A comparison of streaming extensions to BitTorrent

Charilaos Stais, George Xylomenos, Angelos Archodovassilis
Mobile Multimedia Laboratory, Department of Informatics
Athens University of Economics and Business, Athens, Greece
Email: stais@aueb.gr, xgeorge@aueb.gr, archodovassilis@yahoo.com

***Abstract*—Peer-to-peer (P2P) protocols have been widely accepted by users and operators alike as efficient mechanisms for non real-time content distribution. It is therefore reasonable to extend these protocols to also handle more demanding applications, such as multimedia streaming. Many researchers have proposed modifications to the well known BitTorrent protocol in order to adapt it to the needs of such applications. In this paper we present findings from our experiments with three proposals for the integration of multimedia streaming into BitTorrent: the fixed-size window approach, the high-priority set approach and the stretching window approach. We evaluate these proposals under identical circumstances using our detailed packet-level BitTorrent simulator, showing that while all approaches are reasonably capable of supporting multimedia streaming, their different design choices have a pronounced effect on their streaming performance.**

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

I. INTRODUCTION

The explosive growth of the Internet is largely due to the ubiquitous adoption of the TCP/IP protocol stack. TCP/IP-based networks were designed to simply forward traffic between pairs of communicating end hosts, following the communication model of the telephone network. However, communication patterns have evolved, and the use of the Internet has shifted towards information-centric services and applications, such as content delivery networks (CDNs), cloud computing services and peer-to-peer (P2P) [1] file sharing applications like BitTorrent. These services and applications concentrate on the information itself, rather than on the end hosts providing or consuming it. Hence, they are implemented as overlay solutions on top of an information-agnostic network substrate [2].

As P2P protocols and applications became more common, many researchers realized their potential for multimedia streaming purposes. Even though BitTorrent was designed for non real-time content exchange, adapting it for multimedia streaming seems promising, due to its efficient mechanisms and its wide user acceptance. As a result, many proposals for streaming oriented modifications to BitTorrent have surfaced, but their simulation based performance evaluations are lacking: simulations setups are not documented in detail, simulators do not take into account the entire protocol stack overhead or the possible retransmissions in the network, and each evaluation uses different metrics.

In this paper, we aim to address all these problems by presenting a thorough comparison between three proposed multimedia streaming extensions to BitTorrent, using the same assumptions and metrics for each protocol and a detailed packet-level BitTorrent simulator, enhanced with streaming oriented modifications. Our simulator is available on an open source basis to the community, therefore allowing others to duplicate our results and further experiment with the proposed multimedia streaming extensions.

The rest of the paper is organized as follows. In Section II we provide a brief description of the BitTorrent protocols and explain why they are unsuitable for streaming "out of the box". In Section III we present three streaming extensions to BitTorrent. Section IV details our experimental setup, including the simulator used, while in Section V we discuss the results of our experiments. We conclude in Section VI.

II. THE BITTORRENT PROTOCOL

A crucial aspect of BitTorrent is that the exchange takes place by splitting the content to be distributed into fixed-size pieces, thus allowing a client to request different pieces from different peers, either in parallel or at different times. This also allows clients to stop and later resume the data exchange at will, since each piece is essentially exchanged independently of all others.

Typically, the distribution of a new file with BitTorrent starts by preparing and publishing a metafile with a `.torrent` extension. The metafile is distributed using an out-of-band channel, usually by posting it on a web page indexed by a search engine. The `.torrent` metafile contains the tracker address, file size, piece size, and the hashes for the file pieces, among other information. Trackers are responsible for helping peers interested in the same file discover each other so as to form a *swarm*. In most cases, each `.torrent` metafile is served by a single tracker, but extensions to the protocol allow multiple trackers per file or even no trackers at all [3], by using *Distributed Hash Tables* (DHTs) for peer discovery.

