ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΣΤΗΝ ΕΠΙΣΤΗΜΗ ΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία
Μεταπτυχιακού Διπλώματος Ειδίκευσης

«*Evaluation of video streaming extensions to BitTorrent*»

Νικόλαος Μπλάτσας

Επιβλέπων: Γεώργιος Ξυλωμένος

ΑΘΗΝΑ, ΙΟΥΛΙΟΣ 2011

Abstract

It has been observed in the past few years that more than 60% of all Internet traffic is due to by P2P applications. One of the most common protocols is BitTorrent, which is one of the most effective mechanisms for P2P content distribution. Although BitTorrent was created for the distribution of time insensitive content, in this work it is shown that with minimal changes it can support video streaming. There are two the main reasons that this work is important. First, the ability we have of watching a video before the complete download of the file, and second, as a consequence of the first, the possibility to evaluate the quality of the video early and decide if this particular video is worth spending our time and resources on.

The research was conducted using our full featured and extensible implementation of BitTorrent for the OMNeT++ simulation environment developed in the Mobile Multimedia Laboratory of Athens University of Economics and Business. The implementation includes modifications in the BitTorrent piece selection strategy, giving higher download priority to pieces that are close to be reproduced by the player. For this reasonm we had to radically change the original rarest-first BitTorrent policy. This was achieved by using a sliding window (containing the pieces that are close to be reproduced by the player), proportional to the file that we download; within the window, the piece to download is chosen with a variety of techniques. Whenever a piece does not meet the player's deadline, we use one of two strategies. The first is similar to the logic applied in YouTube: when the player finds a piece that it wants to play but does not have it, it stalls until the piece is downloaded and is available to the player. The most important metric in this technique is the time that each user must wait. In the second strategy, when the player finds a missed piece, it checks how many blocks of the piece have been downloaded and then changes its operation to examine the type of frame in each block of the piece. If the remaining blocks are downloaded in time, then the player continues without interruption, otherwise we calculate the lost blocks as well as their importance. In this technique we have two metrics; the total missed blocks and the specific blocks missed, as mapped into frames. Based on these measurements we came up with our results.

Keywords

Peer-to-Peer, BitTorrent, streaming, simulation, OMNeT++, peer, seeder, swarm, piece, block, swarm, sliding window, high priority set, probability, block loss, missed blocks, time loss.

Περίληψη

Τα τελευταία χρόνια έχει παρατηρηθεί πως πάνω από το 60% της κίνησης στο διαδίκτυο οφείλεται στις P2P εφαρμογές. Ίσως το πιο γνωστό πρωτόκολλο είναι το BitTorrent, το οποίο έχει εξελιχθεί στον πιο αποτελεσματικό P2P μηχανισμό για διαμοιρασμό αρχείων. Παρά το γεγονός πως ο σκοπός δημιουργίας του είναι ο διαμοιρασμός αρχείων, στην τρέχουσα εργασία καταφέραμε με λίγες αλλαγές στο πρωτόκολλο του BitTorrent να το κάνουμε να υποστηρίξει και video streaming. Δύο είναι ίσως οι σημαντικότεροι λόγοι που κάτι τέτοιο είναι αρκετά χρήσιμο. Πρώτον η ικανότητα να βλέπουμε το video πριν την ολοκλήρωση της λήψης του και κατά δεύτερον σαν συνέπεια του πρώτου η δυνατότητα που μας δίνεται σαν χρήστες αν δεν μας αρέσει η ποιότητα του video να το απορρίψουμε από την αρχή κερδίζοντας έτσι σε πόρους και χρόνο που θα χρειαζόμασταν για την λήψη ολόκληρου του video.

Για την διεξαγωγή της έρευνας χρησιμοποιήθηκε ένα BitTorrent module για το περιβάλλον προσομοίωσης Omnet++, το οποίο αναπτύχθηκε στο εργαστήριο «Ασυρμάτων Δικτύων και Πολυμεσικών Τηλεπικοινωνιών» του Οικονομικού Πανεπιστημίου Αθηνών. Η υλοποίηση περιλαμβάνει τροποποιήσεις στην στρατηγική επιλογής κομματιών (pieces) του BitTorrent, δίνοντας μεγαλύτερη προτεραιότητα στα κομμάτια εκείνα που είναι πιο κόντα στην αναπαραγωγή τους από τον player. Για να γίνει αυτό έπρεπε να έρθουμε σε αντίθεση με την πολιτική του rarest – first με την οποία λειτουργεί το κλασικό BitTorrent. Για να το πετύχουμε αυτό χρησιμοποιήσαμε ένα κυλιώμενο παράθυρο (το οποίο περιέχει τα κοντινότερα στην αναπαραγωγή κομμάτια), ανάλογο του μεγέθους του αρχείου λήψης, μέσα από το οποίο επιλέγεται κάθε φορά κάποιο κομμάτι, με διάφορες τεχνικές. Στα κομμάτια τα οποία δεν έχουν ληφθεί κατά την χρονική στιγμή που ο player φτάνει για να τα παίξει εργαστήκαμε με δύο τεχνικές. Η πρώτη τεχνική είναι παρόμοια με τη λογική που εφαρμόζεται στο youtube και έχει να κάνει με το γεγονός πως ο player όταν φτάσει σε κάποιο κομμάτι το οποίο πρέπει να παίξει αλλά δεν είναι διαθέσιμο περιμένει έως ότου το ζητούμενο κομμάτι ληφθεί. Είναι φανερό πως εδώ βασική μετρική μας για τα σενάρια προσομοίωσης είναι ο χρόνος που χρειάστηκε κάθε peer του συστήματος να περιμένει. Στην δεύτερη τεχνική κατά τη διάρκεια που διαπιστώσουμε πως ο player έχει φτάσει σε κομμάτι το οποίο δεν είναι διαθέσιμο, ελέγχουμε πόσα blocks από το κομμάτι έχουν ληφθεί και η λειτουργία τροποποιείται με βάση την αντιστοίχιση πλαισίων σε block. Αν μέσα στο χρόνο που γίνεται η αναπαραγωγή προλάβουμε να λάβουμε τα blocks που υπολείπονταν για το κομμάτι η λειτουργία του player συνεχίζεται χωρίς καμία διακοπή ενώ σε διαφορετική περίπτωση υπολογίζουμε τα χαμένα blocks καθώς και τη σπουδαιότητά τους και περνάμε και πάλι σε κανονική λειτουργία. Σε αυτή την τεχνική, μετρική μας είναι τόσο το συνολικό πλήθος των χαμένων blocks από κάθε peer του συστήματος όσο και ποια συγκεκριμένα blocks χάθηκαν σε κάθε peer του συστήματος και σε τι είδη πλαισίων αντιστοιχούν. Βάσει αυτών των μετρικών θα παρουσιαστούν τα αποτελέσματα των προσομοιώσεων.

Λέξεις κλειδιά

Σύστημα ομότιμων, BitTorrent, streaming, προσομοίωση, OMNeT++, χρήστης, seeder, swarm, piece, block, swarm, κυλιώμενο παράθυρο, high priority set, probability, block loss, missed blocks, time loss.

Acknowledgements

First of all, I would like to thank my supervisor, George Xylomenos for giving me the chance to work on this topic and for his support throughout this work. I also would like to thank Professor George Polyzos for evaluating my work.

I owe my deepest gratitude to Charilaos Stais, PhD student of our department for helping, encouraging and his continual support whenever I needed. Furthermore, I would like to thank PhD student Pantelis Frangoudis for his valuable advice.

Last but not least, I want to thank my parents, Athanasios and Andriani, for supporting and encouraging me in all these years of my academic adventures.

Table of Contents

List of Figures

List of Tables

1. Introduction

In the past few years we have witnessed an explosive growth of the Internet. The reason for this is the ubiquitous adoption of the TCP/IP protocol stack. This kind of network simply forwards traffic between pairs of communication end hosts. The traditional model for content distribution over the Internet was the client-server one, where information stored in a central server and clients can search for and download content (e.g. files) from the server. However, in this scheme, the server can become a single point of failure. To solve these problems, novel p2p decentralized services were been introduced on the information itself, rather than on the end hosts providing or consuming it, with BitTorrent being the most well-known of these services. [15][16]

It has been observed that in recent years more than 60% of all Internet traffic is due to p2p applications and 27-55% to BitTorrent [2]. From these numbers we can see the reason that many researchers try to use BitTorrent for multimedia streaming purposes, despite the fact that BitTorrent was designed in the context of non real-time exchanges. A lot of proposals for modifications to the BitTorrent protocol exist, but their simulation based performance evaluation is lacking; simulators do not take into account the entire protocol stack overhead or retransmissions in the network.

In this work, we aim to address all these issues by presenting a thorough comparison between three proposed multimedia streaming extensions to BitTorrent (and two modifications of one extension) with two strategies for dealing with losses, using the same assumptions and metrics for each protocol based on a detailed packet level BitTorrent simulator, enhanced with streaming oriented modifications.

2. Classic BitTorrent

2.1. Overview

BitTorrent [1] is a peer-to-peer file sharing protocol used for distributing large amounts of data and one of the most common protocols for transferring large files. It has been observed in recent years that more than 60% of all Internet traffic is due to p2p applications and 27-55% of all Internet traffic is due to Bittorrent. [2]

The benefits of using BitTorrent are many. First, users can distribute large amounts of data without the heavy demands on their computers that would be needed for standard Internet hosting. Second, we can achieve bandwidth scalability through the operation of BitTorrent, thus overcoming the bandwidth limitations of keeping the file in a server.

Each user can distribute files via BitTorrent, as it plays the role of a file-provider when it makes a file available to the network. This user is called seeder and the other users who download from the seeder are the peers. Initially the peers take any piece that the seeder gives them (that is, a random piece) and after that BitTorrent allows peers to become a source for the pieces they have. In this manner, each user will download the file successfully. After that, the peer is able to shift roles and become an additional seeder, thus helping the overall 'health' of the file.

