that of [6] unless explicitly indicated otherwise.

B. Experiments

Our experimental scenario involves one initial seeder and 120 peers which join the swarm at random times, starting from scratch, thus modeling an incremental join process rather than a flash crowd. The topology we used consists of 4 *Autonomous Systems* (AS) and 192 access routers in total. Peer access links have asymmetric uplink and downlink bandwidths: 20% have 1/4 Mbps capacity (uplink/downlink), 40% have 1/8 Mbps, 25% have 2/12 Mbps and 15% have 2/24 Mbps.

The streaming application modeled is a video player attempting to playback a 256 Kbps video stream. We assumed that the video was encoded in an MPEG like manner, where each *Group of Pictures* (GOP) was mapped to exactly one BitTorrent piece. We set the GOP/piece size to 112 KB (192 KB in [6]), which translates to 3.5 seconds of video per piece, with 8 KB blocks (16 KB in [6]). The entire video size was 200 MB, which corresponds to around 106 minutes of playtime or 1828 pieces (1067 pieces in [6]). A piece that has not been downloaded on time, will cause the player to stall, as it cannot decode the next frames; the player will have to wait until the piece has been downloaded in its entirety, so as to be able to resume decoding. This follows a YouTube like model, unlike in our previous work where the player continued playback using blank frames, essentially dropping pieces that had not arrived on time [6].

We kept all the default BitTorrent settings unchanged e.g. optimistic unchoke interval and number of connections per peer, as given in [7]. To make our model relevant to live streaming, we assumed that a peer will remain in the swarm and seed other peers until the video playback reaches its end with probability 50% (100% in [6]), i.e. that roughly half of the peers only leave the swarm at playback completion. The initial seeder on the other hand remains permanently in the swarm, thus there is always at least one source for every piece. The seed then represents a video source that exploits BitTorrent to offload some of the traffic to the downloading peers.

Each peer starts by prefetching a few initial pieces (either 1 or 5), as in most media players, in order to provide a satisfactory buffer to the player before starting playback. During the prefetch phase, we use the rarest-first policy only for the pieces in the prefetch buffer, until they have all completed downloading, therefore all approaches operate identically in this phase. We set the base window size k to 2% or 8% of the total number of pieces for each algorithm, which in our case translates to 36 or 147 pieces, respectively (21 or 85 in [6]). The probability p of selecting pieces outside of the HPS was set to 80%. The bound l for SW was set to 50 pieces for $k = 36$ and to 200 pieces for $k = 147$, respectively (30 and 100 in [6]), i.e. the window can grow more than in SW, but not without limit as in HPS. Table I summarizes these parameters.

TABLE I
SIMULATION PARAMETERS

Parameter	Value
Video size (in MB)	200
Video bit rate (in Kbps)	256
Piece size (in KB)	112
Block size (in KB)	8
Number of pieces for prefetch buffering	1 or 5
Window size k (% of total num of pieces)	2% or 8%
Probability p (only for HPS)	80%
Bound in pieces (only for SW)	50 or 200
Probability to keep seeding	50%

IV. EXPERIMENTAL RESULTS

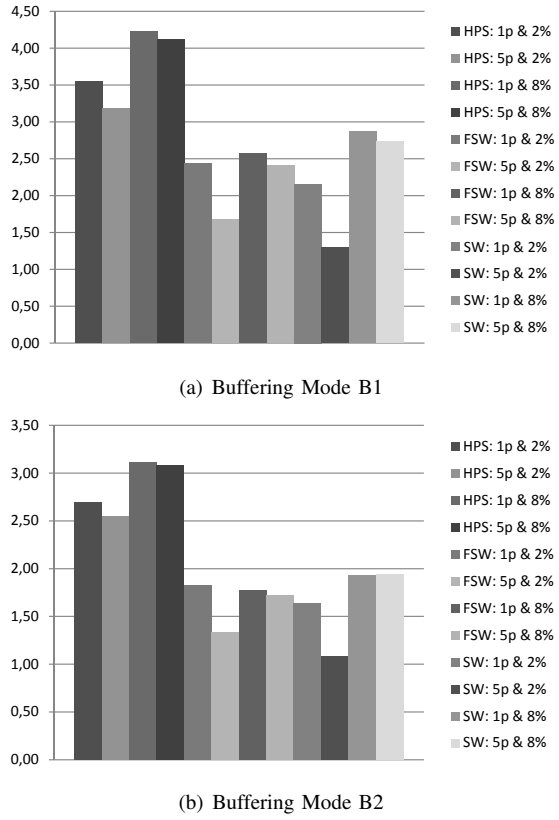
In this section we present and evaluate the simulation results. The main metrics for our evaluation are the number of *stall periods* and their *average duration* in each download, which directly reflect the experience of the user when viewing a streamed video. We tested two different buffering strategies after a stall occurs with each protocol. In mode B1 the player stalls only until the next piece has completed downloading, while in mode B2 the player stalls until the next three pieces have been downloaded; the former strategy tries to minimize the average stall duration while the latter tries to minimize the number of stall periods by prefetching more pieces.