Clients locate a tracker from the tracker's URL stored inside the metafile and communicate with it via a simple text-based protocol, layered on top of HTTP. Each client periodically *announces* its contact details (e.g., IP address, TCP port, identification info) and its progress (in terms of total bytes downloaded/uploaded) to the tracker. Note that most of the information announced is for statistical purposes; only the IP address and TCP port of a client are crucial for the protocol to work. After receiving such a *tracker request* the tracker selects a random set of peers that it knows about and returns their contact details to the client. Since each tracker request provides the contact details of a client to the tracker, the tracker

can record this information so as to be later able to construct these replies. As a result of the tracker requests and replies, over time the peers discover increasing subsets of the swarm.

After receiving a set of peers from the tracker, a client attempts to establish TCP connections with these peers. If the connection succeeds, the two peers exchange HANDSHAKE messages to verify their peer identities and ensure that they are interested in the same file. This handshake is followed by an exchange of BITFIELD messages that contain the *bitfield* of each peer. The bitfield is a bitmap denoting the availability of each piece at the peer, i.e. each bit position shows whether the peer has fully downloaded the corresponding piece or not. Based on that information a client can determine whether it is interested in one or more pieces of the remote peer. By repeating this procedure over multiple peer connections, a client gradually collects information regarding the availability of the file's pieces in the subset of the swarm that it has already explored. Based on this information, it then decides which pieces to request from each peer.

In general, if a peer holds some pieces that the client does not already hold, an INTERESTED message is sent to that peer to indicate the client's interest for the peer's data. Initially peers are assumed not to be interested in each other's pieces. Although at this stage a client knows the peers it is interested in, that is, the peers holding pieces that it needs to download, it cannot make any requests for pieces yet, as data cannot be exchanged until the remote peer actively permits this by sending an UNCHOKE message. That is, each client is by default blocked, or, in BitTorrent parlance, *choked* by the corresponding remote peer. The decision to choke or unchoke a client is made based on several criteria embodied in the *choking algorithm* [4]:

- *Reciprocation*: Peers unchoke the clients providing them with the best upload rates.
- *TCP performance*: TCP works better with a limited number of simultaneous exchanges.
- *Fibrillation avoidance*: Frequent choking/unchoking causes data transfer interruptions that may deteriorate protocol performance.
- *Optimistic unchoking*: New peers are occasionally unchoked in order to discover potentially better opportunities. Peers are thus given the chance to acquire their first pieces.

Reciprocation is key to BitTorrent's success: data exchange normally proceeds in a tit-for-tat fashion, in the sense that peers will only download data to a client that is uploading data to them. The exception is optimistic unchokes, which allow clients to download pieces without having anything in return. Therefore, peers can initially rely on optimistic unchokes to receive their first pieces, but in the long run they must be able to upload pieces to other clients, if they are to receive the pieces they need in return.

When a client is unchoked by a peer, it starts sending REQUEST messages, each asking for a specific *block* of the selected piece. The peer sends back the requested data using PIECE messages. Upon completing the downloading of a piece, a client informs via HAVE messages the peers that it has established connections to. These peers update the *bitfield* for that client and may then, potentially, express their interest for that peer with an INTERESTED message.

While a client can use any algorithm it desires in order to select which piece to request from other peers, normally clients select the piece that is less commonly available in the swarm. This ensures that rare pieces will not disappear if a few peers leave the swarm, and, more importantly, it also provides the client with pieces that other peers will probably be interested in. This *rarest first* policy means that a client receives pieces in a seemingly random fashion.

III. STREAMING EXTENSIONS TO BITTORRENT

A. General

Streaming applications favor a sequential piece download policy, so as to be able to start reproducing the initial data before the entire content has been downloaded. The standard BitTorrent piece selection scheme however does not download pieces in sequence, therefore all the proposed streaming extensions to BitTorrent start by modifying the piece selection mechanism. Their goal is to implement a mechanism appropriate for the needs of streaming applications, where prompt piece arrival is critical in order to maintain playback quality. The question then is how to reconcile the sequential piece selection policy required for multimedia streaming with the rarest-first selection policy that makes tit-for-tat work in BitTorrent. Three answers to this question are described below.