2.2. Operation

BitTorrent clients like KTorrent, µTorrent, rTorrent are examples of programs that implements the Bittorent protocol in which we can prepare, request, and transmit any type of computer file over a network. A peer is any computer running an instance of a client. There are two main differences between BitTorrent and centralized solutions:

- BitTorrent makes data requests over multiple TCP connections to different machines, while in centralized solutions data transfer is typically made via a single TCP connection to a single machine (server).
- In centralized solutions we download the file sequentially, while in BitTorrent we download pieces through the rarest-first [11] approach.

These differences allow BitTorrent to achieve much lower costs for the content provider, much higher redundancy, and much greater resistance to abuse or "flash crowds" than regular server software. However, while in centralized solutions we can rise to full speed very quickly (if the network is not overloaded), in BitTorrent we cannot achieve full speed quickly, because of the TCP connections that must be established with the other peers so that the client can receive and upload data to other peers. In addition, a peer will experience the best download speed when it has a lot of pieces (especially rare pieces) so that it can be an effective uploader.

The most intelligent approach that makes BitTorrent a 'strong' program is its ability to resist the free-rider phenomenon, in which some peers want only to download pieces without uploading to other users. To make this feasible it uses the Tit-for-Tat policy, where each user uploads only to peers that have something to exchange; if this is not the case, the choking mechanism is invoked and the peer stops uploading to its neighboring peer. [4]

2.3. Creating and Publishing Torrents

The peer distributing a data file treats the file as a number of identically sized pieces. For each peace, it creates a hash and records it in the torrent file. The hash of a piece help peers check if they have the correct piece by comparing the hash numbers. The initial peer that makes available the file to the network is called the initial seeder, while any other peer that has downloaded the complete file is called a seeder. [17]

The torrent file in all versions of the BitTorrent protocol has the suffix .torrent and inside this it has an 'announce' section, which specifies the URL of the tracker, and an 'info' section, containing names for the files, their lengths, the piece length used, and a SHA-1 hash code for each piece, used by peers to verify the integrity of the data they receive. Torrent files are typically published on websites or elsewhere, and registered with at least one tracker.

2.4. Downloading Torrents and Sharing Files

In order to download a file via BitTorrent, we first have to find it in the web (in a page where we can browse BitTorrent files), download it and open it with a BitTorrent client. When we open the torrent file, the client connect to the tracker that is contained in the file and after established a connection with the tracker, the tracker responds with a list of peers participating in the swarm. After that, the client connects to those peers to obtain the various file pieces. If the swarm is only the initial seeder, the client connects with it and begins to request pieces.

Clients incorporate mechanisms to optimize their download and upload rates. For this reason peers download from the other peers through the rarest-first policy, so they will later have rare pieces to exchange with other peers, so as to avoid being chocked by the other peers; this is the tit-for-tat policy. While this leads us to a fair distribution, it has one disadvantage: when new peers join the swarm, they are unable to receive any data since they do not have data to exchange so they are in a chocked situation. To counter these effects, the original BitTorrent client program uses a mechanism called "optimistic unchocking", whereby the client reserves a portion of its available bandwidth for sending pieces to random peers in the hope of discovering even better partners. As a side-effect, this ensures that newcomers get a chance to join the swarm. [18]

Although swarming scales well and tolerates flash crowds for popular content, it is less useful for unpopular content. For this reason some peers arriving after the initial rush might find the content unavailable, since some rare pieces are not available; they have thus to wait a lot of time until some seed arrives in the swarm, providing the remaining pieces. Measurements have shown that 38% of all new torrents become unavailable within the first month. [19]

3. BitTorrent Module for OMNeT++

Even though peer to peer content distribution remains one of the most active research areas, little progress has been made towards the study of the BitTorrent protocol, and its possible variations, in a fully controllable but realistic simulation environment. In the Mobile Multimedia Laboratory a full featured and extensible implementation of BitTorrent for the OMNeT++ simulation environment was developed [3]. This implementation is briefly described in the following sections.

3.1. The Tracker Protocol

When we want to use BitTorrent we need to first browse the web and find what we want. After downloading the torrent file, the BitTorrent client uses the information included in the .torrent file to connect with the tracker. Trackers are responsible for aiding peers to discover each other and form a swarm. The .torrent files at most cases including only one tracker that will help us to download the file, but recent extensions allow the inclusion of many trackers.

The communication between the tracker and the clients uses a simple text-based protocol, layered on top of HTTP/HTTPS, using the tracker's URL stored inside the metafile. [12] Periodically a client communicates with the tracker and publishes its progress, as well as its contact details. However, from all the parameters sent to the tracker, only the IP address and TCP port are crucial. After each such message, called a tracker request, the tracker randomly selects a set of peers and returns their contact details in a bencoded dictionary (tracker response). [9]

3.2. The Peer Wire Protocol

The peer-wire protocol provides the core BitTorrent functionality (i.e., interaction with remote peers). In the following we first present an overview of the protocol and then proceed with the details of its operation, focusing on the most important features available in our implementation.

3.2.1. Protocol Overview

When the client connects to a tracker, the tracker randomly selects a set of peers and returns their contact details in a bencoded dictionary. Then the client tries to establish TCP connections with each one of the peers in the list. To establish the connection a three-way handshake is required, to ensure that both ends are interested in the same .torrent file. After the three-way handshake with every peer we connect, BITFIELD messages are exchanged that contain the bitfield of each client (the pieces it has). Based on this the client can determine what pieces it is interested to received from each peer. [13] In case a peer does not have any pieces, this would result in the exchange of useless information, and is therefore avoided in our implementation.

By following the above procedure over multiple peer connections, a client collects information regarding the availability of the pieces that it is still missing in the subset of the swarm explored thus far. The next step for each client is to determine what pieces it is interested in from each peer. If the peer does not hold any piece that the client is interested in, the clients sends a message NOT INTERESTED. At the beginning of a connection a client is in choked mode, which means that it is blocked and does not have the opportunity to exchange pieces. So, the client must wait to receive an UNCHOKE message from a peer. The decision to unchoke, or not, a client is made based on several criteria embodied in the choking algorithm: [9]

- ***Reciprocation:*** Peers unchoke the clients that provide the best upload rates.
- ***TCP performance:*** TCP behaves better when the number of simultaneous upload is capped.
- ***Fibrillation avoidance:*** Frequent (un-)chocking causes data transfers interrupts that deteriorate protocol performance.
- ***Optimistic unchocking:*** New peers are occasionally unchecked so as to discover potentially better connections. This is also how new peers acquire their first pieces.

Only when a client is unchoked, it can then start to request data from a peer. When it downloads a piece, it changes his BITFIELD and informs all the peers in its peer list, and after that peers may express their interest for that piece.

3.2.2.Connections

A client periodically learns about other peers by utilizing the Tracker protocol and parsing the peer list returned. The client joins the swarm by establishing connections with some of those peers. However, as noted in [9], each connection incurs an increase in signaling traffic, especially for bitfield maintenance via the exchange of HAVE messages. Thus, our implementation provides configurable lower and upper bounds for the number of established connections, using the minNumConnections and maxNumConnections configuration parameters (see Table I).

3.2.3.Piece downloading strategy

The piece downloading strategy refers to the policy followed in the selection of the pieces that will be requested from a peer. It is an important aspect of BitTorrent as it heavily affects the diversity of the pieces available in each peer. A low degree of diversity would result in low interest for a peer's pieces, thus causing degraded application performance. We have implemented the two most prevalent piece downloading strategies: rarest first and random first. Based on the information gathered during the BITFIELD and HAVE message exchanges, the former strategy selects those pieces that appear less frequently in a client's set of connected peers. This selection is randomized among several of the less common pieces, according to the rarest list size configuration parameter (see Table I), in order to avoid multiple peers converging on the same piece. This way, peers download pieces that most other peers probably want, therefore facilitating data exchange. However, rare pieces are present only in a few peers, and it is possible that downloading from them may be interrupted due to a choking decision. Clients with no pieces in their possession would therefore have to wait for an optimistic unchoking event from a peer holding the same rare piece in order to continue downloading. The latter strategy avoids this problem by selecting a random piece which is more likely to be available from multiple peers, so that a choking decision would not have such an adverse effect.

3.2.4. Endgame mode

This mode is used by peers that have downloaded most pieces of the file and do not want to face slow transfers for the last data blocks. A client entering this mode sends REQUEST messages for each missing block to all peers that are not chocking it. However in our implementation it does not send these messages to all peers that are in its peer set since a peer chocking the client will discard the request. In our implementation a client enters this mode when the number of missing blocks equals the number of requested blocks.

Parameter	**Default Value**
file size (MB)	700
piece size (KB)	256
block size (KB)	16
DHT port	-1
pstr	BitTorrent protocol
pstrlen	19
keep alive (sec)	120
have suppression	true
chocking interval (sec)	10
downloaders	4
optUnchockedPeers	1
optUnchocking interval (sec)	30
seederDownloaders	4
seederOptUnchockedPeers	1
rarest list size	5
minNumConnections	30
maxNumConnections	55
timeToSeed (sec)	0
request queue length	5
super seed mode	false
end game mode	true
maxNumEmptyTrackerResponses	5
newlyConnectedOptUnchockedProb	0.75
downloadRateSamplingDuration (sec)	20

Table I: Peer-wire protocol parameters

4. Streaming extensions to BitTorrent

4.1. General

Many streaming extensions have been proposed to support streaming applications based on BitTorrent. Their basic policy is to modify the piece selection strategy, since BitTorrent uses either a random first or a rarest first policy, methods inadequate for multimedia streaming. So the basic question is how we can bypass these strategies, without returning to sequential downloads. We describe below three answers. [4][5][6][7]

4.2. Fixed Size Window

The Fixed Size Window (FSW)[7] method uses a sliding window (of a fixed size) for piece selection. The FSW includes two modifications that allow us to deliver multimedia data on time. The first modification concerns the replacement of the rarest-first chunk downloading policy of BitTorrent by a policy requiring peers to download first the chunks that they will watch in the near future. The second modification is a new randomized tit-for-tat peer selection policy that gives free tries to a larger number of peers and lets them participate sooner in the media distribution.

