Master in Artificial Intelligence (UPC-URV-UB)

Master of Science Thesis

Cognitive networking techniques on content distribution networks

Carles Anglés Tafalla

Advisor/s: Pedro García López

5th September 2012

Table of Contents

1. Introduction

Peer-to-peer (P2P) systems use decentralization to allow a huge range of scalable and fault-tolerant services like data-sharing, video streaming and voice-over-IP. In this kind of systems each member (called peer) can communicate directly with any other without the inference of a central server. This means that each peer becomes a content server as well as a client.

P2P networks have emerged as an important architecture for content sharing over the Internet. Nowadays reports indicate that P2P traffic still represents about 45% of the Internet European traffic [1]. In 2006 some studies estimated that up to three quarters of all the bandwidth on the Internet was consumed by P2P traffic. At the end 2010 this bandwidth proportion was down to 39% according to [2], and by 2014 it is expected to go down even further to 17%.

This doesn't mean that there is a decline in interest for P2P or BitTorrent though, as the absolute traffic numbers are expected to rise year after year doubling to 7 Petabytes per month in the next four years [2]. However, as the rest of the total Internet traffic grows even stronger because of Youtube and other streaming sites, P2P traffic as a percentage of all Internet traffic will fall. However, last legal actions of U.S.A government against file sharing sites like Megaupload have resulted in an increase of P2P traffic percentage, fixing it in a 45%.

Nowadays, P2P networks support a wide variety of applications like audio conferencing (e.g. Skype), file sharing (e.g. Bittorrent, Emule) or video streaming (e.g. TVUPlayer, Joost, UStream). Focusing on the file sharing practice, BitTorrent has achieved a great success in the field of P2P file sharing. According to IPOQUE's reports released in 2009 Bittorrent accounts for approximately 32% of Southwestern European P2P traffic [3]. This value reaches values up to 57% in Eastern Europe regions.

BitTorrent (BT) is a protocol that organizes a group of users (peers) sharing the same file into a P2P network and focuses on fast and efficient replication. In Bittorrent systems, shared contents are divided into small pieces. These pieces are downloaded and uploaded between peers interested in the same content. Peers are stimulated to share their acquired pieces thanks to a tit-for-tat incentive mechanism, which reward the peers who are giving more with more downloading capacity. Rarest first strategy ensures that peers will download the pieces whose neighbors have fewest, reducing the probability of certain peer having nothing of interest.

The popularity acquired by BitTorrent has attracted the interest of the research community. This interest has propitiated the appearance of studies and works proposing all sorts of strategies to improve BT performance. Studies in [4] remark that rarest first and choke algorithms have been widely studied and optimized. Despite the diversity of these works, many of them focus their efforts on achieving a greater knowledge of global state of the swarm (Global knowledge). Having a greater global knowledge means a greater knowing of the current situation of each peer and a higher understanding of the swarm global state. This information is a great advantage when enhancing the BitTorrent algorithms. A refined model of BitTorrent using global knowledge is proposed in [5] and its high efficiency is showed.

However, assuming a total global knowledge in a BitTorrent scenario is not realistic. In real implementations peers have very limited information about the rest of the swarm and global knowledge should be inferred from the information that each peer has locally. However, there are many BT clients who take advantage of these strategies to improve their download performance by trading selfishly with the rest of the peers. The major drawback of these kinds of approaches is that the rest of the swarm members are negatively affected. Three different approaches of selfish-peer exploits are designed and implemented in [6], authors conclude that in many torrent scenarios a selfish peer could benefit more significantly at the expense of honest peers.

According to the previously mentioned problems of swarm global knowledge difficulty acquisition and the common selfish orientation of BT strategies, we have two major challenges.

First we want to design a strategy based on Artificial Intelligence (AI) techniques with the aim of increasing peers download performance. Some AI algorithms can find patterns in the information available to a peer locally, and use it to predict values that cannot be calculated by means of mathematical formulas. An important aspect of these techniques is that can be trained in order to improve its interpretation of the local available information. With this process they can make more accurate predictions and perform better results. We will use this prediction system to increase our knowledge about the swarm and the peers who are part of it. This global knowledge increase can be used to optimize the algorithms of BitTorrent and can represent a great improvement in peers download capacity.

Our second challenge is to create a reduced group of peers (Crowd) that focus their efforts on improving the condition of the swarm through collaborative techniques. The basic idea of this approach is to organize a group of peers to act as a single node and focus them on getting all pieces of the content they are interested in. This involves avoiding, as far as possible, to download pieces that any of the members already have. The main goal of this technique consists of reaching as quickly as possible a copy of the content distributed between all members of the Crowd. Getting a distributed copy of the content is expected to increase the availability of parts and reduce dependence on the seeds (users who have the complete content), which would represent a great benefit for the whole swarm. Another aspect that we want to investigate is the use of a priority system among members of the Crowd. We consider that in certain situations to prioritize the Crowd peers at expense of regular peers can result in a significant increase of the download ratio.

Finally, we outline that our contribution is simple to implement in real settings. Our modifications do not require changing the BitTorrent messaging protocol. Thus, our modified implementation can be deployed seamlessly. We hope that this work results can lead to new ways of optimizing content distribution using the BitTorrent protocol.

2. Background and related work

In this work, we will study the optimization and improvement of BitTorrent protocol algorithms. Our work is focused on artificial intelligence techniques which increase the overall performance of the protocol. In order to understand how these optimizations have been built we need some understanding about what the Bittorrent protocol defines and how its algorithms operate. In further sections we provide some background about BitTorrent protocol and the last research works that have been conducted on it.

2.1. BitTorrent Protocol

BitTorrent is a peer-to-peer file sharing protocol used for distributing large amounts of data designed by the programmer Bram Cohen in 2001. BitTorrent is one of the most common protocols for transferring large files, and it has been estimated that it represents around the 27–55% of all Internet traffic (depending on geographical location) [7].

2.1.1. Description

BitTorrent protocol allows users to distribute large amounts of data without the heavy demands on their computers that would be needed for standard Internet hosting. A standard host server can be easily brought to a denial of service state if it receives high levels of simultaneous data flow. The protocol works as an alternative data distribution method that makes even small computers (e.g. mobile phones) with low bandwidth capable of participating in large data transfers.

First, a user playing the role of file-provider makes a file available to the network. This first user's file is called a *seed* and its availability on the network allows other users, called *peers*, to connect and begin to download the seed file. As new peers connect to the network and request the same file, they receive different pieces of the data from the seed. Once multiple peers have multiple pieces of the seed, BitTorrent allows each of them to become a source for that portion of the file. With this each peer take on a small part of the task and relieve the initial user, distributing the file download task among the seed and many peers.

After the file is successfully and completely downloaded by a given peer, this peer is able to shift roles and become an additional seed, helping the remaining peers to receive the entire file. This eventual shift from peers to seeders determines the overall “health” of the file (as determined by the number of times a file is available in its complete form).

This distributed nature of BitTorrent leads to a flood like spreading of a file throughout peers. As more peers join the swarm (swarm are all seeds and peers that are connected together), the likelihood of a successful download increases. This provides a significant reduction in seed’s hardware and bandwidth resource costs. It also provides redundancy against system problems and reduces dependence on the original distributor.

The BitTorrent protocol can be split into the following five main components:

- **Data:** The files being transferred across the protocol.
- **Metainfo File:** a file which contains all details necessary for the protocol to operate.
- **Tracker:** A server which helps to manage the BitTorrent protocol.

- **Peers:** Users exchanging data via the BitTorrent protocol.
- **Client:** The program which sits on a peer's computer and implements the protocol.

2.1.2. Data

BitTorrent is very versatile, and can be used to transfer a single file, of multiple files of any type, contained within any number of directories. File sizes can vary hugely, from kilobytes to hundreds of gigabytes.

2.1.2.1. Piece Size

Data is split into smaller pieces which sent between peers using the BitTorrent protocol. These pieces are of a fixed size, which enables the tracker to keep tabs on who has which pieces of data. This also breaks the file into verifiable pieces, each piece can then be assigned a hash code, which can be checked by the downloader for data integrity. These hashes are stored as part of the "metainfo file" which is discussed in the metainfo section (2.1.3).

The size of the pieces remains constant throughout all files in the torrent except for the final piece which is irregular. The piece size of a torrent depends on the amount of data. Piece sizes which are too large will cause inefficiency when downloading (larger risk of data corruption in larger pieces due to fewer integrity checks), whereas if the piece sizes are too small, more hash checks will be needed.

As the number of pieces increase, more hash codes need to be stored in the metainfo file. Therefore, as a rule of thumb, pieces should be selected so that the metainfo file is no larger than 50 – 75kb. The main reason for this is to limit the amount of hosting storage and bandwidth needed by indexing servers. The most common piece sizes are 256kb, 512kb and 1mb. The number of pieces is therefore: total length / piece size. Pieces may overlap file boundaries. For example, a 1.4Mb file could be split into the following pieces:

5 * 256kb pieces, and a final piece (p6) of 120kb.

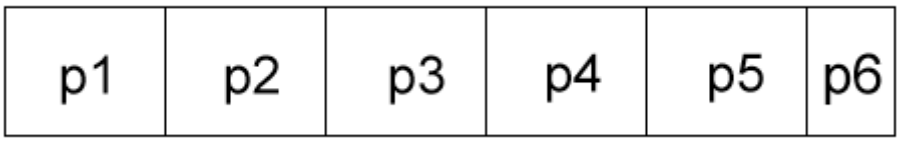

Figure 2-1. A file split into 6 pieces.

2.1.3. Metainfo File

When someone wants to publish data using the BitTorrent protocol, they must create a metainfo file. This file is specific to the data they are publishing, and contains all the information about a torrent, such as the data to be included, and IP address of the tracker to connect to. A tracker is a server which "manages" a torrent, and is discussed in the next section (2.1.4). The file is given a ".torrent" extension, and the data is extracted from the file by a BitTorrent client (a BitTorrent client is a program which runs on the user computer, and implements the BitTorrent protocol). Every metainfo file must contain the following information, (or "keys"):

- **info**: A dictionary which describes the file(s) of the torrent. Hashes for every data piece, in SHA 1 format are stored here.
- **announce**: The announce URL of the tracker as a string.