B. Fixed-Size Window

The *Fixed-Size Window* (FSW) approach [5] modifies the BitTorrent piece selection strategy by limiting it to a fixed-size sliding window over the pieces. The window starts from the first non available piece and includes k consecutive pieces, where k is a configuration parameter; additional pieces can only be selected for downloading within this window. Essentially, rarest-first operates within the window, thus allowing the pieces closest to their playback time to be downloaded, without "wasting" bandwidth to download other pieces. Periodically, the next piece in sequence is delivered for playback; if the piece is incomplete, a loss event is signaled and the window slides to the right, thus abandoning this piece. Note that the window also slides to the right when its first piece has been downloaded, therefore there is no need to also slide the window when the player consumes a downloaded piece.

The designers of the FSW approach have also proposed a modification to the optimistic unchoke scheme. Specifically, they proposed that each client should randomly select some peers at every playback interval and optimistically unchoke them, so as to help new clients receive their first pieces [5]. We did not consider this additional extension in our performance comparison.

C. High-Priority Set

The basic problem with the FSW approach is that it does not exploit the available piece download opportunities that well. First, by fixing the window size it limits the choice of pieces as the window fills up, thus wasting available downlink

bandwidth. Second, by limiting the client 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 [6], a fixed-size set (the HPS) holds the next pieces in sequence that have not been downloaded already. While in the FSW approach a 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 a set of size k covers at least k pieces in the piece sequence space, out of which exactly k pieces need to be downloaded.

Besides maintaining a fixed size for the HPS, in this approach pieces outside the HPS can also be requested. For this decision, a probability p is configured as a parameter; with probability p a client will select a piece from the HPS and with probability $1-p$ the client will select a piece beyond the HPS, in both cases using the rarest-first policy within each set. As in [5], when the player reaches a non available piece, a loss event is signaled and the HPS is modified as that piece is abandoned; the HPS is also modified whenever any of its pieces completes downloading, as this piece must be removed from the HPS.

The HPS scheme also assesses if a currently downloading piece will be completed on time, i.e. before the player reaches it, dropping pieces that cannot be completed in time [6]. We did not consider this additional extension in our performance comparison.

D. Stretching Window

In the *Stretching Window* (SW) approach, elements of the FSW and HPS approaches are combined [7]. The SW behaves like the HPS in that it contains (up to) k non downloaded pieces, but pieces are only requested from within the SW. In addition, 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. Other than that, SW operates exactly as HPS.

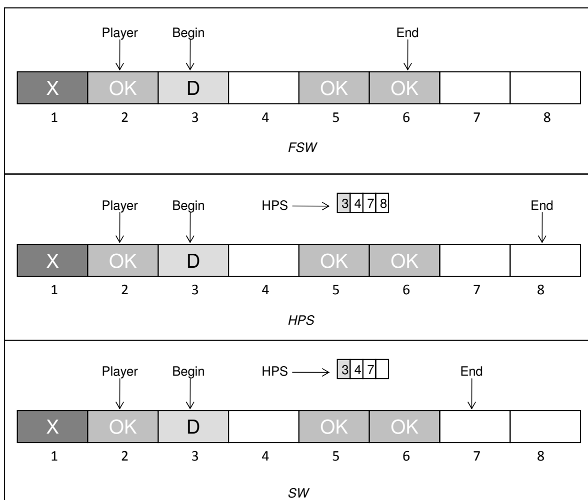

Fig. 1. Protocol extension functionality.

The functionality of these three protocol extensions is explained in Figure 1. Pieces marked with an X were not downloaded in time, 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 progress piece 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 numbers of the pieces in the HPS are also shown. 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.

IV. Experimental Setup

A. BitTorrent Simulator

For the simulation-based comparison, we modified our own detailed OMNeT++ Simulator for BitTorrent [8], in order to add 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 [9], a fact that makes the results more realistic and stresses protocol performance to its limits. We have used transit-stub topologies generated by the GT-ITM module [10], including both core and access routers. Each scenario was executed 10 times with different random seeds, which means that peers were deployed in different ways in the network.