The window in FSW covers k consecutive pieces starting from the first non available piece (k is a parameter of the program). The window contains both downloaded pieces, pieces that are currently downloaded and non requested pieces. Inside the window, the selection of a piece follows the rarest-first policy, so that the BitTorrent client will receive rare pieces to exchange later, so to avoid being choked by the other peers. Pieces for downloading are chosen only inside this window. Note that the window slides to the right when its first piece has been downloaded. We did not consider the second modification in our implementation.

4.3. High Priority Set

4.3.1. Classic mode

One problem with FSW is that it does not consider rare pieces outside the window, which may become useful in the tit-for-tat exchange. Another problem is that by fixing the window size it limits the choice of pieces as the window fills up, thus wasting available downlink bandwidth. These problems are solved by the High Priority Set (HPS) or BiTOS approach [4], in which a fixed size set (the HPS) holds the next pieces in sequence that have not been download already. In contrast to the FSW method where a window of size k covers exactly k consecutive pieces, out of which at most k pieces have not been downloaded, in the HPS method 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 or are currently being downloaded.

In the HPS method we can download pieces outside the window, using again the rarest first piece selection policy, so that we do not waste downlink bandwidth and have the opportunity to download rare pieces outside the window. This increases the probability that the client will not be chocked in the future (due to the Tit-for-Tat policy), by having enough rare pieces to exchange with the other peers. We choose to download from inside the window with a probability p (which is a parameter of the program), and outside the window with a probability 1–p, in both cases choosing a piece with the rarest–first policy.

The HPS is modified whenever any of its pieces completes downloading, as this piece must be removed from the HPS and added to the Received Pieces Set. This is the main difference from FSW, as in HPS the window only holds pieces that are currently being downloaded or have not been requested yet.

4.3.2. Variable probability per piece

In media-streaming applications we want the initial pieces as the buffer starts to play, so that the player will not stall or skip frames. Therefore we examined some changes to the basic HPS approach. In HPS we normally download a piece through a rarest-first selection policy in the entire HPS, while in our modified approach we set a probability for each piece inside the window. The probability is based on the position of the piece in the HPS; for piece i, the probability for selection is $1/2^i$, as shown in Fig.1. Thus, earlier pieces have a higher probability than the latest pieces of the HPS, since we want to download the initial pieces early; the first piece will be chosen with a probability of 50%. Each time we want to download a piece we randomly select a number in the range (0, 1] and compare it with the cumulative probability of each piece inside the window. If the probability of the piece is higher than the random number we have chosen, then this piece will be downloaded. This applies only to pieces in the HPS. If we are to select a piece outside the window, then we use the rarest-first policy.

The negative aspect of this approach is that we do not use the rarest-first policy to choose inside the window, a fact that may conflict with the Tit-for-Tat policy: we may not have any rare pieces to exchange, ending up choked by our peers.

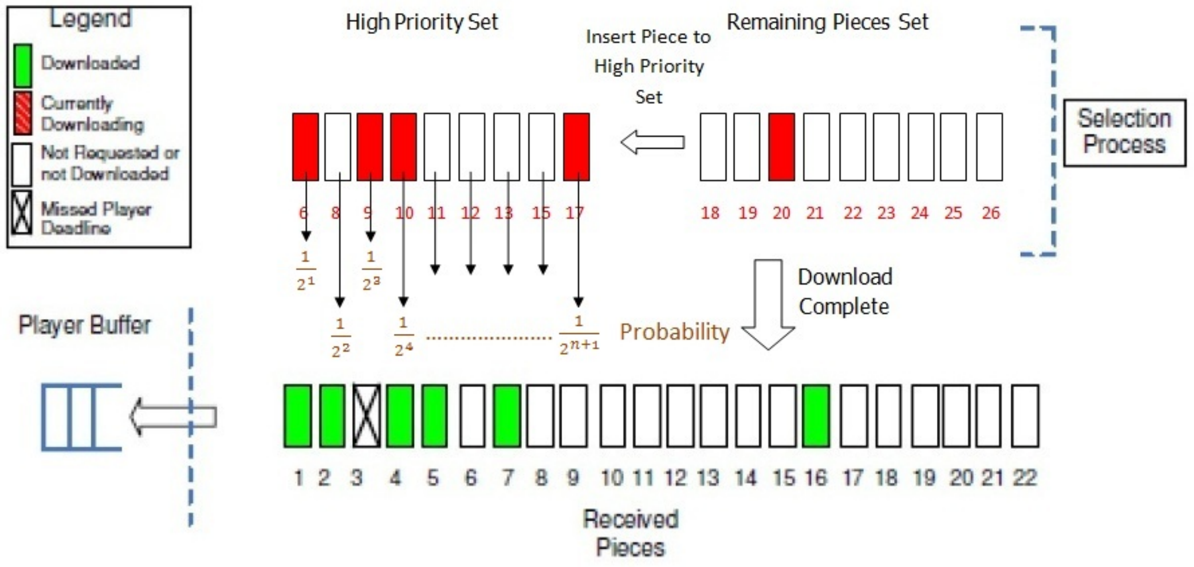

Figure 1: BiToS extension with probability per group inside the HPS

4.3.3. Variable probability per group

Another variant of the HPS that we examined, assigns a different probability to each group of pieces, again favouring earlier pieces by assigning group i a probability of $1/2^i$. After selecting a group, we use rarest-first within the group. The size of the group is a parameter of the program, and it is calculated as 25% of the window size. The reason that we separate pieces into groups is to avoid the overhead of calculating the probability of each piece as in the second approach, while also allowing the rarest-first policy to be used within each group, in the hope that we may experience better performance overall. This variant is illustrated in Fig. 2.

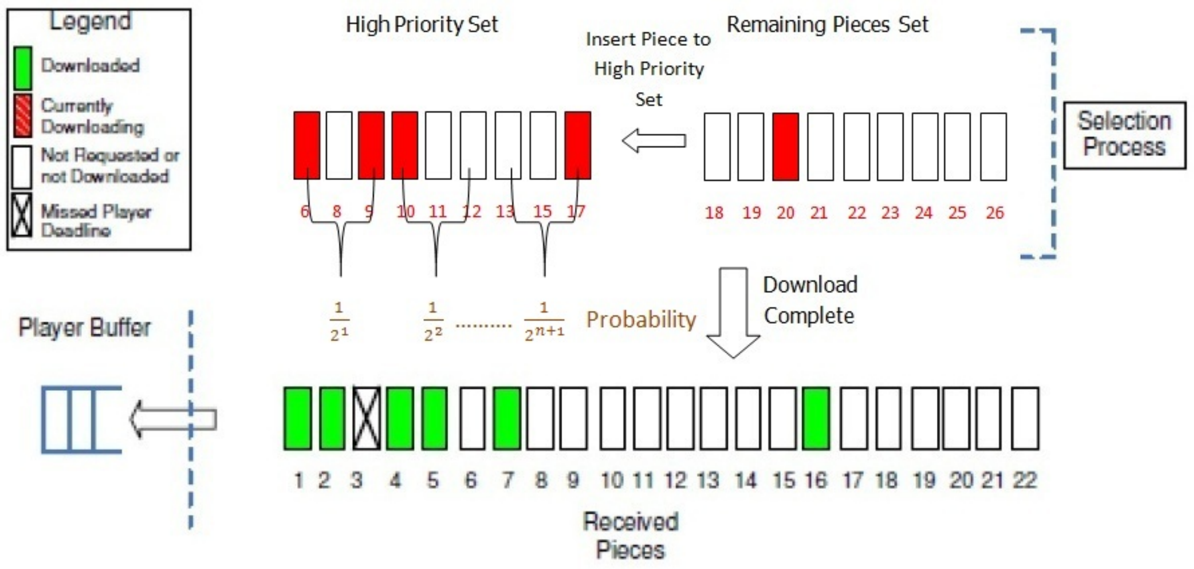

Figure 2: BiToS extension with probability per group inside the HPS

4.4. Stretching Window

The Stretching Window (SW) approach [6] is a mix of the FSW and HPS approaches. The SW resembles HPS in the fact 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. The differentiation with FSW is only that the size of the window can be stretched rather than being constant. A parameter of this policy is the l to be used for limiting the growth of the window.

4.5. An example of the three policies

In this section we describe with an example how the three streaming extensions that we use in this work operate. As we can see in Fig. 3, we use the same parameters for the three extensions. With X we represent missed pieces, that is, those that the player did not have available when it was time to play them. With OK we represent pieces that have completed downloaded and are available for the player. With D we represent the pieces that are currently downloading.

In the first part of the picture we see the Fixed Size Window extension. The player is in piece two, while the fixed size window (which contains four pieces) contains pieces three to six. The window includes non downloaded pieces, currently downloading and already downloaded pieces. This is the main difference from the other extensions, as we include in the window the pieces that have been downloaded.

In the second part of the picture we have the High Priority Set extension. With HPS we symbolize the window of this extension. As we describe in the HPS section, this window contains only pieces that are currently being downloaded or have still not been requested. The size of window is four and contains pieces three (currently downloaded), four, seven and eight.

In the third part of the picture we have the Stretching Window extension. In this extension the window contains (as in HPS) only pieces that are currently being downloaded or have still not been requested, but also the window can grow up to a bound in the piece sequence space (in this example the bound is five). The window now contains pieces three, four and seven.