The following are optional keys which can also be used:

- **announce-list**: Used to list backup trackers creation date: The creation time of the torrent by way of UNIX time stamp.
- **comment**: Any comments by the author.
- **created** by: Name and Version of the program used to create the metainfo file.

Thanks to the fact that all information which is needed for the torrent is included in a single file, this file can be easily distributed via other protocols. The most popular method of distribution is using a public indexing site which hosts the metainfo files. A seed will upload the file, and then others can download a copy of the file over the HTTP protocol and participate in the torrent.

2.1.4. Tracker

A tracker is used to manage users participating in a torrent (know as peers). It stored statistics about the torrent, but its main role is to allow peers to "find each other" and start communication, i.e. to find peers with the data they require. Peers know nothing of each other until a response is received from the tracker. Whenever a peer contacts the tracker, it reports which pieces of a file they have. That way, when another peer queries the tracker, it can provide a random list of peers who are participating in the torrent, and have the required piece.

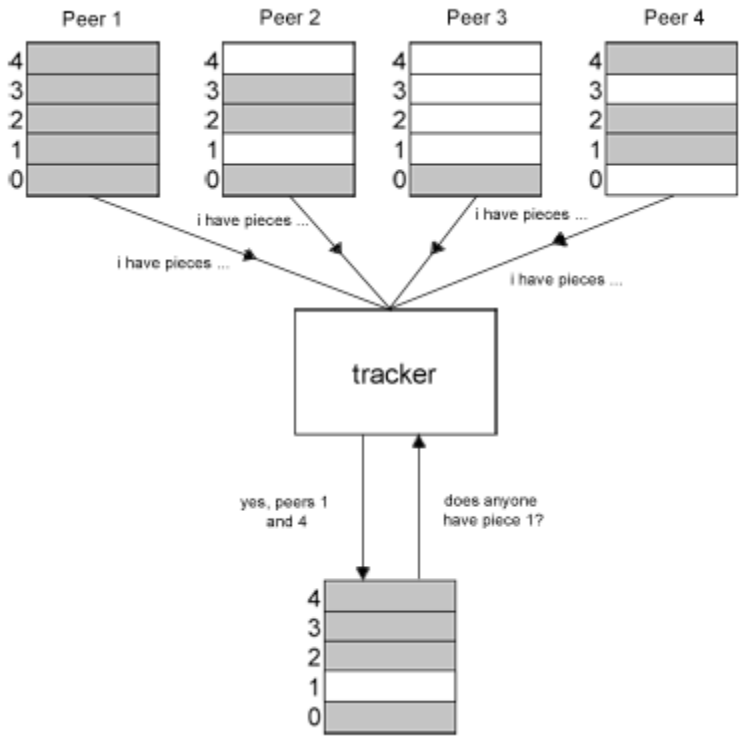

Figure 2-2. Tracker providing a list of peers with the required data.

A tracker is a HTTP/HTTPS service and typically works on port 6969. The address of the tracker managing a torrent is specified in the metainfo file, a single tracker can manage multiple torrents. Multiple trackers can also be specified, as backups, which are handled by the BitTorrent client running on a user's computer.

2.1.5. Peers

Peers are other users participating in a torrent, and have the partial file (known as leechers), or the complete file (known as a seed). Pieces are requested from peers, but depending on the status of the peer, are not guaranteed to be sent.

2.1.5.1. Piece Selection Algorithm

Which piece is requested depends upon the BitTorrent client. There are three stages in the piece selection algorithm, which change depending on which stage of completion a peer is at (Figure 2-3).

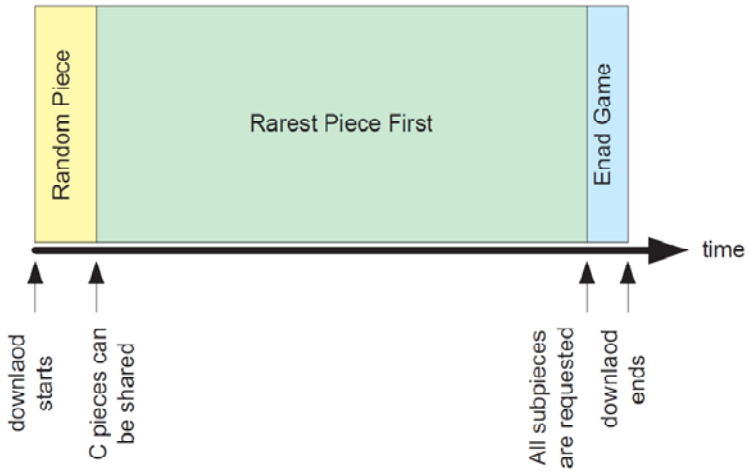

Figure 2-3. Bittorrent piece selection algorithms.

Random First Piece:

When downloading first begins, as the peer has nothing to upload, a piece is selected at random to get the download started. Random pieces are then chosen until the first piece is completed and checked. Once this happens, the "rarest first" strategy begins.

Rarest First:

When a peer selects which piece to download next, the rarest piece will be chosen from the current swarm, i.e. the piece held by the lowest number of peers. This means that the most common pieces are left until later, and the peer focuses in replication of rarer pieces.

At the beginning of a torrent, there will be only one seed with the complete file. There would be a possible bottle neck if multiple downloaders were trying to access the same piece. Rarest first avoids this because different peers have different pieces. As more peers connect rarest first will load off the seed, as peers begin to download from one another.

Eventually the original seed will disappear from a torrent. This could be for cost reasons, or most commonly because of bandwidth issues. Losing a seed has the risk of pieces being lost if

no current downloaders have them. Rarest first works to prevent the loss of pieces by replicating the pieces most at risk as quickly as possible.

If the original seed goes before at least one other peer has the complete file, then no one will reach completion, unless a seed re-connects.

Endgame Mode:

When a download nears completion, and waiting for a piece from a peer with slow transfer rates, completion may be delayed. To prevent this, the remaining sub-pieces are request from all peers in the current swarm.

2.1.5.2. Choking algorithm

When a peer receives a request for a piece from another peer, it can opt to refuse to transmit that piece. If this happens, the peer is said to be choked. This can be done for different reasons, but the most common is that by default, a client will only maintain a default number of simultaneous uploads (max_uploads). All further requests to the client will be marked as choked. Usually the default for max_uploads is 4.

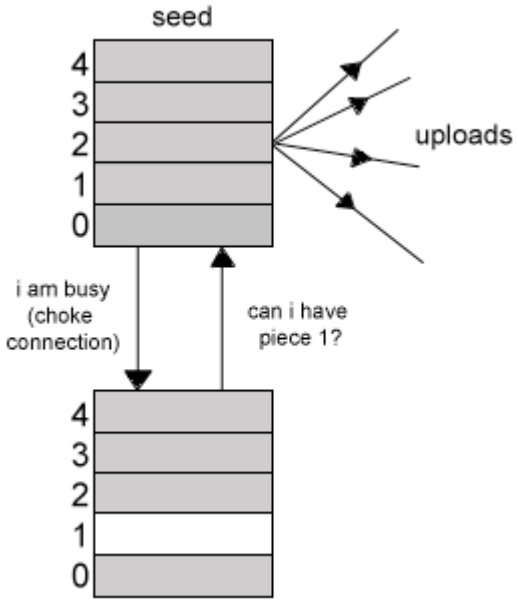

Figure 2-4. A seed chokes the connection to a peer because it has reached its maximum uploads.

The peer will then remain choked until an unchoke message is sent. To ensure fairness between peers, there are mechanisms which rotate the peers that are downloading. These are known as rechoking and optimistic unchoking.

Rechoking or Tit-For-Tat:

To ensure that peers who have highest upload ratios are rewarded, and at the same time penalize the ones who give less, each peer does a rechoking process every 10 seconds.

Rechoking consist in sorting all interested remote peers by their upload rate towards the peer who is rechoking. Then the peer doing the rechoking send unchokes to the top four and at a time send chokes to the rest.

Optimistic Unchoke:

BitTorrent applies the optimistic unchoke mechanism in parallel to the tit-for-tat mechanism. The goals of the optimistic unchoke mechanism are: 1) to enable a continuous discovery of better peers to reciprocate with, 2) to bootstrap new leechers who do not have any content piece, to download some data and start reciprocate pieces with others.

To ensure that connections with the best data transfer rates are not favored, each peer has a reserved upload slot called “optimistic unchoke”. The peer who receives that extra slot is left unchoked regardless of the current transfer rate. This slot is rotated every 30 seconds. This is enough time for the upload/download rates to reach maximum capacity and the optimistic unchoked peer can compete for enter in the rechoking top four.

2.1.5.3. Communication between peers

Peers who are exchanging data are in constant communication. Connections are symmetrical, and therefore messages can be exchanged in both directions. These messages are made up of a handshake, followed by a never-ending stream of length-prefixed messages. This constant stream of messages allows all peers in the swarm to send data, and control interactions with other peers.

The messages used by peers to communicate are the following:

- **choke message**: this enables a peer to block another peers request for data.
- **unchoke message**: indicate to the receiver that the sender allows to download pieces from him.
- **interested message**: a peer sends this message to indicate to the receiver that has required pieces.
- **not interested message**: a peer send this message to indicate to the receiver that does not have required pieces.
- **have**: a peer indicates to the receiver that have the piece included in the message
- **handshake message**: an introduction message, indicate that a peer wants to start a piece exchange.
- **bitfield message**: sent immediately after handshaking. Optional, and only sent if client has pieces.
- **piece message**: response to request message. With this message is send the asked block of the requested piece.
- **request message**: message used to request a block of a piece.
- **cancel message**: used to cancel block requests.

Figure 2-5 shows an example of a possible message flow among peers that have an active connection in a BitTorrent overlay network.