B. Experiments

Our results are based on a scenario with one initial seeder and 120 peers (*leechers*) which join the swarm at random times, starting from scratch. This means that peer joins are incremental, instead of a flash crowd join. The topology we used consists of four *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 was 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 192 KB, which translates to 6 seconds of video. In this manner, a piece that has not been downloaded on time, will cause a 6 second gap in the video, but it will not prevent the next piece from decoding. The block size was set to 16 KB. The entire video size was 200 MB, which corresponds to around 106 minutes of playtime, or 1067 pieces in total.

TABLE I
SIMULATION PARAMETERS

Parameter	Value
Video size (in MB)	200
Video Bit rate (in Kbps)	256
Piece size (in KB)	192
Block size (in KB)	16
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% or 90%
Bound in pieces (only for SW)	30 or 100

We kept all the default BitTorrent settings unchanged e.g. optimistic unchoke interval, number of connections per peer, as given in [8]. To make our model relevant to live streaming, we assumed that each peer will remain in the swarm and seed other peers until the video playback reaches its end, even if the peer has already completed the video download, i.e. that the peer leaves 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, modeling a video source that exploits BitTorrent extensions to offload some of the traffic to the peers.

In addition, we assumed that 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 period, we use the rarest-first policy only for these first pieces, until they are all completely downloaded. We set the base window size k to 2% or 8% of the entire number of pieces for all algorithms, which in our case translates to 21 or 85 pieces, respectively. The probability p was set to either 80% or 90% for HPS. The bound l for SW was set to 30 pieces for $k = 21$ and to 100 pieces for $k = 85$, therefore SW can grow its window more than FSW, but not as much as HPS. Table I summarizes these parameters.

C. Metrics

We used the following metrics for the evaluation of the various proposed BitTorrent extensions:

- *Piece loss*: The most important metric in our case is piece loss. A piece is characterized as lost if the player reaches it and the piece is not completed, leading to a gap in the video; this includes the case where the piece is currently being downloaded.
- *Prefetch time*: We measure the necessary time for prefetching the first pieces of the video file. In other words, the time the user of the video application should wait for the player to start.
- *Download duration*: This metric shows how much time the entire download took; it is interesting to compare this with the (fixed) playing time.

V. EXPERIMENTAL RESULTS

In the following we present the simulation results concerning the protocol modifications presented above.

A. Piece loss

In Figure 2 we show the piece loss for each protocol modification, with the window set to 2% of the video size. It is obvious that with more prefetched pieces the performance slightly improves with all protocols; the penalty is an increased delay before starting playback, as shown in the next section. While all protocols exhibit acceptable performance, HPS works best, with piece losses of around 1-1.5%, while FSW and SW suffer from losses of around 1.5-2.5%. Among the two different probabilities in HPS for selecting pieces within the window, the highest one (90%) works slightly better. On the other hand, FSW and SW have nearly identical performance, despite the fact that SW can grow its window to nearly 50% more than FSW.

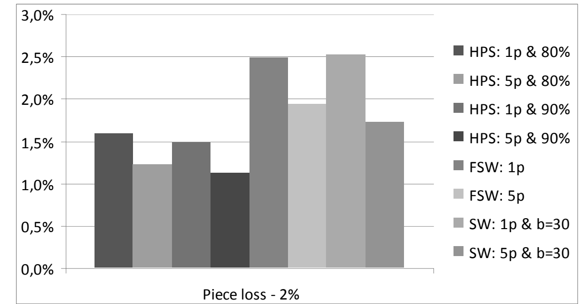

Fig. 2. Piece loss for 2% window size (%).