A. Number of Stall Periods

Figure 2 shows the average number of stall periods during playback, that is, how many times the user will experience a stall during playback. As expected, the number of stall periods in Mode B1 is significantly higher (around 30% on average) than that in Mode B2 for all protocol extensions, since in Mode B1 the player only buffers the next piece before continuing. However, the absolute number of stalls is quite small in all cases (from 1.5 to 4) despite the large playback duration (more than 100 minutes), which is good news for user experience. For each individual protocol and for both buffering modes, the best performance is achieved by having a smaller window (2% rather than 8%) and prefetching more pieces (5 rather than 2). Among the different protocols, HPS has the worst behavior, as it has the largest effective window and it also downloads pieces outside the window, thus paying less attention to the pieces that are nearing their playback time. FSW and SW have very similar performance; FSW is slightly better with a larger window, while SW is slightly better with a smaller window.

B. Average Stall Duration

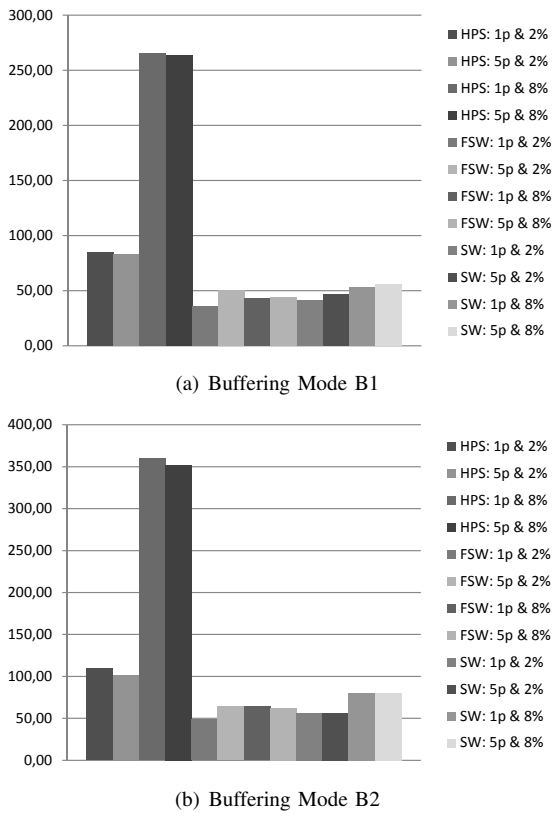
Figure 3 shows the average stall duration during playback, that is, how long the user will have to wait in each stall period during playback. As expected, the average stall duration in Mode B2 is significantly higher (again around 30% on average, exactly counterbalancing the stall period metric) than that in Mode B1, since in Mode B2 the player waits until the next



(a) Buffering Mode B1

(b) Buffering Mode B2

Fig. 2. Number of Stall Periods per download



(a) Buffering Mode B1

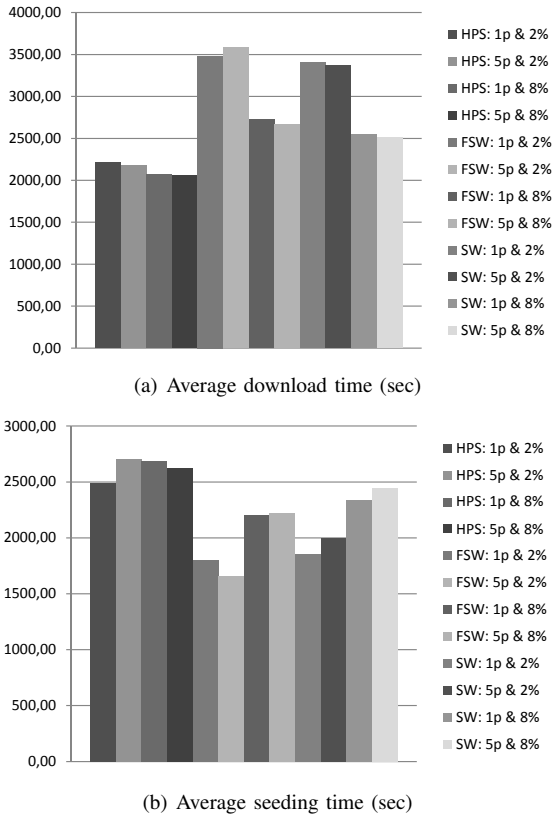
(b) Buffering Mode B2

Fig. 3. Average Stall Duration per download (sec)

three pieces are buffered. The absolute average stall duration ranges from noticeable (from 40 to 90 sec for FSW and SW) to quite long (from 80 to 350 sec for HPS). Again, a smaller window (2% rather than 8%) works best for all approaches, but the effect of the prefetch buffer is insignificant. The FSW and SW protocols exhibit very similar performance as in the previous metric, with FSW slightly better with a larger window and SW slightly better with a smaller window, but this time the HPS scheme is clearly unacceptable with the larger window size.