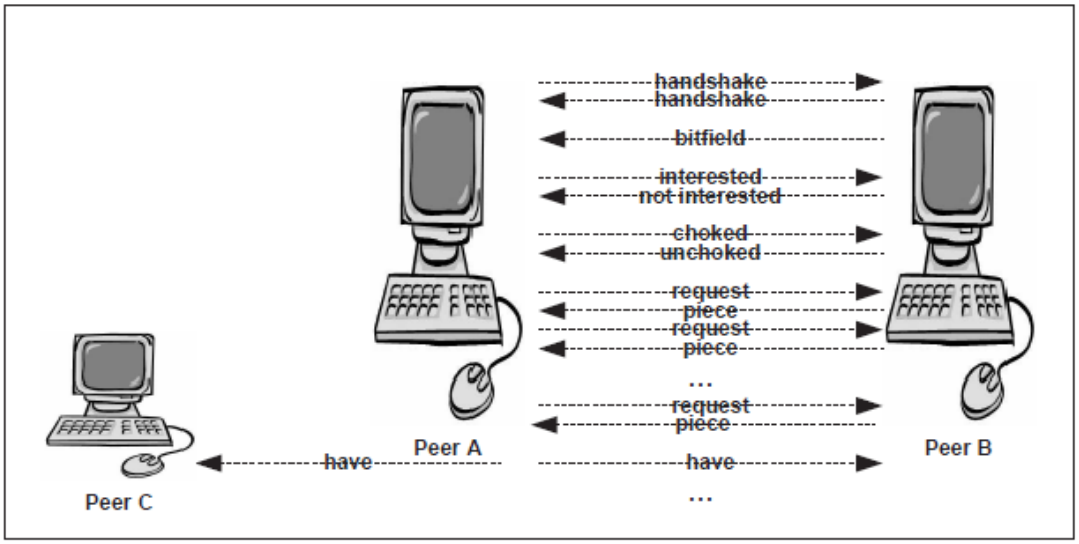

Figure 2-5. An illustrative example for a message flow among peers.

In the example the connection is established after peer A sends a "handshake" message, and B responses with one as well. Then peer B sends a "bitfield" message but peer A does not. Such a scenario might happen if peer A has no piece ready to be shared. Peer B sends a "not interested" message to peer A, and peer A sends a "choke" message to peer B, thus data will not flow from peer A to peer B until both messages will be replaced. On the other hand, data does flow from peer B to peer A since peer A sends an "interested" message to peer B and peer B sends an "unchoked" message to peer A. Then, peer A requests pieces of a particular piece and peer B responses with "piece" messages, by uploading the requested pieces. Once peer A gets an entire piece and confirms the validity of the piece, it sends "have" messages regarding the received piece to all the peers it is connected to in the BitTorrent overlay network.

2.1.6. BitTorrent clients

A BitTorrent client is an executable program which implements the BitTorrent protocol. It runs together with the operating system on a users machine, and handles interactions with the tracker and peers. The client is sits on the operating system and is responsible for controlling the reading / writing of files, opening sockets etc.

A metainfo file must be opened by the client to start partaking in a torrent. Once the file is read, the necessary data is extracted, and a socket must be opened to contact the tracker. BitTorrent clients use TCP ports 6881-6999. To find an available port, the client will start at the lowest port, and work upwards until it finds one it can use. This means the client will only use one port, and opening another BitTorrent client will use another port. A client can handle multiple torrents running concurrently.

There are a wide range of different Clients, and can range from basic applications with few features to very advanced, customizable ones. For example, some advanced features are metainfo file wizards and inbuilt trackers. These additional features mean that different clients behave very differently, and may use multiple ports, depending on the number of processes it is running. As all applications implement the same protocol, there are no incompatibility issues,

however because of various tweaks and improvements between clients, a peer may experience better performance from peers running the same client.

2.2. Related Work

The main mechanisms and design of BitTorrent were first described by Bram Cohen, the protocol creator, in [7]. Since then, an extensive research work has focused on performance and behavior of BitTorrent systems.

A survey of performance studies of BitTorrent from 2003 to 2008 were presented in [8]. Raymond Lei Xia et al. categorize these studies based on the techniques used, the mechanisms studied and the resulting observations about BitTorrent performance. When it came to mechanisms studied they note that can be viewed as belonging to two categories: (a) modifications to peer selection strategy, and (b) modifications to the piece exchange mechanisms. The same distinction was made in [9], where Felber et al. compare different peer and piece selection strategies in static scenarios using simulations.

Most of peer selection approaches are focused on which peers are received from the tracker. Bindal et al. [10] designed a new algorithm based on biased neighbor selection, in which a peer chooses its neighbors mostly from peers within the same ISP, and only selects a few from the outside its ISP domain. This is because peers in the same ISP are highly connected. However, some connectivity to peers outside the ISP boundaries needs to be maintained in order to obtain new blocks. They indicated that the biased neighbor selection can be implemented by changing the tracker and client. The tracker can know the peers' ISP location either by using the AS mapping or IP address to identify the ISP. Another similar strategy was studied in [11]. Yamazaki et al. proposed a series of strategies to reduce ISP costs called Cost-Aware BitTorrent (CAT). In CAT, the peers first acquire ISP cost information. Thereafter, path costs between any two peers can be computed using the ISP costs. The tracker selects peers in the list of peers based on the shortest cost to the requesting peer. The strategy is to minimize the cost incurred in the transactions, consequently minimizing the ISP costs. Through simulations for different scenarios they show that CAT can reduce the ISP cost and improve the performance.

As for the piece exchange mechanisms, Chi-Jen Wu et al. proposed a weighty piece selection strategy instead of the local rarest first strategy in [12]. The proposed strategy is based on the greedy concept that a peer assigns each missing piece a weight according to total number of neighbor's downloaded pieces. The peer selects the missing piece with the highest priority for next download. This strategy can speed up the cooperation between heterogeneous peers while making the BitTorrent more efficient in terms of the average download time and the total elapsed time.

All these improvements are limited by the lack of information that the BitTorrent protocol provides about the rest of the swarm and its components. The vision of a peer on other components of the swarm is mainly restricted to two elements: peers list provided by the tracker and the pieces that every peer have. This restricts the optimizations that can be done without altering the BitTorrent protocol. Despite the limited information available to peers, there are several approaches which assume a greater knowledge of the global state of the

swarm and its components. This is the case of BitMax [13], which proposes a protocol modification that all nodes must implement. A BitMax node uploads content to less simultaneous neighbors but at a higher bandwidth rate. The upload bandwidth to each neighbor is adjusted to the neighbor download capacity. This implies that every node should also know its counterpart capacities. Compared to regular BitTorrent, BitMax can distribute data faster implementing an unchoke strategy not based on reciprocation but in a distribution interest. This makes sense for controlled environments but it is not feasible in open swarms with heterogeneous clients.

Qiu and Srikant [5] extend the initial work presented in [14] by providing an analytical solution to a fluid model of Bit-Torrent. Their results show the high efficiency in terms of system capacity utilization of BitTorrent, both in a steady state and in a transient regime. However, a major limitation of this analytical model is the assumption of global knowledge of all peers to make the peer selection. However, in a real system, each peer has only a limited view of the other peers.

Other researchers have looked into the feasibility of free-riding behavior. Free-riders are those peers who attempt to circumvent the protocol mechanism and download data without uploading any data to other peers in the network. BitTyrant [15] is a strategic client that adjusts its upload bandwidth to the minimum required to be unchoked by its neighbors. This allows BitTyrant to barter with as many peers as its bandwidth supports, not giving extra upload bandwidth to any peer. BitTyrant authors claim that using this strategy can reduce download times up to a 70%. Although it benefits BitTyrant clients, this strategy degrades the global fairness and download speed over the swarm [12], so other clients service is lower. The authors of BitTyrant tested their client in an environment with many other BitTyrant clients and concluded that their performance dramatically decreases when multiple nodes are using this client in the same warm.

Another well known free-riding approach is [16]. BitThief studies the feasibility of downloading in BitTorrent without uploading. A BitThief client attempts to enter as many peers' optimistic unchoke slots as possible. This strategy results in a tragedy of the common peers. Entering as many optimistic unchoke slots as possible is a rational strategy for all peers to make (it always improves the expected completion time), but when a significant percentage of the peers run the BitThief client, overall performance will logically decrease.

Some have also considered implementation-specific attacks, such as uploading garbage data [6] or downloading pieces only from seeds [17]. All this strategies mean a great improvement for the BitTorrent client which uses them. However, they suppose a very negative impact for the rest of the swarm.

Lastly, there have been some other approaches to having peers cooperate to download content. Wong's thesis [18] presents a system based on volunteer helper peers, who download popular content pieces and upload them to others. A helper does not have interest in the content and does not need to download all the content to contribute. Similar cases are 2Fast and Tribler. The 2Fast protocol presented in [19] enables a collector peer to task others to download data on its behalf. Tribler [20] is a BitTorrent client that implements 2Fast. Tribler implements social-based mechanisms that let the user add external clients as buddies. These

buddies can be claimed as helpers using 2Fast protocol when they have idle upload capacity. Helping your friends is a cross-torrent solution, the helper client have to join the swarm just for helping the collector. However, this system provides no mechanism to enforce reciprocation of this help in the future, but rather relies on social (friend) pressure instead.

BTSlave [21] is a BitTorrent protocol extension that also attempts to leverage a friends approach. Similar to 2Fast BTSlave, it also suffers from the absence of a mechanism to enforce reciprocation in the future. Lastly, Wang et al. [22] propose to utilize helpers for improved performance, by having them download a small number of random content pieces. Then they upload these pieces to those with a small fraction of the content. Their approach does not address the issue of incentives for helpers, and also creates an additional opportunity for free-riders, who can misreport their download progress to the tracker to receive more data from helpers for free.

In this master thesis we propose a novel improvement over BitTorrent protocol based on Artificial Intelligence techniques. We outline that AI techniques never have been used on a BitTorrent improvement before. Finally, in order to avoid a free-rider behavior, we will combine the benefits of this improvement with cooperative techniques which minimize the negative impact for the rest of the swarm.

3. Problem Statements

The main goal of this project is the optimization and improvement of BitTorrent protocol algorithms. Our work focuses on artificial intelligence techniques and algorithms which increase the overall performance of the protocol. The aim of these techniques is to make predictions about the status of peers and achieve a greater knowledge about the swarm. The acquisition of this knowledge will be used to improve the algorithms focusing on increasing its download capacity.