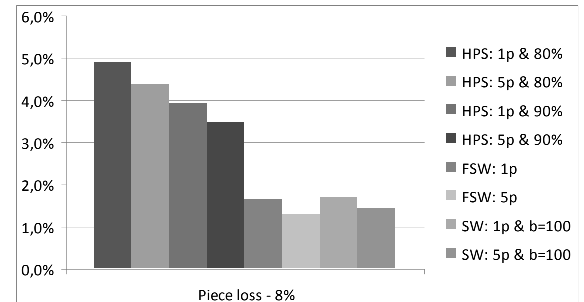

Fig. 3. Piece loss for 8% window size (%).

Figure 3 shows the corresponding data for a window size of 8%. In this case HPS becomes the worst performer, with losses of roughly 3.5-5%, while FSW and SW actually work better than in the previous case, with losses of 1.5%. Again prefetching more pieces leads to better performance in all schemes. For HPS, the higher probability for selecting pieces within the window works noticeably better, while FSW and SW have again nearly identical performance.

Looking at both scenarios, it is clear that the window size is an important parameter. With a 2% window (21 pieces), FSW and SW cannot keep up with HPS which can grow its window more. With a 8% window however, FSW and SW have enough pieces to download; there is no need to grow the window too much. Regarding the HPS option of downloading outside the window, in both scenarios HPS works better when it is less likely to do so, while FSW and SW that do not

offer this option work even better when the window is not too small. Finally, note that FSW and SW are more predictable than HPS, since their performance changed only slightly from one scenario to the other.

B. Prefetch time

The buffering time at the beginning of a video fluctuates between 21 and 33 seconds (average values), depending on the number of pieces we have chosen to download; Figure 4 shows the data for a 2% window size, while Figure 5 presents the case for a 8% window size. Note that when only one piece is prefetched, there is essentially no buffering: it is necessary for the first piece to be downloaded in order for the player to begin anyway, therefore these experiments show the minimum buffering time. With five pieces prefetched, the user must wait a little bit longer, around 50% more, hoping for better performance later on, although the improvements in loss rates are low, as discussed in the previous section.

These waiting times seem to be large, but this is due to the latency from the first joined peers in the swarm, when there are not many sources available, except for the initial seeder, therefore peers have to wait in order to receive these first pieces via opportunistic unchokes. Note that there are hardly any differences between the various protocols in this metric, since the protocol modifications have not started operating yet. The same is true of the window size, which is disregarded during the prefetch period.

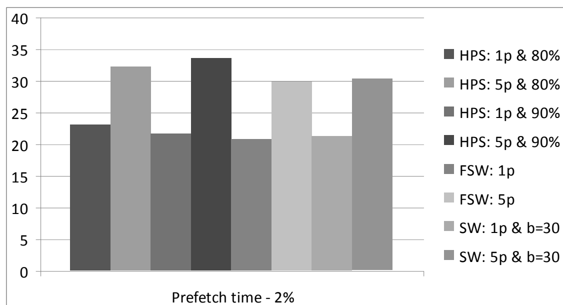

Fig. 4. Prefetch time for 2% window size (sec).

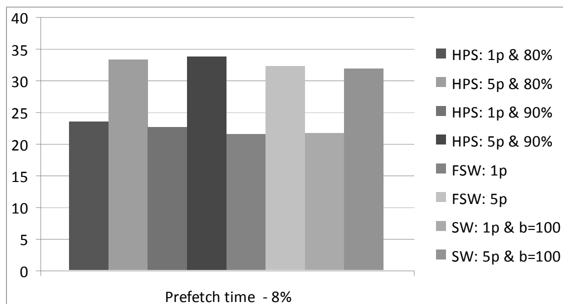

Fig. 5. Prefetch time for 8% window size (sec).

C. Download time

The time required to complete the video download, as shown in Figures 6 (2% window size) and 7 (8% window size), is lower for HPS and higher for FSW and SW. While SW and FSW are nearly identical, in HPS it seems that a higher probability to download within the window leads to worse download times (recall that it also led to lower loss rates, so there is a tradeoff here). On the other hand, the number of prefetched pieces only has a minor impact on completion time, as the prefetch period is dwarfed by the rest of the download.