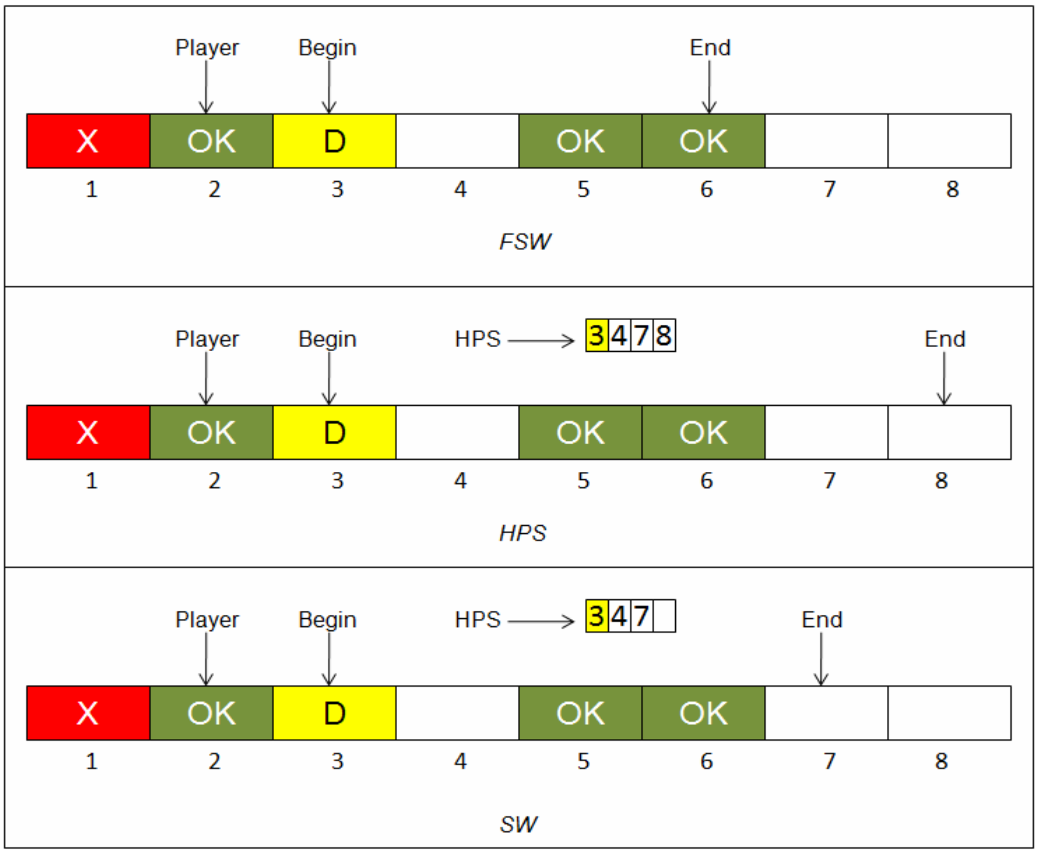

Figure 3: Example of Windowing Algorithms

This figure shows that HPS provide us with more pieces for download and for that reason we expect the diversity of pieces in HPS to be bigger than in the other two approaches. If we include the possibility that we have to request pieces outside the window, we are sure that we will have rare pieces to exchange later, avoiding to be choked by other peers (Tit-for-Tat policy). On the other hand, FSW is in the most difficult situation because piece four is currently being downloaded and piece three is ot requested yet. In the next step the FSW will only have one piece to request so in case that this piece is not available, we will waste available download bandwidth, increasing the possibility of choking. SW is an improvement as on average it has more pieces to request than FSW.

5. Streaming Player Modifications

In these extensions to BitTorrent, the player may reach a point where the next piece has not been downloaded yet. The simplest policy is to simply skip the piece, and move to the next one, displaying a blank screen for the duration of the piece, but this policy leads to a large degradation in quality. In this case, the metric of interest is the fraction of lost pieces. This policy was discussed in [20], so we do not include this metric in our experimental results. We describe below two other policies.

5.1. Stall Player

The first policy uses the same idea as web based streaming video players, such as YouTube. When the player does not have next the next piece it wants to play, it stalls until this piece is downloaded and is available to the player. Despite the fact that we have to wait until the missed piece is downloaded, we improve the quality of the video since we do not miss any piece. In addition, we do not miss pieces that may contain critical information for seeing and understanding the meaning of the video. In our approach, the buffer actually stalls until two pieces are downloaded, a modification that helps avoiding multiple stalls. In this approach we do not measure the missed pieces but the total time that each peer waits for missed pieces, until these pieces have been downloaded and become available for player, when video playback resumes.

5.2. Buffer player per block

In the second policy, instead of treating each piece as an indivisible unit of video, we examine the blocks in each piece and map them to video frames, so as to be able to playback those frames that have been received. While some of the blocks are being played, the application may even be able to download the remaining blocks of the piece. In contrast, in the original mode, if we have thirteen of fourteen blocks but not the last block of a piece, then the entire piece is characterized as missed, even though if the player started playing the available blocks, the last block may be available when the player wants it.

As in the original approach, the player is triggered by an event every x seconds and asks for the next block (contrary to the classic [20] per piece approach) to play. In case the player reaches a block that is not available, the block is characterized as missed, and possibly some other blocks will be characterized as missed, due to their reliance on the missed block, as explained below. The player stalls for the duration of the missed pieces. In this approach the critical metric is not the total missed pieces but the total missed blocks.

Since encoded video has a specific frame structure, with some frames depending on other frames to be decoded, in order to map video frames to blocks and calculate their dependencies and therefore the actual losses, we have assumed that the pieces are structured as follows:

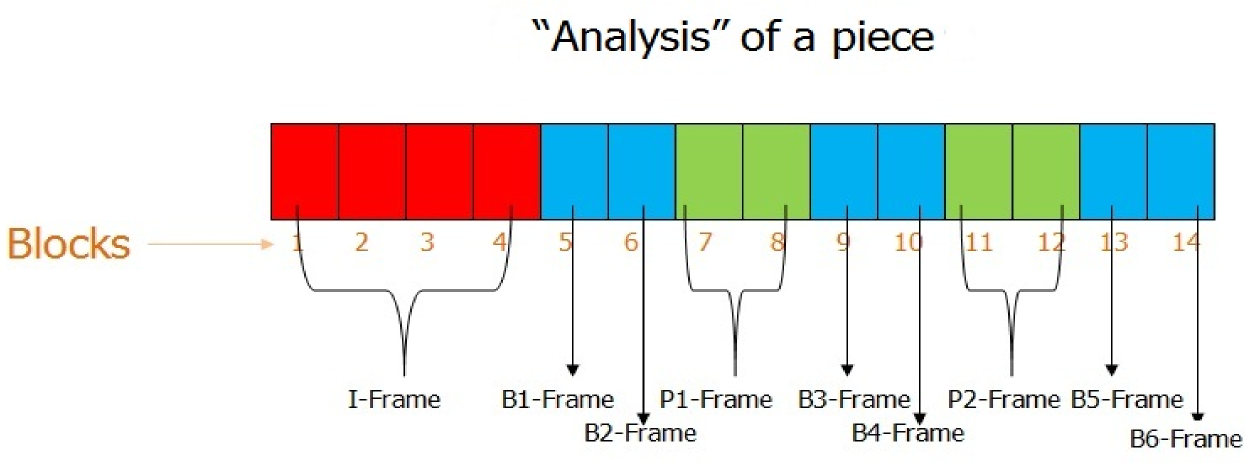

Figure 4: 'Analysis' of a piece

Each piece is an MPEG Group of Pictures (GOP), that is, it starts with an I-Frame, and ends right before the next I-Frame. P-Frames appear periodically between the I-Frames, each one depending on the previous P or I-Frame. Also, B-Frames appear between them, depending on the previous and next P or I-Frame.

The most important frame in a piece is the I-Frame, which occupies the first four sequential blocks (see Fig. 4). Losing even one block from this frames, causes the entire frame to be missed, even if some of its blocks have been downloaded, whole piece since I-Frame contains critical information for all the other frames of the piece until the next I-Frame of next piece. Less important than I-Frame are the P-Frames, covering two blocks each (see Fig. 4). In case we lose one P-Frame, we cannot decode the rest. Finally, the B-Frames are less important than the other frames, covering one block (see Fig. 4). If we lose any of them, we do not miss anything else, just that frame.

Actually, this is a simplified view of reality, as normally the P-Frames are transmitted before the B-Frames that depend on them, so as to allow decoding to follow reception rather than playback, and the final B-Frames also depend on the I-Frame in the next piece. In our model blocks of a piece are always downloaded in sequence, which means that whenever we find a missed block, it is impossible to have received any later blocks in that piece. As a result, whenever we encounter a missing block, we simply skip the entire remaining piece, stalling the player until the time comes to playback the next piece.

Despite the simplifying assumptions, by mapping blocks to frames, we are able to calculate not only how many blocks were missed, but also how many and which types of frames were lost, providing a more accurate estimation of video quality that what is possible by relying solely on block-based and piece-based metrics.

6. Experimental Setup

6.1. BitTorrent Simulator

In our experiments we modified our own detailed OMNeT++ Simulator for BitTorrent, 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 BitTorent peers and the tracker, it also models all the TCP/IP messages exchanged, using the INET framework [21], 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, including both core and access routers. Each scenario was executed 10 times with different random seeds, thus deploying peers in different ways in the network.

6.2. Experiments

Our results are based on a scenario with one initial seeder and 120 peers (leechers) in a swarm, 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. Peers access links have asymmetric uplink and downlink bandwidths: 20% have ¼ Mbps capacity (uplink/downlink), 40% have 1/8 Mbps, 25% have 2/12 Mbps and 15% have 2/24 Mbps.

The streaming application model was a video player attempting to playback a 256 Kbps video stream. We assumed that the video was encoded in MPEG like manner, where each group of Pictures (GOP) was mapped to exactly one BitTorrent piece. We set the GOP/piece size to 224 KB, which translates to 7 seconds of video. In this manner, a piece that has not been downloaded on time would cause a 7 second loss in the video in the original player mode, but it will not prevent the next piece from decoding. In the mode where the player stalls for missing pieces, a piece that has not been downloaded on time will cause the player to wait for some seconds until the piece downloaded and be available for the buffer. In the mode where the player examines the type of block/frame missing, a block that has not been downloaded on time will cause one or more 0.5 second losses in the video, depending on its position in the piece. The block size was set to 16 KB. The entire video size was 200 MB, which corresponds to around 106 minutes of playtime, or 914 pieces in total.

We kept all the default BitTorrent settings unchanged e.g. optimistic unchoke interval, number of connections per peer, as given in [3]. To make our model relevant to live streaming, we assumed that each peer will remain 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 competition. 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.