Some BitTorrent clients manage to increase its download capacity by trading selfishly with other peers. This kind of strategies has the main problem that they negatively affect the rest of peers downloading the same content. To avoid this situation, our second objective is to design a collaborative strategy which focuses its efforts on improving the overall state of the swarm. This second system will be used as a support for our Artificial Intelligence strategy.

3.1. IA Strategy: Download Price Prediction

Both the original algorithms of BitTorrent protocol and new strategies, that are designed to improve them, face major restrictions at the time making decisions. These limitations are caused because they can only have access to local information. This means that a particular peer does not have access to other peers' data, except for the one which BitTorrent protocol provides directly (e.g. piece information) or indirectly (bandwidth calculated via piece arrival ratio). Having extra information about other peers in the swarm could be a great advantage at the time of making decisions, especially if these decisions are going to influence directly on peers which are going to provide us the needed pieces.

There are some kinds of artificial intelligent algorithms that can be trained to make numerical predictions. These algorithms use large amounts of information, collected in similar environments, to make their predictions. Using this information, which represents the set of system experience, the algorithm can look for patterns in past situations to observe similarities with the current ones and guess the most probable value.

The predictive method we propose can be used to estimate any relevant value of BitTorrent environment. For this to be possible, a customized experience dataset is required for each case. Our goal is to achieve an improvement in download time, so we will predict a value which can be used to optimize BT algorithms in this way.

Let us define the Download Price as a value that indicates the cost of downloading pieces from a peer. How profitable it is to download pieces from a certain peer would be another way to define it. This value relates to two factors: i) the bandwidth we get from a peer. ii) the bandwidth we must give. We want to outline that the second value is a little bit complex to determine. That is because a peer will only give us permission to download if we give him a certain amount of bandwidth. Furthermore, this amount is not fixed and its value depends on the number of peers who are offering their bandwidth at a certain moment.

As the relation between the two factors mentioned above, Download Price can help us to identify which peers are better suited to download from. A low download price means that a peer is giving more bandwidth than which is receiving. This may indicate that a peer has high

bandwidth capacities or a low pieces demand. In any case, selecting those peers as objectives to download can result in download time improvement, and at the same time we are reducing the probability of giving a portion of our bandwidth to a peer who will never return anything in exchange.

3.2. Collaborative Strategy

The main idea of our collaborative strategy involves the creation of a group of peers (Crowd) interested in the same digital content. This group will use collaborative policies to increase the performance of BitTorrent protocol.

We must take in count certain considerations when we are defining the Crowd. We assume that these groups will be reduced in size and their number will range from two to seven members. In real scenarios Crowd peers must be trustable users who do not betray the group, in other words, peers who do not have dishonest or selfish behavior towards its companions. With bigger groups this could be more difficult to handle. We consider outside the scope of the project to determine how peers will establish coalitions to form Crowd groups. When we perform simulations we assume that groups will be predefined.

We consider two main approaches regarding collaborative strategies. The first approach will lead group collaborative efforts on obtaining, as soon as possible, all content pieces among all group members. With this process they will ensure the content completion for the Crowd, as well as for the swarm. The other approach will focus exclusively in the Crowd and how its members should work together to improve their overall performance.

These two approaches result in two improvements over the existing BT protocol for clients participating in this collaborative group:

- Piece selection: group members cooperate to improve the piece selection process.
- Unchoke policy: group members will obtain priority in front of regular peers, but only under certain circumstances.

3.2.1. Group Piece Selection Strategy

Group Piece Selection is an improvement of the piece selection algorithm that is aimed at collaboration among peers Crowd. The goal is to get the group behave as a single entity when its members are downloading the same content. Peers using this new strategy will select content pieces according to the needs of the Crowd, giving preference to those parts that do not have any of its members. If they collaborate in this way, a distributed copy among the Crowd components will be achieved.

A distributed copy is a full copy of the content shared among the group peers. The distributed copy exists adding up all the pieces that each peer has achieved individually. Getting this distributed copy as quickly as possible should be the main goal of the group members. Theoretically the larger the group the sooner the distributed copy should be achieved. This is because the copy is more distributed and each peer has to download a smaller portion of the content. However, we must take in count that piece collusions may occur. A piece collusion happens when at least two group members are downloading the same

piece of the content. Obviously a larger group has higher collusion probability, so we will pay special attention when we study the group optimum size.

With this policy we expect an improvement on the pieces availability, as for the crowd as for the whole swarm as well. Another important objective of this strategy is to ensure the completion of the content for crowd members. Once the distributed copy is completed we can state that members of the Crowd have guaranteed the completion of the shared file. That is because crowd members have the whole set of pieces of the content. At this point, even if the rest of nodes leave the swarm, the crowd members will finish downloading their contents.

3.2.2. Crowd Members First Strategy

We want to expand and apply the collaborative algorithm for Group Piece Selection to the BitTorrent Rechoking algorithm.

We propose a Crowd Members First policy that seeks that members of the Crowd have more privileges when they are downloading content between them. The basic idea is that a peer of the crowd will always have a download slot reserved for group members. Thus, when a Crowd peer requests downloading parts it will be allowed without having to compete with other regular peers of the swarm.

Our objective with this policy is to achieve strongest peers (more download/upload capacity) or richest peers (more pieces diversity) help the rest of crowd members. Especially those peers which, either by its weakness (low download/upload capacity) or simply because they have entered late in the swarm, does not have anyone to get pieces from. This strategy is also proposed to increase the download ratio of peers of the Crowd. This is reasonable because each one of them always will have someone to download pieces from.

We must emphasize that this strategy and the Group Piece Selection proposal, detailed in the previous section, pursue opposing objectives. One seeks to complete a distributed copy of the content, and the other wants to facilitate the exchange of pieces between the members of the crowd. The strategy Crowd Members First has been designed as complementary to Piece Selection Group. It is oriented at facilitating that peers can finish their contents once the distributed download is complete. The use of this strategy at any other point of download life cycle would be completely counterproductive.

4. Download Price Prediction Strategy

There are a huge variety of BitTorrent clients which use different rechoking approaches to increase they performance in one way or another. However, all of them are limited at the time of getting information about the rest of the swarm. As we have mentioned in section 3.1, a peer only knows the information that BitTorrent protocol provides of the rest of the swarm.

Artificial Intelligence offers different algorithms that can be trained to make numerical predictions based on historical traces or samples they had seen before. For this purpose, Download Price Prediction strategy uses the k-Nearest Neighbours algorithm to predict values that would be impossible to calculate by means of mathematical formulas. The objective of this strategy is to use the kN Neighbours algorithm to calculate the Download Price of other peers. Download Price is a value that indicates the cost of downloading pieces from a peer. It can also be seen as how profitable is to download pieces from a peer.

An important part of our strategy is the training process of Download Price Prediction system. For this purpose we will use a Genetic algorithm. This algorithm will focus on optimizing the results of our predictor by searching the optimum weight of input variables.

In the following sections we will explain how our Download Price Prediction strategy has been built. Firstly, we will describe how the kN Neighbours algorithm calculates download prices. Secondly, we will explain how the system training is performed. Finally, we will focus on integrating this strategy to BitTorrent.

4.1. Building the dataset

The first important element we have to build for implementing the Download Price Prediction strategy is a solid dataset. This dataset will be used as “experience” for the k-Nearest Neighbours to make numerical predictions. In this section we will explain how our dataset is built and which elements form it. The explanation of how the k-nearest Neighbours make use of this experience will be explained in further sections.

A dataset is composed of multiple entries containing information about the value we want to predict, in our case the Download Price. Each one of these entries contains information about the real price in certain specific conditions.

```
Feature1;feature2;…;featureN;Result1
Feature1;feature2;…;featureN;Result2
Feature1;feature2;…;featureN;Result3
                …
Feature1;feature2;…;featureN;ResultN
```

Figure 4-1. Dataset structure.

Figure 4-1 shows the structure of a dataset. Each dataset entry is composed by a group of Features and their corresponding result (result is the variable we want to predict). Features are a group of variables which have some relation with the value we want to predict. So, in order to create a dataset we need to collect different value combinations of chosen features and its respective results. Our system uses these datasets as experience to make its predictions.

In the following sections we will see how our dataset is built and which variables take part in the process.

4.1.1. Calculating Download Price

The k-Nearest Neighbours algorithm can be used to estimate any relevant value from the BitTorrent environment. In order to make this possible we need a customized dataset for each value we want to predict. Our goal is to optimize peers download time, so we should choose a value which can be used to optimize BT algorithms in this way. In our case, the result field of the dataset will be the Download Price.

In previous sections we have defined Download Price as the cost of downloading pieces from a peer. Other definition that we have used is how profitable it is downloading pieces from a certain peer. Next formula shows that Download Price relates two factors:

$$Download\ Price = \frac{(A)\ Bandwidth\ we\ give\ to\ get\ unchoke}{(B) Bandwidth\ we\ get\ from\ that\ peer} \times 100$$

Figure 4-2. Download Price formula.

Figure 4-2 shows the two values needed to calculate the Download Price. The (A) value is the bandwidth that we give to a peer. Due to the tit-for-tat strategy used by BT protocol, a peer will only give us permission to download (unchoke state) if we give him a certain amount of our bandwidth. This amount is not fixed and depends of the number of peers that are offering their bandwidth at certain moment. The second value (B) is the bandwidth that we are receiving from that peer when we receive the unchoke state.

Both values, (A) and (B), cannot be known by the peer making the prediction. In order to construct the dataset, we will use the omniscient information that the simulator provides. That will make possible to calculate the Download Price anytime we want.

4.1.2. Choosing the features

In the previous point we have seen a formula to calculate the Download Price and which variables are needed obtain its value. The next step to build the dataset is to choose a group of features that can affect directly or indirectly to the Download Price of a peer. However, another consideration must be taken into account. These features are going to be used as inputs by kN Neighbours algorithm for making predictions. So, all of them must be accessible to the peer who is using our system. The eight chosen features fulfill the proposed conditions:

Seeds Proportion (*Seedp*):

This feature is expressed as a percentage and indicates how many peers in the swarm are seeds. Usually, the more seeds are in the swarm, the lower the download prices are. So, this value has a great impact over our system. To calculate this feature we only need to know how many peers have all content pieces and how many peers are in total. All this information can be extracted from BitTorrent protocol.