We point out that since the playback time is 6400 sec, all schemes manage to complete the download well before the playback ends. With a larger window, the downloads complete even faster. Since clients remain in the swarm until their download completes, this means that there are always plenty of peers to exchange pieces with, except in the very beginning of the exchange, when everyone has to rely on the initial seeder; we surmise that most of the observed losses are due to these first peers.

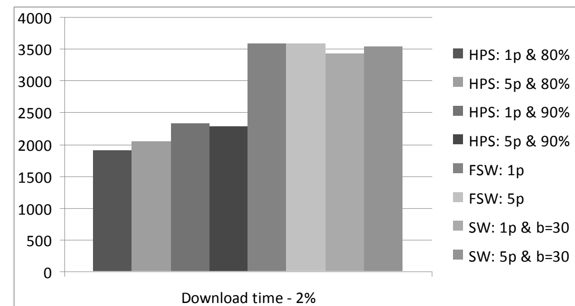

Fig. 6. Download time for 2% window size (sec).

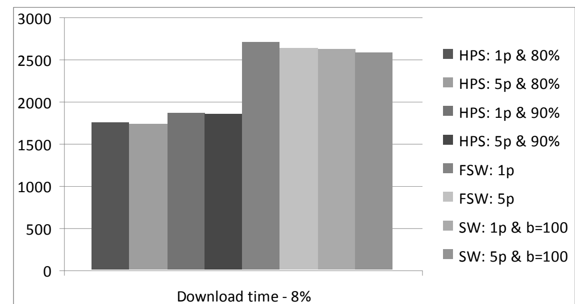

Fig. 7. Download time for 8% window size (sec).

D. Overall Evaluation

Based on the above evaluation, it is clear that there is no dominant scheme; HPS exhibits lower loss rates than FSW and SW for smaller windows, but with larger windows the situation is reversed. Since HPS works slightly better with a lower probability to request pieces outside the window, the ability of HPS to download outside the window does not seem beneficial. FSW and SW perform nearly identically, therefore the extra complexity of SW does not seem to be worthwhile.

Prefetching does lower the loss rates slightly, but at the cost of adding nearly 10 extra seconds of startup delay until the pieces are downloaded. Regarding download times, HPS is clearly the winner, but since all protocols complete the download well before the end of playback, this may not be as important as a reduced loss rate.

VI. Conclusion and Future Work

In this paper we presented a simulation based performance comparison of three protocol extensions that add multimedia streaming capabilities to BitTorrent. We explored, through simulations using detailed TCP/IP message exchanges, the factors that impact protocol performance and behavior, using the exact same setup for each protocol extension. We focused on three metrics: *piece loss*, *prefetch time* and *download time*. The collected data showed that the HPS approach presents the best performance in terms of download time, but its loss rate heavily depends on the window size. The SW and FSW approaches provided nearly identical performance, therefore there seems to be no point in the additional complexity of SW over FSW. While SW and FSW led to higher download times than HPS, they offered lower loss rates when the window size was large enough.

Future work includes running additional experiments with a larger set of parameters, such as different data rates, numbers of peers, network topologies, prefetch window sizes and stretch limits and HPS probabilities. We also plan to study the distribution of losses among downloads, in order to verify whether the average observed loss rates are mostly affected by the first peers joining the swarm. Another direction is changing the player model to stall when the next piece is missing instead of skipping that piece, as in most current streaming players; in that case, the main metric would be the number and duration of stall periods, rather than the number of lost pieces.

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*, 2005, pp. 63–76.

[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] BitTorrent.org, “Torrent file extensions,” Available at http://www.bittorrent.org/beps/bep_0005.html, 2011.

[4] BitTorrent development community, “Bittorrent protocol specification v1.0,” Available at http://wiki.theory.org/BitTorrentSpecification, 2011.

[5] P. Shah and J. Paris, “Peer-to-peer multimedia streaming using BitTorrent,” in *Proc. of the IEEE International Performance, Computing, and Communications Conference*, 2007.

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

[7] 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.

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

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

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