Parameter	***Value***
Video size (in MB)	200
Video Bit rate (in Kbps)	128
Piece size (in KB)	224
Block size (in KB)	16
Num of pieces for prefetch buffering	1 or 5
Window size (% of total num of pieces)	2% or 8%
Probability p (only for HPS)	80% or 90%
Bound (only for SW)	30 or 100
Player Mode	1 or 2
Num of Pieces in group (only for HPS)	25% (of window size)

Table II: Simulation Parameters

6.3. Simulation Parameters

We assumed that each peer starts by prefetching a few initial piece (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 but only for the 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 (914) for all algorithms, which in our case translates to 18 or 73 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 = 18 and to 100 pieces for k = 73, therefore SW can grow its window more than FSW, but not as much as HPS. For the HPS modifications, the number of pieces that the group can contain is 25% of the window size (HPS size), that is, the window is divided into four groups. Table II summarizes these parameters.

As explained above, the player uses three modes to deal with missing pieces. In Mode 1 it plays back the piece in terms of blocks (frames) skipping missed blocks (frames), in Mode 2 it stalls until it downloads the entire piece, and in Mode 3 it skips the entire piece (our results do not use this mode).

6.4. Metrics

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

- 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 show how much time the entire download

took place; it is interesting to compare this with the (fixed) playing time.

- Wait time: The total time that the player stalls, waiting for some pieces/blocks that it does not have, so it can download them and the player can continue. This is valid only in player Mode 1.
- Block loss: A block is characterized as lost if the player reaches it and the block is not downloaded, leading to a gap in the video; if the missed block is not the last of the piece then all the above blocks of the piece are characterized as missed. This is valid only in player Mode 2.
- Frame loss: Each block that we lose corresponds to a frame of the piece. It is important for us to know what kind of frames we have lost, so we can understand how good the quality of the video was. This is valid only in player Mode 2.

7. Experimental Results

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

7.1. Wait time (Player mode 1)

For the variant of the player that stalls on misses (Mode 1), in Fig. 5 and Fig. 6 we show the waiting for each peer on average, with the window set to 2% of the video size. With more prefetched pieces the performance slightly improves with all protocols; the penalty is an increased delay before starting playback, as we will see in the next section. While all protocols exhibit acceptable performance, FSW works best in both cases. In all HPS variants (HPS is the classic mode, BP is the per piece probability and BG the per group probability), the higher probability to download from within the window works better; among the HPS variants, the one that assigns different probabilities to each piece works best. 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.

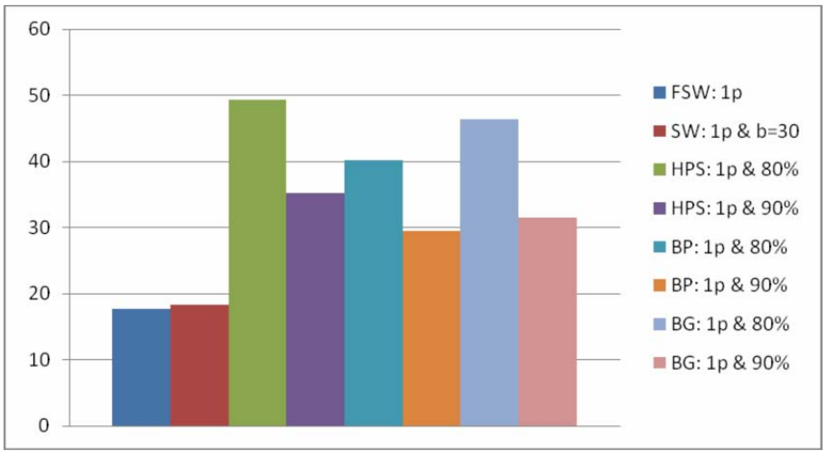

Figure 5 – Wait time for 2% window size and 1 piece prefetch

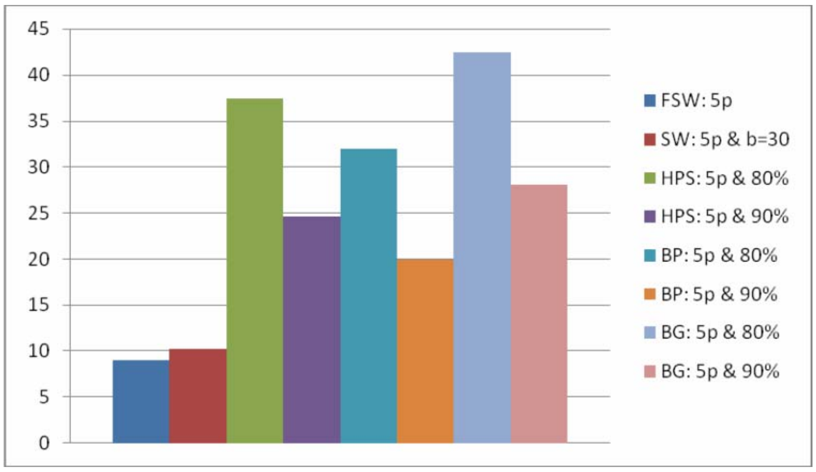

Figure 6 – Wait time for 2% window size and 5 pieces prefetch

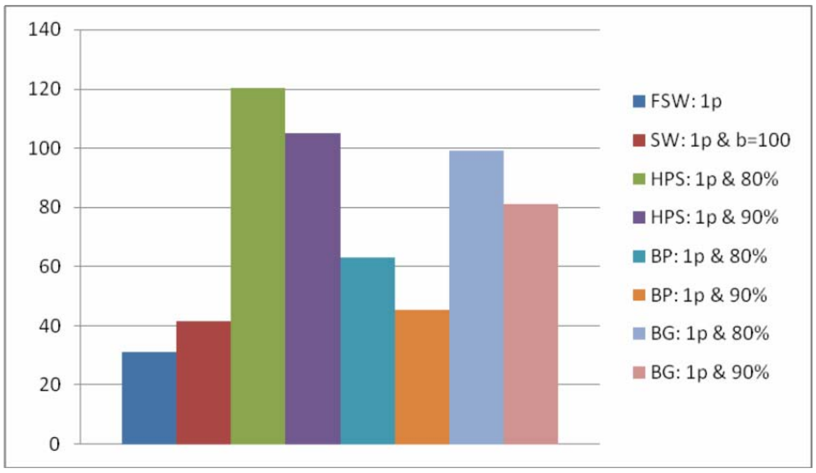

Figure 7 – Wait time for 8% window size and 1 piece prefetch

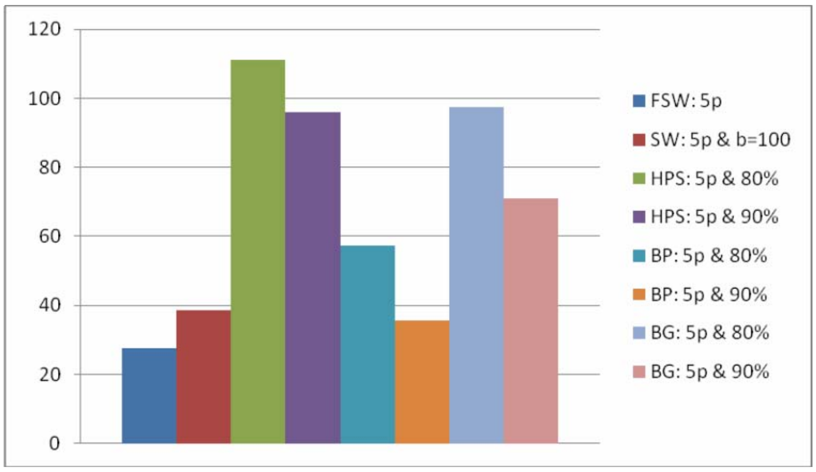

Figure 8 – Wait time for 8% window size and 5 pieces prefetch

Fig. 7 and Fig. 8 shows the corresponding data for a window size of 8%. As we can see a higher number of prefetched pieces has a beneficial but generally small effect. FSW is still the best performer, closely followed by SW. Among the HPS variants, again the one that differentiates probabilities per piece works best, and with a better margin than before, and generally the higher the probability to download from within the window, the better.

Looking at both scenarios, we can see that we have much better results with a 2% window. With a 2% window (13 pieces) all methods concentrate on the immediate pieces to be played next, while with a 8% window (73 pieces), the player has to stall more often. Our modifications to HPS work better than the classic approach of BiToS in terms of waiting time, especially the modification that uses a different probability for every piece. In both scenarios the HPS variants work better when they are less likely to download pieces outside the window.

7.2. Block loss (Player mode 2)

For the variant of the player that drops missed blocks (Mode 2), in Fig. 9 and Fig. 10 we show the block loss for each protocol, with the window set to 2% of the video size. As in player Mode 1, with more prefetched pieces the performance slightly improves with all protocols, with the disadvantage of an increased delay before starting playback. The HPS variants work better with a higher probability to download within the window; in this case, the original HPS works best, followed by SW, BP, FSW, and BG. In all cases however the losses are less than 0.5%, therefore the differences are very small.