Download progress percentage of the target peer (*FinishPerTarg*):

Indicates the download progress percentage of the peer which we are predicting the price. This value could be important, as indicates how many pieces the target peer has. Depending on the number of pieces it has would be more o less peers challenging for the upload slots, and this can affect the Download Price. This information is also provided by the BT protocol, as it can be calculated from peers' pieces.

Have message ratio of target peer (*PeerHaveMR*):

With this feature we can guess the bandwidth capacity of the target peer. Peer capacity could be a very important factor because indicate how many upload bandwidth can give to unchoked peers. As we mentioned in the previous point, this value has direct implications on the download price.

Calculate this feature it is more complex than the others. That is because it is calculated from arrival frequency rate of HAVE messages of the BT protocol (this messages indicate that a peer has acquired a new piece). To estimate the capacity we counted the elapsed time between the arrival of HAVE messages of a same peer. As the size of a piece is fixed, we can approximate which is the download capacity of the peer. However, this variable can be a bit imprecise until we receive enough "HAVE messages" to calculate a solid average.

Percentage of interested peers in the target (*estimInter*):

This feature is an estimation of the number of peers interested in the target. As to calculate how many peers are interested in the target would be very costly, we have made an estimation. We suppose that only the peers with fewer pieces than the target peer are interested. This way the results are similar and the cost lower. To calculate this feature we only use the accomplished download percent of peers, which can be easily obtained from the info that BT protocol provides.

Piece rarity average of the target (*rarityAv*):

It indicates the average rarity of the pieces which the target peer has. To measure the rarity we check how many peers are missing a piece and we express it as a percentage. We repeat this process for all pieces that target peer has and we calculate the average.

Having the rarest pieces could lead to have more peers challenging for target peer's upload. This probably will mean an increase in the price of the download. To calculate this variable we use the same data that peers use for the rarest first algorithm, so the information is fully accessible.

Most rare piece of the target (*rarestP*):

Very similar to the previous feature, but instead of calculating the average we find for the rarest piece. This feature is also expressed as a percentage of peers who are missing the piece.

Download progress percentage of the asker peer (*AskerFinishedPercent*):

This feature contains the download progress of the peer who is making the prediction. This variable is special case and we do not think that it alters the Download price in any way. The reason to add this feature is to demonstrate the proper work of the training algorithm, which will probably discard it when we perform the features optimization.

Upload bandwidth of asker peer (*AskerUp*):

This last feature represents the upload capacity of the peer who is making the prediction. We must note that the information needed in this case corresponds to the peer who is using the Download Price Prediction strategy. In this case information is stored locally and can be reached without any problem.

4.1.3. Build Process

To build the dataset we run a simulation with a pre-set scenario. During the simulation, every time a peer calls the rechoking algorithm we calculate all the feature values and the download price as well. Once we have this information we add a new entry to the dataset. While we are building the dataset, we will use the omniscience of the simulator in order to obtain any needed information. But we have to remind that at the time of making predictions we will only have access to the data that BT protocol provides.

With this procedure we obtain a very complete dataset which covers a huge variety of cases. Of course this dataset will be specialized in scenarios which are similar to the one we used to gather the experience. However, nothing prevents us to build several datasets, each one specialized in different environments, and select the most indicated in any case.

4.2. K-Nearest neighbours

Once we have built a suitable dataset, we need a way to use this information in order to make price predictions. The easiest approach to our Download Price problem is the same we would use if we were trying to price something manually. This consists in finding a few of the most similar items and assume the prices will be roughly the same. By finding a set of items similar to the item that interests us, the algorithm can average their Download Prices and make a guess at what price this item should have. This described approach is called k-nearest neighbours (kNN).

4.2.1. Number of neighbours

The k in kNN refers to the number of items that will be averaged to get the final result. If the data were perfect, we could use k=1, this means that we just pick the nearest neighbour and use its price as the answer. But in real-world situations there are always aberrations. For this reason, it is better to take a few neighbours and average them to reduce any noise.

Choosing the number of neighbours is a difficult task. Having too few neighbours will make us susceptible to dataset entries noise, and having too much will reduce accuracy because the algorithm will be averaging items that are not similar to the query.

For avoiding the problem of choosing neighbours number we will use weighted neighbours. With this approach we can compensate the fact that the algorithm may be using neighbours

that are too far away. That is because we are weighting them according to their distance. So, we can choose k = 4 without fear of reducing accuracy.

4.2.2. Defining similarity

The first thing we will need for the kNN algorithm is a way to measure how similar two items are. For our case we will be using Euclidean distance. Pseudocode can be seen in Figure 4-3:

```
FUNCTION euclidean(Feature_List v1, Feature_List v2 )
    d = 0.0
    FOR i in v1.size DO
          d += (v1[i]-v2[i])^2

    RETURN math.sqrt(d)

END FUNCTION
```

Figure 4-3. Euclidean distance.

We can notice that this function treats all features the same when calculating distance, even though in any real problem, some variables have a greater impact on the final price than others. This is a well-known weakness of kNN, and in further points we will see ways to correct it.

4.2.3. Gaussian weighted neighbours

In a previous point we have proposed the problem of selecting too many neighbours. A way to compensate for the fact the algorithm may be using neighbours that are too far away, is to weight them according to their distance.

The more similar the input features are, the smaller the distance between them, so we need a way of converting distances to weights. For this purpose we will use the Gaussian function in figure 4-4:

```
FUNCTION gaussian(float dist)
    sigma=1.0
    gauss = math.e^(-dist^2/(2*sigma^2))
    RETURN gauss

END FUNCTION
```

Figure 4-4. Gaussian function.

4.2.4. Implementing k-Nearest Neighbours

The k-Nearest Neighbours algorithm is relatively simple to implement. It is computationally costly, but it has the advantage of no requiring training every time we add new data to the dataset.

The first step for implementing this algorithm is building a function to get the distances between a given item and every item in the original dataset:

```
FUNCTION getdistances(dataset, Feature_List vec1)
     distancelist = create empty list for saving distances
     //Calculate the distance of vec1 with all entries of dataset
     FOR i in dataset.size DO
           vec2 = dataset[i].getFeatures
           distancelist[i][0] = euclidean(vec1,vec2)
           distancelist[i][1] = dataset position

     //Once all distances are calculated
     Sort distancelist
     RETURN distancelist
END FUNCTION
```

Figure 4-5. Function for calculate distances.

The function on figure 4-5 calls the Euclidean function (figure 4-3) on the given vector against every other vector in the dataset and puts them in a big list (distancelist). The list is sorted so that the closest item is at the top.

The kNN function uses the list of sorted distances and averages the top k results (four in our case). Of course, we consider weighted averages which are calculated with the Gaussian function on figure 4-4. The weighted average is calculated by multiplying each item's weight by its value before adding them together. After the sum is calculated, it is divided by the sum of all the weights.

```
FUNCTION weightedknn(dataset, Feature_List vec1, int k)
     //Calculate distances
     dlist = getdistances(dataset,vec1)

     //Initialize variables
     avg = 0.0
     totalweight = 0.0

     //Get weighed average
     FOR i in k DO
           dist = dlist[i][0]
           datasetPos = dlist[i][1]
           weight = gaussian(dist) //Get the Gaussian weight
           //Calculate the avg price
           avg += weight*data[datasetPos].getDownloadPrice
           totalweight += weight

     avg = avg/totalweight
     RETURN avg
END FUNCTION
```

Figure 4-6. Weighted k-Nearest Neighbours.

The k-NN pseudocode is presented in Figure 4-6. This function loops over the k Nearest Neighbours and passes each of their distances to the Gaussian function defined in figure 4-4. The avg variable is calculated by multiplying these weights by the neighbour's value. The *totalweight* variable is the sum of the weights. At the end, *avg* is divided by *totalweight*.

4.2.5. Testing k-Nearest Neighbours

Once we have the dataset and the kNN algorithm ready, we can make some predictions to check how the Download Price Prediction system works. For this test we have used the simulator advantages to calculate several Download Prices and their respective features. Then we set the features as kNN inputs to make predictions and compare the results to the real Download prices.

```
INPUT FEATURES:

SeedProp: 21.42             TargetFinishPer: 63.77      TargetCapacityHR: 44.63
EstimInterest: 28.57        RarityAverage: 70.17        RarestPiece: 60.61
AskerFinishPer: 70.0        AskerUpload: 100.0

REAL RESULT:
Dowload price: 64.61

PREDICTION for 4-Nearest Neighbours
     Distance: 1.28        Result: 65.60
     Distance: 5.01        Result: 49.25
     Distance: 5.07        Result: 68.93
     Distance: 5.82        Result: 68.28

Download Price prediction: 65.54
Prediction error: 0.93
```

Table 4-1. kNN accurate prediction.

The results in table 4-1 show an accurate prediction of the kNN algorithm. As we can see, the items with nearest distances have similar results, so it is obvious that the prediction is performed correctly. However, this is only one example of well-performed test. Now let's see its counterpart:

```
INPUT FEATURES:

SeedProp: 5.26              TargetFinishPer: 27.04      TargetCapacityHR: 42.41
EstimInterest: 89.47        RarityAverage: 34.45        RarestPiece: 5.2
AskerFinishPer: 22.95       AskerUpload: 25.0

REAL RESULT:
Dowload price: 164.61

PREDICTION for 4-Nearest Neighbours
     Distance: 1.32        Result: 120.54
     Distance: 6.01        Result: 101.80
     Distance: 9.01        Result: 170.63
     Distance: 9.39        Result: 83.64

Download Price prediction: 120.54
Prediction error: 43.97
```

Table 4-2. kNN inaccurate prediction.

In table 4-2 we see an opposite case in which the prediction is not accurate at all (of course this is the worst prediction we have found). Although the algorithm has found similar items in the dataset, the obtained results are not similar to the real download price. As the download prices have an approximate range of 0 to 200, we can consider that an error of 21% is big. Despite this level of error, we can still determine if the price is high or low, which is the main objective of this strategy.