C. Download Time

Figure 4 shows the average download time and the average seeding time per download for buffering Mode B1; the results are nearly identical in Mode B2, as the buffering mode only changes the behavior of the player and not the protocol. As also found in [6], HPS offers the lowest download times due to the fact that its larger effective window and occasional downloads outside the window provide it with more opportunities for downloading rare pieces. In addition, the faster a peer completes the download, the more time it will remain as a seeder if it chooses to do so, since the playback duration is fixed, therefore HPS also exhibits higher seeding ties. FSW and SW exhibit very similar performance, as in the previous metrics. However, since all protocols complete the download well before the end of the playback duration (which is 6400 sec), emphasis should be placed on the user visible performance metrics discussed above, where HPS has clear problems, rather than on the download time.



(a) Average download time (sec)

(b) Average seeding time (sec)

Fig. 4. Average download and seeding time per download (sec)

D. Overall Evaluation

Unlike our previous work that led to mixed results [6], in the sense that each protocol variant provided lower loss rates for some scenarios, the more realistic stall-based evaluation model adopted in this paper clearly shows that HPS does not work well: compared to FSW and SW, it suffers from both more stalls and higher delays per stall, thus leading to visibly worse user experience, which is especially noticeable in the average stall delay metric. On the other hand, FSW and SW perform similarly, with SW better with smaller windows and FSW better with larger windows, indicating (as in [6]) that the extra complexity of SW is not worthwhile, provided the window is large enough. With FSW, the simplest but best performing approach, the worst case behavior is less than 2.5 stalls per download each lasting no more than 60 sec, which is reasonable for such large downloads. The two buffering modes tested led to the expected results, i.e. with less buffering per stall (mode B1) we have shorter stall periods, while with more buffering per stall (mode B2) we have fewer stall periods.

V. Conclusion and Future Work

In this paper we extended our previous work [6] by introducing a more realistic model for the media player, which stalls when data are not available, as in YouTube and other streaming services. We explored, through simulations using detailed TCP/IP message exchanges, the factors that impact protocol performance and behavior, using the same setup for each protocol extension, focusing on the number of *stall periods* and their *average duration* when playing back a large video file. Our results indicate that the simplest approach works best and that the overall user experience that it offers is quite reasonable, i.e. 1 to 2.5 stalls of 40 to 60 seconds each, for a video lasting 106 minutes. On the other hand, a more complex approach, even though it offers markedly shorter download times, leads to much worse user visible performance.

Future work includes the study of modifications to the protocols so that they can focus on the next pieces when the player is in stall mode, so as to avoid lengthy stall periods, and an investigation of how the performance of the first few peers, which rely only on the original seeder to receive pieces, affects the results of the entire experiment.

References

[1] T. Karagiannis, P. Rodriguez, and K. Papagiannaki, "Should Internet service providers fear peer-assisted content distribution?," in *Proc. of the Internet Measurement Conference*, pp. 63–76, 2005.

[2] D. Clark, B. Lehr, S. Bauer, P. Faratin, R. Sami, and J. Wroclawski, "Overlay networks and the future of the Internet," *Communication & Strategies*, vol. 63, no. 3, pp. 1–21, 2006.

[3] P. Shah and J. Paris, "Peer-to-peer multimedia streaming using BitTorrent," in *Proc. of the IEEE International Performance, Computing, and Communications Conference*, pp. 340–347, 2007.

[4] A. Vlavianos, M. Iliofotou, and M. Faloutsos, "BiToS: Enhancing BitTorrent for supporting streaming applications," in *Proc. of the IEEE INFOCOM*, 2006.

[5] P. Savolainen, N. Raatikainen, and S. Tarkoma, "Windowing BitTorrent for video-on-demand: Not all is lost with tit-for-tat," in *Proc. of the IEEE GLOBECOM*, 2008.

[6] C. Stais, G. Xylomenos, and A. Archontovasilis, "A comparison of streaming extensions to BitTorrent," in *Proc. of the IEEE ISCC*, pp. 1068–1073, 2011.

[7] K. Katsaros, V. Kemerlis, C. Stais, and G. Xylomenos, "A BitTorrent module for the OMNeT++ simulator," in *Proc. of the IEEE MASCOTS*, 2009.

[8] B. Cohen, "Incentives build robustness in bittorrent," in *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, 2003.

[9] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *Proc. of the IEEE INFOCOM*, vol. 2, pp. 594–602, 1996.

[10] I. Baumgart, B. Heep, and S. Krause, "OverSim: A flexible overlay network simulation framework," in *Proc. of the IEEE Global Internet Symposium*, 2007.