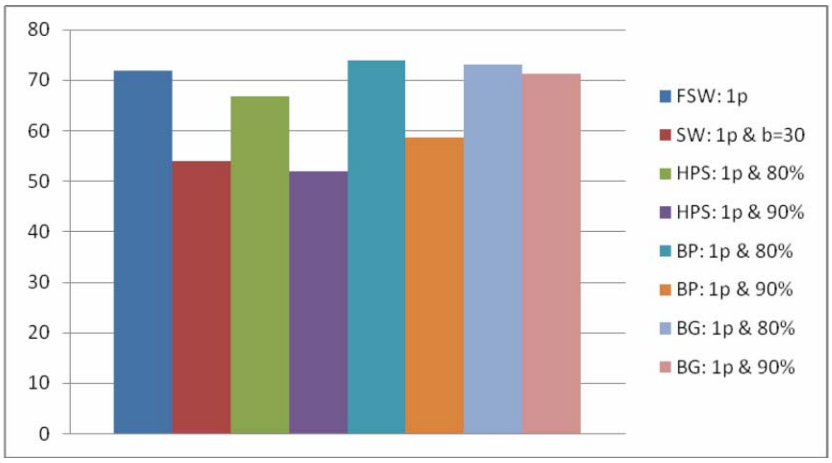

Figure 9 – Block loss for 2% window size and 1 piece prefetch

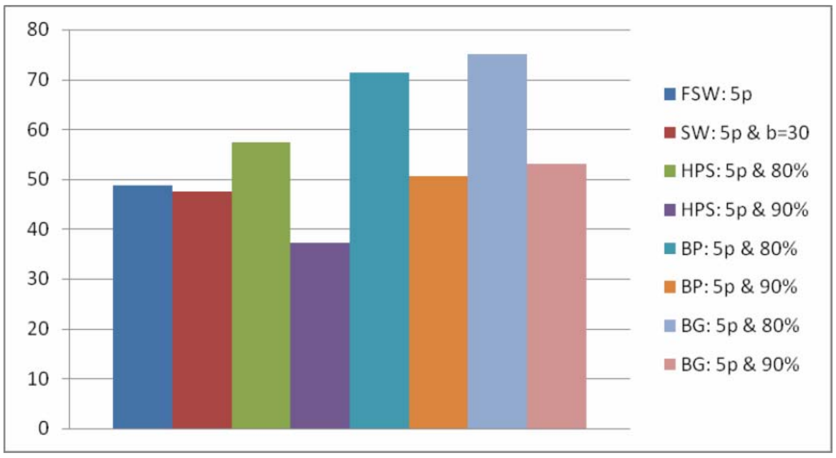

Figure 10 – Block loss for 2% window size and 5 pieces prefetch

Fig. 11 and Fig. 12 shows the corresponding data for a window size of 8%. In this case see the HPS variants become the worst performers, while FSW and SW actually work slightly better than in the previous case. The modified variants of HPS (BP and BG) work better than the original. In this case losses range from less that 0.5% to up to 0.85%, which is still quite small, but more noticeable.

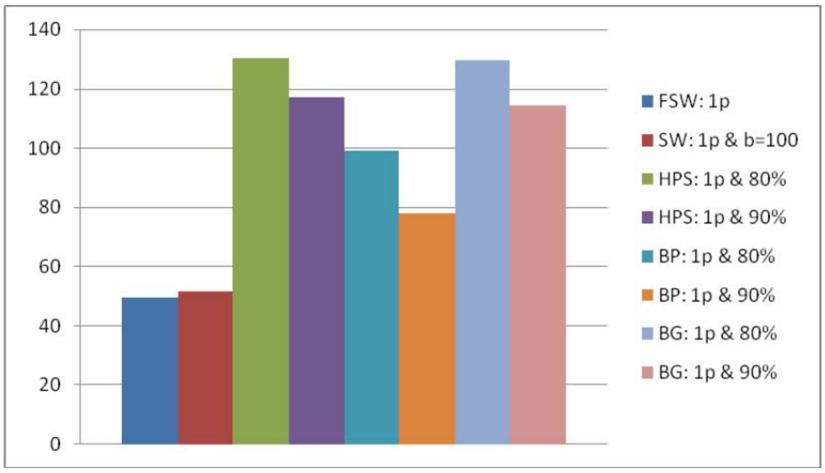

Figure 11 – Block loss for 8% window size and 1 piece prefetch

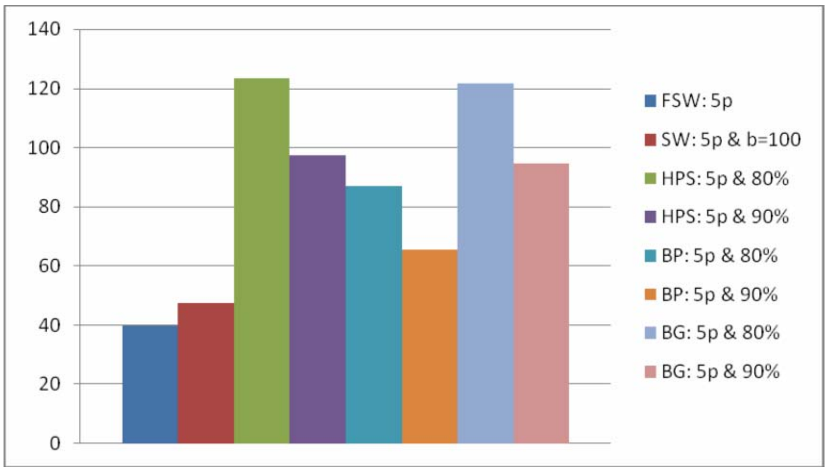

Figure 12 – Block loss for 8% window size and 5 pieces prefetch

Looking at both scenarios, it is clear that the window size is an important parameter. With a 2% window (13 pieces), FSW and SW actually work slightly worse than HPS which can grow its window more, and ask for pieces outside the window. With a 8% window however, FSW and SW have enough pieces to download; there is no need to ask for pieces outside the window as HPS does, or grow the window too much. In both scenarios HPS and our modifications works better when it is less likely to download pieces outside the window.

7.3. Frame loss (Player mode 2)

Here we delve deeper into the analysis of block loss for player Mode 2. As we said in the previous section, each piece is divided in blocks each one corresponding to a specific frame. In Fig. 13 and Fig. 14 we presented the frame loss for each protocol, with the window set to 2% of the video size. The most frames lost were B-Frames, which are less important than the other frames and influence lightly the quality of the video. In second place we have P-Frames which have some critical information for the other P-Frames of the piece and for the next piece. The most important frames, I-Frames, as we can see have the smaller percentage in loss frames, but in case of losing one I-Frame we lose the entire piece. The same metrics for a window set to 8% of the video size are shown in Fig. 15 and Fig.16.