Despite what this results show, these performed tests cannot be taken in count because only represent a very small part of all possible cases. In the next section we present a method for doing more accurate testing.

4.3. Cross-Validation testing

At this point we need a way to verify the proper running of the k-Nearest Neighbours algorithm and the accuracy of its predictions. We can make some manual predictions for testing, but this results cannot be considered as relevant because only cover a little area of whole dataset. Therefore, we need a methodology which can make exhaustive tests that verify the algorithm accuracy. For that purpose we have used the Cross-validation technique.

4.3.1. Implementing Cross-validation technique

Cross-validation is a set of techniques that divide the data (in our case the dataset) into training sets and test sets. The training set is given to the algorithm, along with the correct answers (in this case, Download prices), and becomes the set used to make predictions. The algorithm is then asked to make predictions for each item in the test set. The received answers are compared to the correct answers, and an overall score is calculated. This way we can know how well the algorithm has performed.

Usually this procedure is done several times, dividing the dataset differently each time. Normally, the test set is a small portion, more or less 5 percent of the whole data, with the remaining 95 percent as the training set.

The first step to implement this technique is to split the dataset in two smaller sets.

```
FUNCTION dividedata(dataset)
    trainset = new list for saving the training set
    testset = new list for saving the test set
    //5% of entries are used to the testset
    FOR each line in dataset DO
          IF (random 1 to 0 number) > 0.05 THEN
                testset.add(line)
          ELSE
                trainset.add(line)

    RETURN testset, trainset

END FUNCTION
```

Figure 4-7. Cross-Validation dataset division.

Once we have a testset it is time to start making massive predictions. The next step is testing the kNN algorithm by giving it a training set (used as dataset to make predictions) and calling it with each entry in the test set. The function in figure 4-8 calculates the differences and combines them to create an average score.

```
FUNCTION kNNtest(traintest, testset)
    error = 0.0
    FOR each line in testset DO
          //to calculate the error we calculate the difference the
          //between the real result and the guess
          guess = weightedknn(trainset,line.getFeatures,4)
          error += math.absolute(line.getResult – guess)

    //we return the average error
    RETURN error/testset.size

END FUNCTION
```

Figure 4-8. Cross-Validation single testing.

The presented function in figure 4-8 accepts a dataset and a group of queries as inputs. It loops over every entry of the test set and then calculates the best guess by applying the 4-Nearest Neighbours algorithm. It then subtracts the guess from the real result. At this point, as we want to know the average error of the made predictions, we add up the absolute values of the differences.

The final step is to create a function that makes several different divisions of data and runs kNNtest function on each, adding all the results to get a final score. Figure 4-9 shows the final function to call the Cross-validation procedure.

```
FUNCTION crossvalidate(dataset, trials)
    error = 0.0
    FOR i in trials DO
          trainset,testset = dividedata(dataset)
          error += kNNtest(traintest, testset)

    RETURN error/trials

END FUNCTION
```

Figure 4-9. Cross-Validation function.

4.3.2. Testing kNN with Cross-validation

Now that we have implemented a technique that permits to do massive testing and obtain the average error, we can proceed to evaluate the performance and accuracy of kNN

predictions. For this we will launch several Cross-Validation tests to study the average error of Download Price prediction system.

```
CROSS-VALIDATION TESTS:

CALL crossvalidate(dataset, trials = 100)

ERROR AVERAGES              TOTAL AVERAGE
Test 1: 18.27               Error: 18.02
Test 2: 17.65
Test 3: 18.56
Test 4: 17.90
Test 5: 17.69
```

Table 4-3. Cross-Validation error averages.

Output 4-3 shows five Cross-Validation tests. With these five tests together we estimate that about 75000 predictions have been done, so we can assume that these results are reliable and accurate.

The results show that the total error average of cross-validation predictions is 18.02. Considering that Download Prices values normally oscillate between 0 and 200, this represents an error of 9.01%. Thanks to this value we can confirm that the extreme error ratio in kNN test performed on point 4.2.5 it is rare and does not occurs frequently. However we have to assume that this extreme fails in prediction can occur.

This 9.01% error can be considered as a good result considering that system training still has to be performed. Once the optimization process is completed, we expect this error ratio decrease even more.

4.4. Using Genetic Algorithm for system training

In this point we will train our download price predictor. With this training we want to achieve better predictions in order to decrease error ratio obtained in Cross-Validation tests. This training consists in finding the importance of each input feature in price prediction process. To achieve this we will give different weights to input variables and see how this affects to predictions and to error rates. Using this system we can find which variables are important and which ones can be rejected. The first step to make this training process is to find a way to weight the eight chosen features.

4.4.1. Scaling Features

In point 4.2 we have defined our prediction algorithm. This prediction algorithm consists in kNN system that uses Euclidean distance to find nearest items to the input features. The fact of using Euclidean distance is important, because it turns the kNN sensible to the scale changes of its input variables. This means that if one of our chosen features has a larger scale than the others, it will have more weight when the prediction is done.

So, in order to give different weights to the input variables we have to rescale them. The simplest form of rescaling is multiplying the value by a constant. However, for our system

rescaling is not that easy, that is because for rescaling a feature we have to rescale the whole dataset. For this purpose we have implemented the rescale function showed in figure 4-10. This function takes a dataset and a parameter called scale, which is a list of integer numbers. It returns a new dataset with all features values multiplied by the values in scale.

```
FUNCTION rescale(dataset, scale)
    scaleddataset = create new dataset
    FOR each line in dataset DO
          FOR i in line.getNumberOfFeatures DO
                //We scale each feature of an entry
                line[i]= line[i]*scale[i]

          //Once we have scaled all features of a line
          //we add the line to the new dataset
          Add line to scaleddataset

    RETURN scaleddataset

END FUNCTION
```

Figure 4-10. Rescale function.

Once this rescale function is done we can change the weights of the chosen features as we like. By combining this function with the Cross-Validation tests we can estimate how the features weights affect to Download Price prediction. At this point we can start doing tests for discovering which weights perform better, but this would be very tedious. In the next point we propose a way to optimize this process.

4.4.2. Genetic Algorithm for scale optimization

In section 4.1.2 we have chosen a total of eight features. We considered that this features could have some impact over the download price. However we do not know for sure which variables are unimportant and which ones have significant impact.

In theory, to make the process of scaling, we could try a lot of numbers in different combinations until we found one that performs well enough. As we have eight features, the number of combinations might be thousands of variables to consider and it would be nearly impossible. Fortunately there are algorithms which by using optimization can find a good solution automatically when there are many input variables. To this purpose we have used a Genetic Algorithm.

4.4.2.1. Genetic Algorithm

In a genetic algorithm, a population of strings (called chromosomes), which contains candidate solutions (called individuals) to an optimization problem, evolves toward better solutions. Traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly

randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. In figure 4-11 we can see the main points of a Genetic Algorithm.

```
1. Choose the initial population of individuals
2. Evaluate the fitness of each individual in that population
3. Repeat on this generation until termination: (time limit, sufficient fitness
   achieved, etc.)
      1. Select the best-fit individuals for reproduction
      2. Breed new individuals through crossover and mutation operations to
         give birth to offspring
      3. Evaluate the individual fitness of new individuals
      4. Replace least-fit population with new individuals
```

Figure 4-11. Genetic Algorithm.

A genetic algorithm as many other heuristic searches requires that we specify a domain (which gives the number of features), a range (values that each element of the domain can get) and a cost function.

4.4.2.2. Cost function building

The cost function is the key to solving any optimization problem. The goal of any optimization algorithm is to find a set of inputs (in this case BitTorrent client properties) that minimizes the cost function, so the cost function has to return a value that represents how good or bad the solution is. The only requirement for a cost function is that return larger values for worse solutions.

In our case we already have a function that accomplishes these requirements. The cross-validation function returns higher values for worse solution, so it is basically a cost function. The only thing we need in order to have a complete cost function for using in the Genetic Algorithm is the following one. Take as input a list of scales (one for each feature we want to weight), rescale the dataset (with function in figure 4-10) and calculate the cross-validation error. The final cost function is showed in figure 4-12.

```
FUNCTION costf (scale)
    datset = get original dataset
    scaleddataset = rescale(dataset)
    RETURN crossvalidate(scaleddata,20)
```

Figure 4-12. Cost function.

Once the cost function is completed we have everything we need to automatically optimize the weights. The genetic algorithm will call the costf function each time it needs to check new scale set, and cross-validate will tell if the new feature weights perform better or worst.

4.4.3. Genetic algorithm optimization test

Now we can optimize the scales in order to determine which variables have more importance in the prediction process. To this end, we use the genetic algorithm giving as inputs the cost function and the domains of scale. The domain is the range of weights that the scale can take. For our training the domain range it is from 0 to 10. It is important to note that zero is included in the range. A zero indicates that a feature is useless and can be ruled out. Now let's see which solution the genetic algorithm determines:

```
OPTIMIZATION TEST:

CALL geneticalgorithm([0,10], costf, maxiterations = 25)

RESULT: [5,2,3,1,1,0,0,2]

FEATURES WEIGHTS ASSIGNATION:
SeedProp: 5            TargetFinishPer: 2     TargetCapacityHR: 3
EstimInterest: 1       RarityAverage: 1       RarestPiece: 0
AskerFinishPer: 0      AskerUpload: 2
```

Table 4-4. Features weight.

Genetic Algorithm returns the results as an array of integers: [5,2,3,1,1,0,0,2]. Each one of these integers represents a weight for one of our eight input features. In table 4-4 we can see which weight is assigned to each variable. Genetic algorithm determines that two features are useless and reduces its scale to 0. If we remember section 4.1.2, we added a useless feature in order to evaluate this training. This variable was *AskerFinishedPercent*. As we see in the results, this is one of the variables that are scaled to zero, so we can assume that optimization have performed well.

Now let's see how the features have changed. As the results show, the most important features to determine Download prices are: the proportion of seeds (*seedProp* = scale 5) and the Bandwidth capacity of target peer (*TargetCapacityHR* = scale 3). On the other side, we have the two discarded features. First, *AskerFinishedPercent,* was expected because it was introduced with this finality. Second, the rarest piece of the target peer (*RarestPiece*). It was a bit unexpected because having a rare piece involves having more peers interested in download from you. This could lead to an increase of Download price.

Regardless of the results, these are the most optimal weights for our Download Price Prediction system. Now that features importance has been determined, we must proceed to test our system in order to confirm that has increased its accuracy.

4.5. Testing the trained Download Price Prediction system

Now that our predictor has been trained and we know how important each feature is, we can proceed to test if our system is performing better. Before starting the testing process we have to modify our old dataset and rescale its entries with the new optimal weights. We will

use the rescale function designed in point 4.4.1 to do this process. Once rescale operations has been done, we are ready to proceed with testing process.

As we have done in previous testing operations, we will use Cross-Validation method to get reliable and complete results. For this test, we will use exactly the same system as in the validation of the untrained Download Price Predictor. We will perform five Cross-Validation tests that involve about 75,000 prediction operations.

```
CROSS-VALIDATION TESTS:
trainedDataset = rescale(dataset, scale)
crossvalidate(trainedDataset, trials = 100)

ERROR AVERAGES          TOTAL AVERAGE
Test 1: 11.23           Error: 11.10
Test 2: 11.78
Test 3: 10.99
Test 4: 10.53
Test 5: 11.01
```

Table 4-5. Average errors for trained system.

In Table 4-5 we can appreciate the results of the five cross-validation tests. This results show that the average error obtained during the process is 11.10 points. The download price can reach a maximum value of 200 points. This means that trained Download Price Prediction system has an average error of 5.55%. Before that we take out the training, performed tests (Table 4.3) showed an error of 18.02 points. This means an error of 9.01%. So we can state that thanks to the optimization performed by the genetic algorithm, the error has been reduced by 6.92 points. This represents that our prediction system is 3.46% more accurate.

From the showed results we can state that k-Nearest Neigbours algorithm can be used to predict values like Download Prices in an accurate way. As we have seen in all the performed tests, there is always a margin of error which is impossible to eliminate. However, this error rate can be minimized thanks to some training performed by genetic algorithms. Thanks to this training we have found two useless input features and our error rate has decreased to acceptable values. In the following sections we will integrate our prediction system to BitTorrent and we will evaluate how this affects to its performance.

4.6. Integrate Download Price Prediction in BitTorrent

Calculate Download Prices thanks to our Download Price Prediction system can be a great advantage for BitTorrent peers. Thanks to this value we can distinguish which peers are more profitable to download from. With this information we can allocate a certain amount of our upload bandwidth to obtain permission to download (unchoke state) from that peers. This way we will be able to obtain more simultaneous unchokes with the same upload capacity. In order to incorporate our Download Price Prediction system to BitTorrent protocol we are going to modify the Optimistic Unchoke algorithm.

BitTorrent applies the optimistic unchoke mechanism in parallel to the Rechoking mechanism. These two algorithms are who decide which peers are going to use a BitTorrent peer upload bandwidth. Each peer has 4+1 upload slots. Rechoking algorithm decides four of these slots and the optimistic unchoke algorithm assigns the last one. The goal of the optimistic unchoke algorithm is to enable a continuous discovery of better peers to download from. The optimistic unchoke mechanism assign its slot to a peer who is chosen randomly. This peer have 30 seconds, until the next optimistic unchoke slot rotation, to reciprocate with its upload capacity and demonstrate that is better than other peers.

With this approach, we want to modify the way optimistic unchoke algorithm assign its upload slot. As we have remark, in the original algorithm the optimistic slot is chosen randomly. We are going to replace this random selection by a download price approach. The main idea is to assign the optimistic slot to peer with the lowest download price.

```
PROCEDURE newOptimisticUnchokeSelection()

peersList = get all known peers;
peersList.shuffle();

FOR EACH peer IN peersList
     minimalDowloadPrice = MAX_VALUE;
     //We calculate download price for each peer
     peerFeatures = peer.getFeatrues();
     currentDownlodPrice = getDownloadPrice(peerFeatures);

     //We save the peer with lower download price
     IF minimalDownloadPrice > currentDownloadPrice THEN
           minimalDownloadPrice = currentDownloadPrice;
           peerToAssign = peer;
     END IF;

END LOOP;

RETURN peerToAssign;

END PROCEDURE;
```

Figure 4-13. New optimistic unchoke peer selection pseudocode.

Figure 4-13 shows the new optimistic unchoke selection process. As we can appreciate, the algorithm gets a list of all known peers. For each one of these peers their Download Price features are calculated. These features were detailed in section 4.1 and their weight in the prediction process was defined in section 4.4. Our Download Price Prediction system uses these features to find out the price of each peer. When all peers prices are have been estimated, the algorithm can determine which peer has the lowest download price. Once we know which peer is most profitable to download from, the algorithm assigns that peer to the optimistic slot.

Once our Download Price Prediction system has been integrated in BitTorrent, we can proceed to validate how the new optimistic unchoke algorithm works and how it affects to the BitTorrent performance.

4.7. Download Price Prediction validation

In previous performed tests we have concluded that our system can be used to predict Download Prices in an accurate way. However, even with the genetic algorithm training, there is always a slight error of 5.5% which is impossible to eliminate. In this section we have performed several experiments to evaluate how the optimistic unchoke algorithm operates when uses our download price prediction approach. All experiments we have made in this project were conducted in discrete event-based peer-to-peer simulator (GPS [23]).

All the experiments to validate our optimistic unchoke algorithm will be performed under the same conditions. For that reason, we have set a common scenario with a heterogeneous environment. In this environment we have categorized two peer models with different upload capacities. These capacities are 0.2 Mb/s for regular peers and 1.0 Mb/s for high capacity peers. As it concerns the download capacity, it is linked to upload capacity and its value is ten times higher: 2.0 Mb/s for regular peers and 10.0 Mb/s for high capacity peers. The swarm will be composed of 31 BitTorrent peers. From these 30 peers, five are high capacity peers and the remaining 25 are regular peers. The last one is the initial seed which starts sharing the digital content. During the simulation progress these 30 will join the swarm following the equation 1 described by Guo et al. in [24]. In order to get solid results we are going to perform 20 simulations for each experiment. In each one of these simulations peers will join in different locations of the p2p network topology. Below we can see a review of simulation scenarios:

Torrent parameters:
- File size: 50 MB
- Pieces: 196
- Peers: 30
- Seeds: 1

Clients are configured as:
- 4 unchoke slots + 1 optimistic
- Upload: 0.20 / 1.00 Mb/s
- Download: 2.0 / 10.0 Mb/s

4.7.1. Test 1: Simulating perfect download price prediction

In this first test we are going to prove that Download Price value can be a determining factor to optimize BitTorrent. Furthermore, we want to show that using this value in our modified optimistic unchoke algorithm can improve the download time of peers. Previous tests have shown that our Download Price system has an average error of 5% in its predictions. This deviation could affect the study of Download Price variable usefulness. Hence, we will use the simulator global knowledge to perform predictions without errors. With this perfect prediction we can study the Download Price value without a miscalculation that alters its importance.

In this experiment, we have performed two different simulations under the previously detailed scenario. In these simulations we monitor the same peer using different strategies. In the first one, the peer does not use any special strategy. Our objective is to observe its normal behavior. In the second one, the peer uses our optimistic unchoke strategy with a perfect Download Price Prediction system.

In Figure 4-14 we can appreciate a great improvement when the peer is using our Optimistic Unchoke algorithm modification. The numerous performed simulations show a decrease in peer download time of 298.42 seconds. This represents a performance increase of 25.48%.

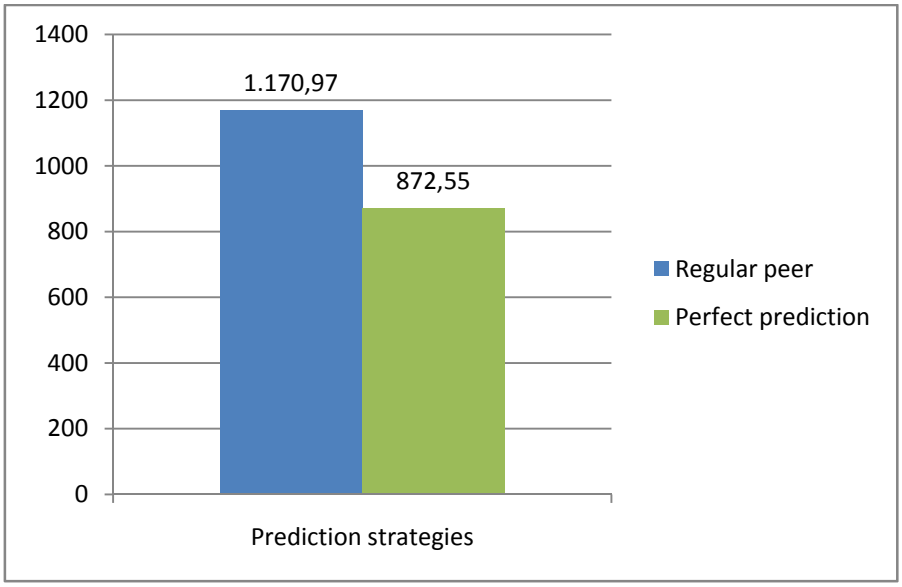

Figure 4-14. Optimistic Unchoke with perfect download price prediction.

Seeing the obtained results we can state that the download price supposes a great advantage for the peer that has that knowledge. Our strategy uses Download Price, which indicates how profitable is downloading from a peer, to optimize the optimistic unchoke algorithm. As we can recall, one of the goals of the optimistic unchoke algorithm is to discover better peers to download from. Thanks to Download Price the algorithm can refine its search instead of assigning the optimistic slot randomly. This way, we can establish faster contact with better peers and, therefore, improve our download time.

However, these results are not realistic. In these experiments we are assuming that we have a global knowledge about the state of the swarm. This global knowledge can only be recreated in simulated environments. In real environments we can only predict this value by using a system like the Download Price Prediction we designed this section.