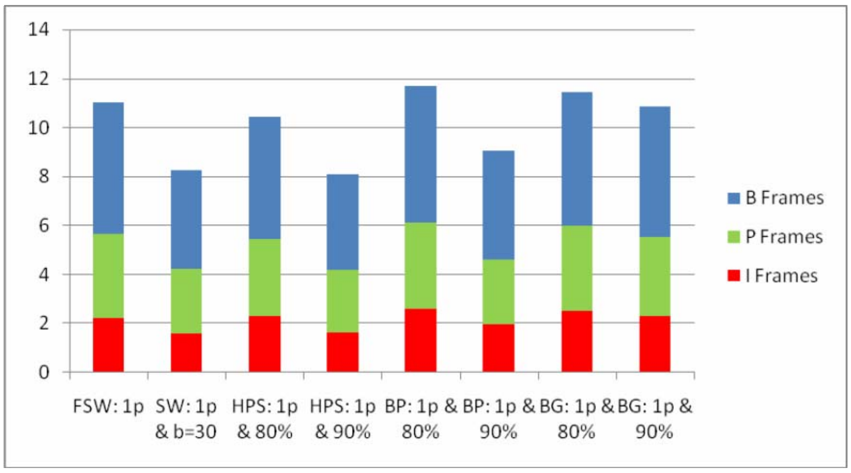

Figure 13 – Frame loss for 2% window size and 1 piece prefetch

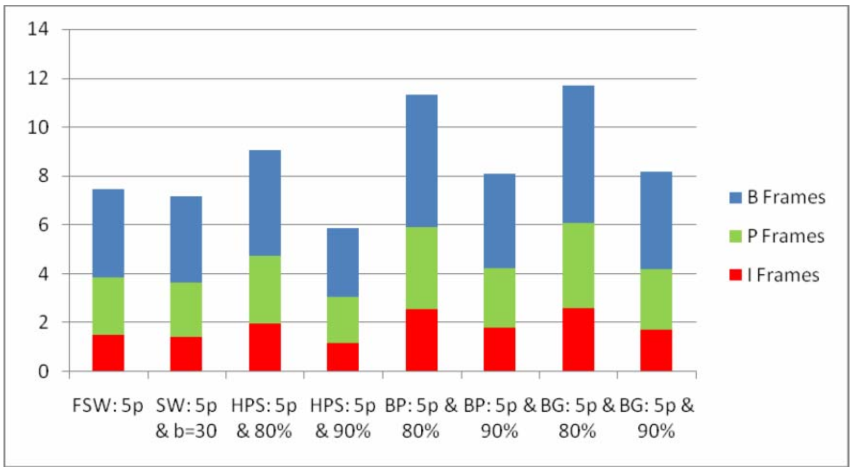

Figure 14 – Wait time for 2% window size and 5 pieces prefetch

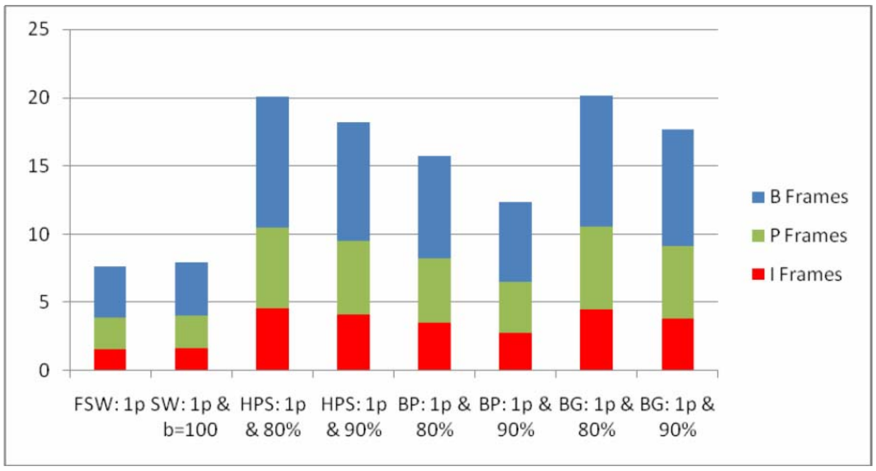

Figure 15 – Frame loss for 8% window size and 1 piece prefetch

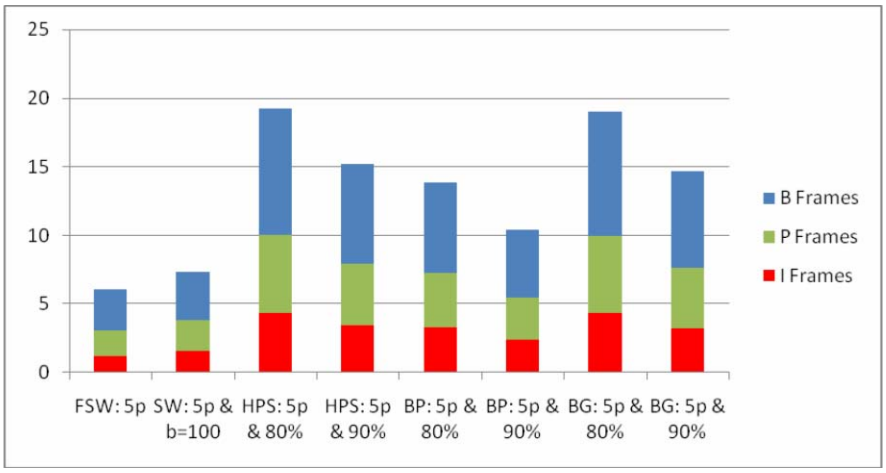

Figure 16 – Wait time for 8% window size and 5 pieces prefetch

As we said in the previous section, each time we lose an I-Frame or a P-Frame we lose all the remaining frames in the piece. But the relationship between frames and blocks is complex, since I-Frames and B-Frames consist of multiple blocks (4 and 2 respectively). However, the general behaviour follows the previous metric (block loss), in the sense that the HPS variants are slightly better for a 2% window, but FSW and SW work better for an 8% window.

7.4. Download duration

As we show in Fig. 17, Fig. 18, Fig. 19 and Fig. 20, the required time to complete the video is lowest for HPS, slightly more for our HPS modifications and higher for FSW and SW. While FSW and SW are nearly identical, in HPS and our modifications it seems that a higher probability to download within the window leads to worse download time (recall that it also led to lower loss rates in mode 2, so there is a trade-off here). Since the playback time is 6400 sec, it is clear that all schemes manage to complete the download well before the playback begins. Note also that with a larger window, downloads complete even faster. Since clients remain in the swarm until their player reaches the end of video, 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 the observed losses are mostly due to these first peers. Finally, the difference between the two player modes in tiny.

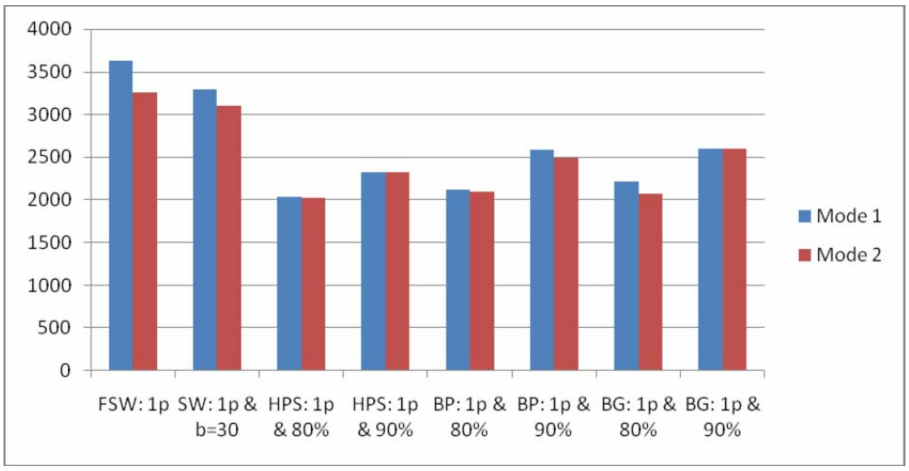

Figure 17 – Download time for 2% window size and 1 piece prefetch

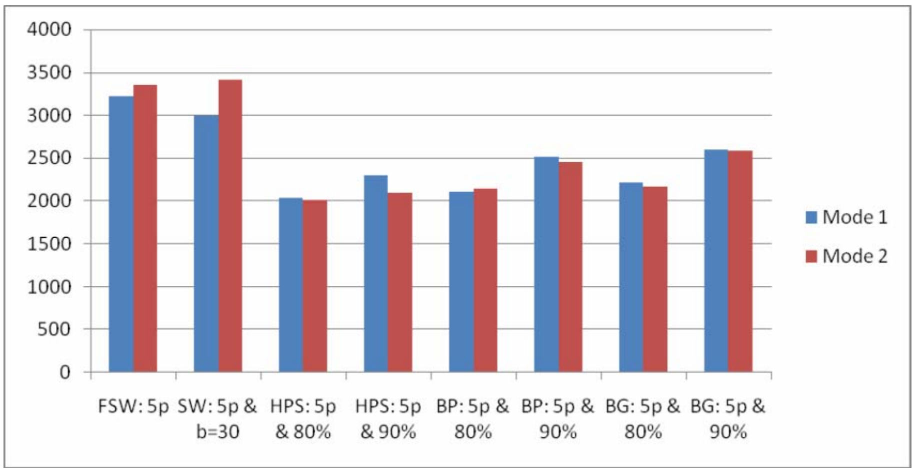

Figure 18 – Download time for 2% window size and 5 pieces prefetch

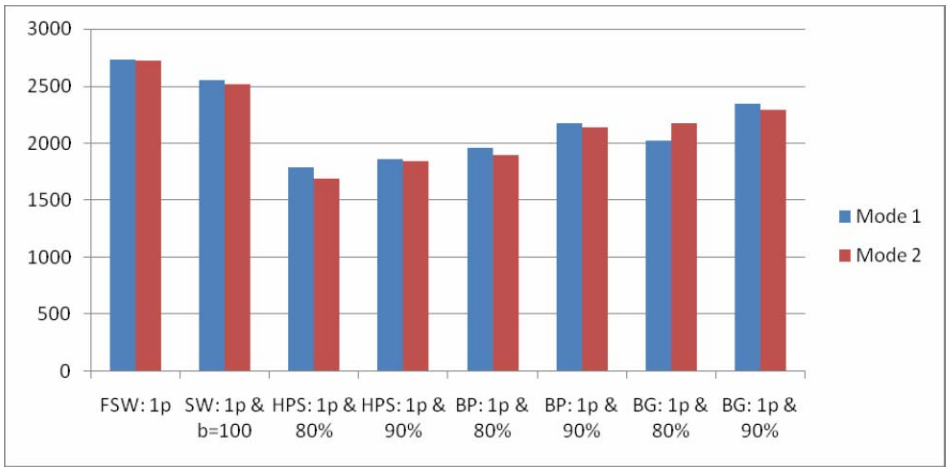

Figure 19 – Download time for 8% window size and 1 piece prefetch

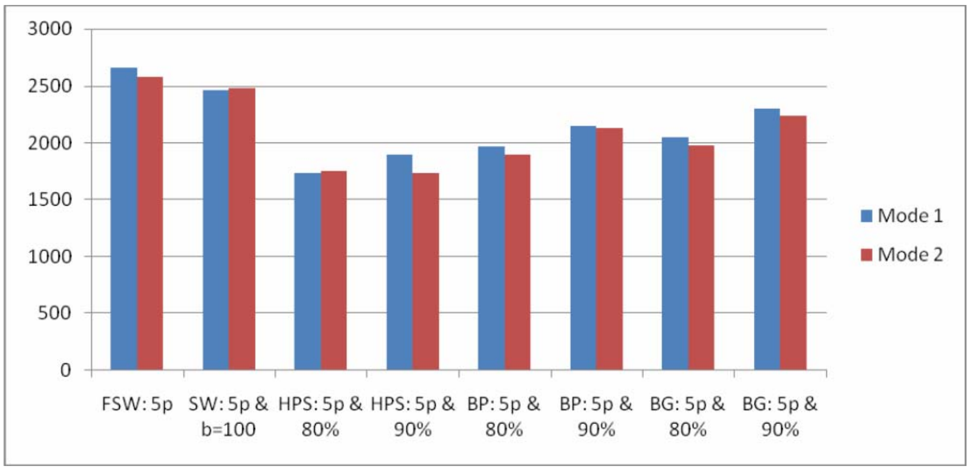

Figure 20 – Download time for 8% window size and 5 pieces prefetch

7.5. Prefetch time

The experiments show that the buffering time at video beginning fluctuates between 21 and 35 seconds (average values), depending on the number of pieces we have chosen to download; Fig. 21 and Fig. 22 shows the data for a 2% window size, while

Fig. 23 and Fig. 24 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 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 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 the differences between the various protocols regarding their prefetch delay time before starting are very small in absolute terms, since the protocol modifications have not started operating yet.

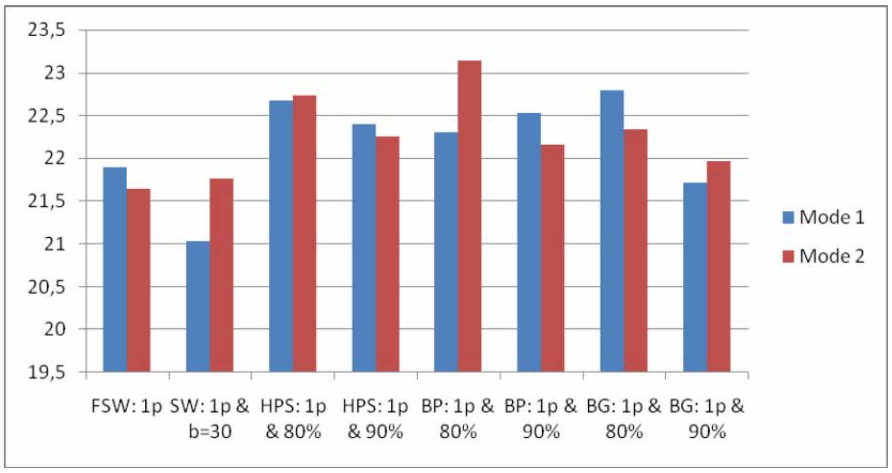

Figure 21 – Prefetch time for 2% window size (1 piece prefetch)

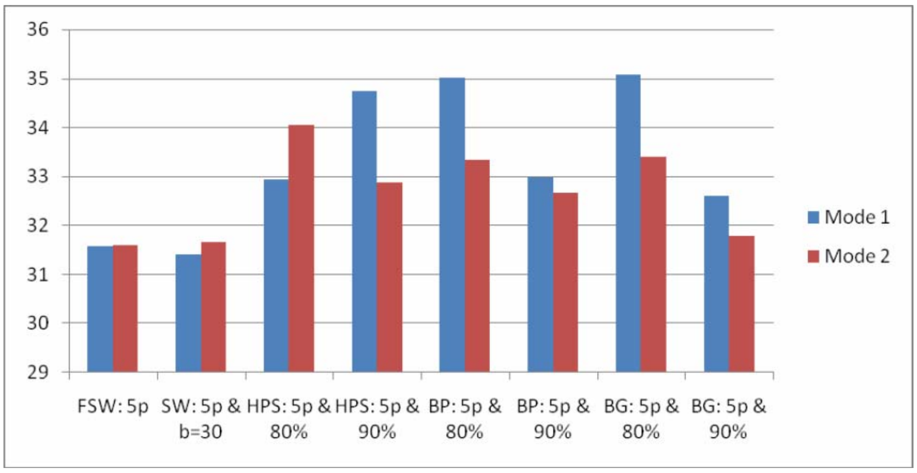

Figure 22 – Prefetch time for 2% window size (5 pieces prefetch)

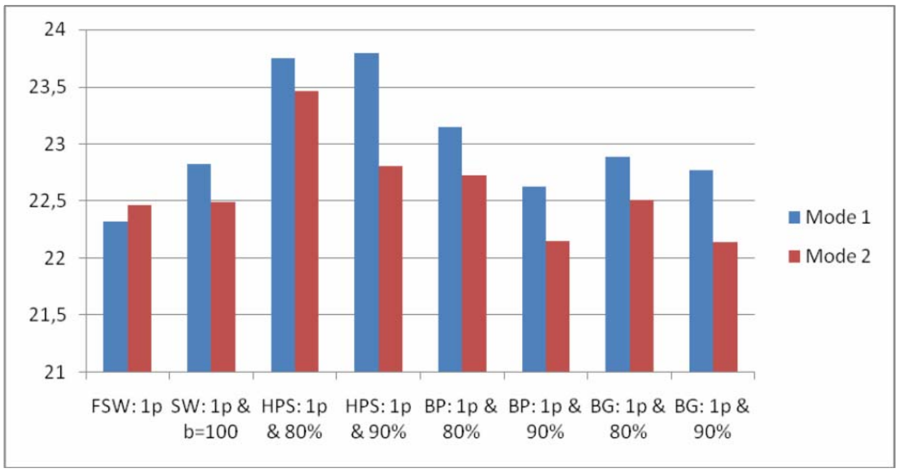

Figure 23 – Prefetch time for 8% window size (1 piece prefetch)

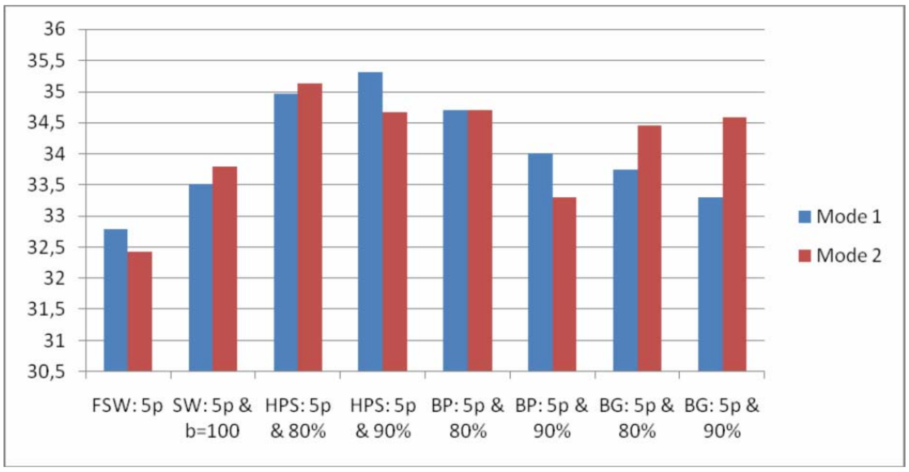

Figure 24 – Prefetch time for 8% window size (5 pieces prefetch)

8. Conclusions

In this thesis we show that with minimal changes BitTorrent can support video streaming. We have studied three piece selection strategies (plus two modifications we made in one of these) in two different player modes, leading to the following conclusions:

- FSW exhibits lower wait times than SW, HPS and its modifications, both with a 2%, as well as an 8% window size.
- FSW exhibits lower block loss rates than SW, HPS and its modifications for larger windows, but with smaller windows HPS works better.
- Prefetching does lower the block loss rates and the wait time 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 is not as important as a reduced loss rate or wait time.
- FSW and SW perform nearly identically, therefore the extra complexity of SW does not seem to be worthwhile.

One other thing that we must say is that one of our modifications (BiToS with probability in each piece inside the window – BP) works better than the classic approach of HPS (BiToS) in most cases. The only disadvantage with this modification is that it does not use the policy of Tit-For-Tat of classic BitTorrent when we need to select a piece inside the window.

<u>9. Future work</u>

9.1. Adaptation of HPS probability

The adaptation of the probability p which reflects a balance between asking for pieces inside and outside the HPS, can be triggered by events, such as a deadline miss. For example, a miss of a piece’s deadline while there are many pieces unplayed inside the Received Pieces Buffer indicates that the probability p should be increased in order to give higher priority to the pieces that have shorter deadlines. On the other hand, if we miss many deadlines and there are no other received pieces and the download rate is small, this could indicate that the peer is chocked by most of its peers, because it does not have pieces to exchange. Therefore, the decrease of the value of the probability can be helpful in order to acquire some rare pieces that the peer can use as leverage.

9.2. Download missed blocks

Missing blocks is the most crucial affecting streaming performance. We could reduce loss rate by trying to download the missed blocks that we have, until the player completes the playback of a partially downloaded piece, and seeding to other peers, so we can increase the diversity of the pieces and through this the total overall system efficiency.

9.3. Improve block to frame mapping

The current mapping of video frames to blocks is not very accurate. We need to first modify the transmission order of frames to reflect actual practice, that is, first transmit the P-Frames and then the B-Frames depending on them. In this manner, and assuming that blocks are still being downloaded in sequence, whenever we encounter a missing block, we can safely mark the next blocks in the piece as missed and correctly assume that earlier blocks correspond to frames that can actually be played back.

10. References

[1] Wikipedia – BitTorrent

[2] BitTorrent Still King of P2P Traffic

[3] K. Katsaros, V. P. Kemerlis, C. Stais and G. Xylomenos, "A BitTorrent Module for the OMNeT++ Simulator," Proc. 17th Annual Meeting of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), London, Great Britain, September 2009

[4] A. Vlavianos, M. Iliofotou, and M. Faloutsos, "BiToS: Enhancing BitTorrent for Supporting Streaming Applications", in Global Internet Workshop in conjunction with IEEE INFOCOM 2006, April 2006

[5] Streaming Video, By Damian Sofsian

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

[7] P. Shah and J.-F. Pâris, "Peer-to-peer multimedia streaming using BitTorrent," in Proceedings of the 26th IEEE International Performance, Computing, and Communications Conference (IPCCC '07), pp. 340–347, New Orleans, La, USA, April 2007.

[8] R. LaFortune, C. D. Carothers, W. D. Simth, J. Czechowski and X. Wang, "Simulating Large-Scale P2P Assisted Video Streaming", In Proceedings of the Hawaii International Conference on System Sciences (HICSS-42), Waikoloa, Big Island, Hawaii, January 2009.

[9] BitTorrent Development Community. BitTorrent Protocol Specification v1.0 http://wiki.theory.org/BitTorrentSpecification

[10] András Varga OMNeT++ network simulator homepage http://www.omnetpp.org

[11] A. Legout, G. Urvoy-Keller and P. Michiardi, "Rarest first and choke algorithms are enough", In Proc. ACM SIGCOMM'06.

[12] T. Berners-Lee, L. Masinter, and M. M. (eds). Uniform Resource Locators (URL). Internet Request for Comments, December 1994. RFC 1738.

[13] A. Bharambe, C. Herley, and V. Padmanabhan, "Analyzing and improving BitTorrent performance", Technical Report MSR-TR-2005-03, Microsoft Research, 2005.

[14] J. Pouwelse, P. Garbacki, D. Epema, H. Sips, "The Bittorrent P2P File-Sharing System: Measurements and Analysis", In IPTPS 2005, Ithaca, USA, February 2005.

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

[16] D. Clark, B. Lehr, S. Bauer, P. Faratin, R. Sami, and J. Wroclawsk, "Overlay networks and the future of the internet," Communication & Strategies, vol. 63, 2006.

[17] Cohen, Bram (October 2002). "BitTorrent Protocol 1.0". BitTorrent.org. Retrieved 2008-10-27.

[18] Tamilmani, Karthik (2003-10-25). "Studying and enhancing the BitTorrent protocol". Stony Brook University. Retrieved 2006-05-06.

[19] Unraveling BitTorrent's File Unavailability:Measurements and Analysis by Sebastian Kaune, Ruben Cuevas Rumin, Gareth Tyson, Andreas Mauthe, Ralf

Steinmetz

[20] C. Stais, G. Xylomenos, A. Archodovassilis, "A comparison of streaming extensions to BitTorrent", Proc. of the IEEE ISCC 2011, Corfu, Greece 2011.

[21] I. Baumgart, B. Heep, and S. Krause, "OverSim: A flexible overlay network simulation framework," in Proc. of the IEEE Global Internet Symposium, Anchorage, AK, USA, Jan 2007, pp. 79–84.