4.7.2. Test 2: Real download price prediction

In the first test we proved the usefulness of the download price value in BitTorrent environment. For that reason, we simulated an ideal system which does not have prediction error. However, the performed simulations are made to test the value usefulness and not our prediction strategy. In this test we will focus on studying the proper operation of our Download Price predictor, and how affects to the performance the average 5% prediction error of our trained system.

For this second test we have performed an extra simulation. In this simulation we have monitored a peer under the same conditions as in test 1. However, this time the peer uses our optimistic unchocke modification with the Download Price Prediction system that we have built and trained through section 4.

Figure 4-15 shows the results of this extra simulation together with the ones obtained in the first test. In the previous test, where predictor peer simulate a perfect prediction, we highlight a decrease of 25.48% in the download time. Using our Download Price Predictor, the peer takes 105.58 seconds more than the perfect prediction approach to complete its download. Even so it is 192.84 seconds faster than regular peers. This time decrease represents an improvement of 16.46%, only a 9.02% below the best possible result.

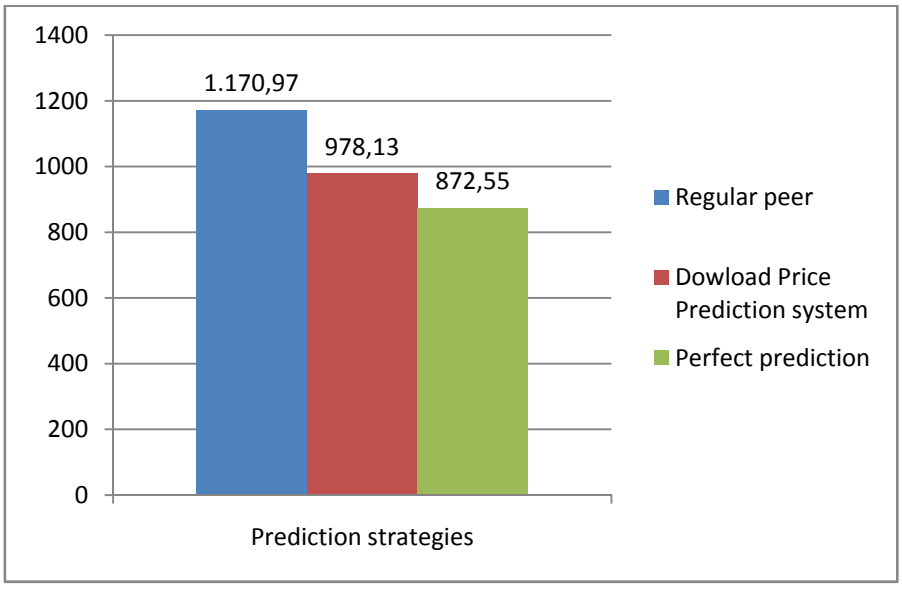

Figure 4-15. Optimistic Unchoke with Download Price Prediction strategy.

During the construction process of our price prediction system we have used techniques such as Cross-validation to determine its accuracy. After performing the training process on our system, it showed a prediction error of 5.55%. This error causes that in certain situations the optimistic unchoke algorithm does not assign to its slot the most optimal peer. As a result the system cannot maintain the same performance as the ideal case. Even so, the obtained improvement over a regular peer is still very high.

The approach we have evaluated uses download price to decide which peer is assigned to the optimistic slot. However, this value can be used in many ways to optimize BitTorrent rechoking operations. An example of how can be these optimizations is BitTyrant [15]. It must be taken into account that most of these approaches are based in a free-rider behavior. These kinds of behaviors are known for having a negative impact over the rest of swam, which is not the case of our approach (Figure 4-16).

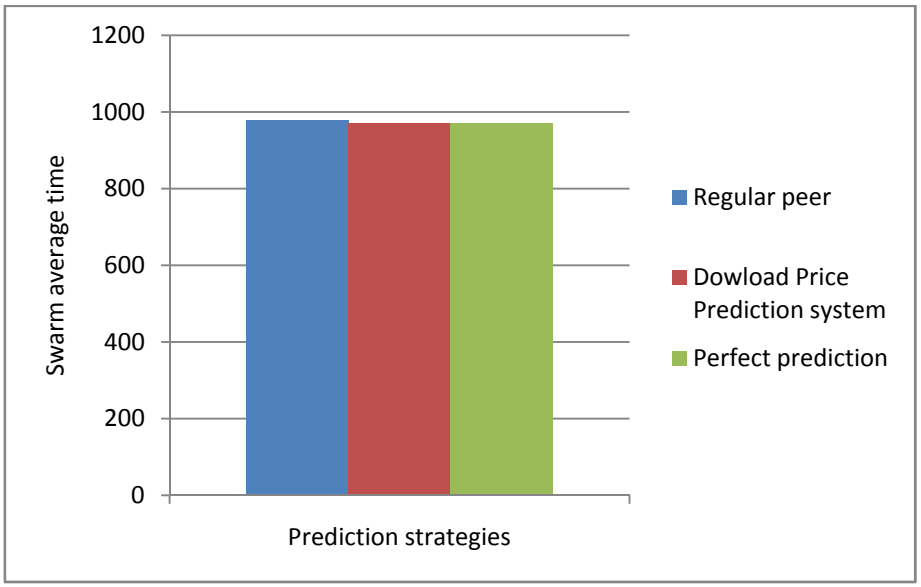

Figure 4-16. Average download times of swarm peers.

4.7.3. Test 3: Extremely bad download price prediction

In the previous test we saw how the Download Price Prediction system we have implemented affects to the download time of peers. The results showed that with an average error of 5% the performance already suffers. In this last test we want to investigate what would happen if our price prediction system had an unusually bad operating and errs in most of its predictions. We can see it as the opposite side of the perfect prediction.

Following this objective, we have included some changes in our Download Price Predictor. We have altered the dataset that our system uses as experience. In this way our system will not be able to make predictions correctly. With these modifications we have performed several simulations under the same environment that we used in the previous tests. Results are showed in Figure 4-17.

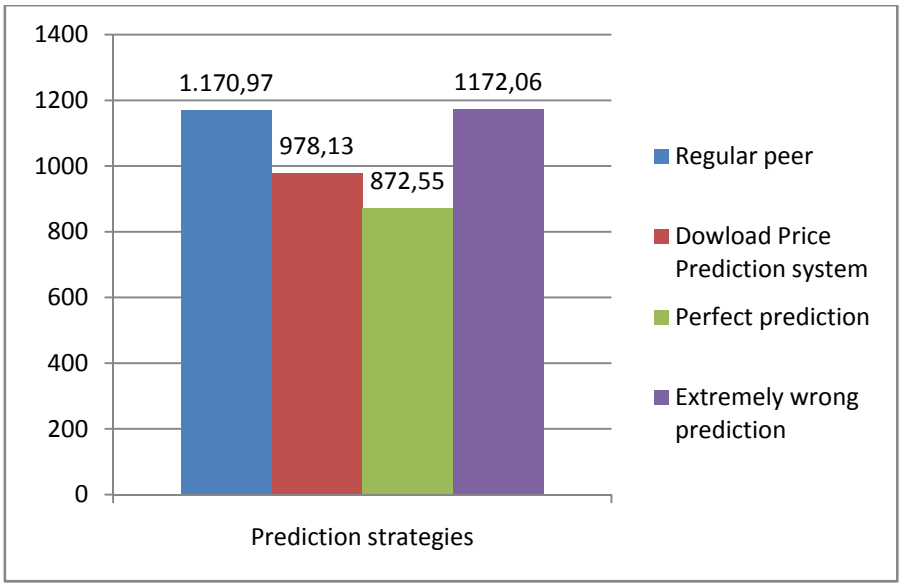

Figure 4-17. Optimistic Unchoke with totally wrong predictions.

As we can see when we simulate extremely poor predictions, the system loses any performance improvement. The obtained Download time is almost the same as any other regular peer (1.09 seconds higher). This phenomenon occurs because our Download Price predictor is conducting value predictions based on false data. As these data have no connection with the scenario where the simulation is being performed, the predicted values have no relation with the real download prices values. In other words, our predictor is behaving like a random function. Therefore, we are randomly choosing which peer is going to occupy the optimistic slot. This is exactly what the original Optimistic Unchoke algorithm does, and explains why this approach has the same download time as regular peers.

5. Collaborative Strategy

In this section we will focus in developing a collaborative strategy that can be used as a support for our Download Price prediction strategy. This collaborative strategy is focused on creating a group of peers interested in the same digital content. We know this coalition as Crowd. The idea is to achieve that all the components of the group behave as a single entity. To achieve this objective we have designed two different approaches. On the one hand, a new piece selection strategy (Group Piece Selection) which will urge peers to download the pieces according to the group needs. On the other, an improvement of unchoke policy (Crowd Members First) that will allow group members get priority when they download pieces between them.

In the following sections we focus on explaining step by step how this collaborative strategy has been built. In Section 5.1 we describe how Group Piece Selection strategy has been implemented and we show its simulation results. In Section 5.3 we do the same process with Crowd Member First strategy.

5.1. Group Piece Selection strategy

Group Piece Selection strategy is an improvement of BitTorrent rarest first algorithm that operates over a group of nodes. In this strategy collaboration is achieved in the way that pieces are downloaded. The main idea is that Crowd peers cooperate by selecting pieces that other members do not have. However, this is not always possible and eventually peers will be forced to get pieces that the group already has (we know this as a collusion). When this occurs we use a different method to select pieces, but keeping the same collaborative philosophy. Peers then select the least replicated piece within the group. We note that a peer should never waste the opportunity to download a piece, in other case it would be wasting its bandwidth and this would be counterproductive. General guidelines of Group Piece Selection can be seen at Figure 5-1.

```
1. Select pieces that the whole Crowd does not have.
2. In case multiple matches in 1, resolve with rarest first policy.
3. If no pieces found in 1, select least replicated pieces
4. In case of multiple matches in 3, resolve with rarest first policy.
```

Figure 5-1. Group Piece Selection strategy guidelines.

The objective of this behavior is to achieve a distributed copy of the content as soon as possible. As we recall a distributed copy is a full copy of the content shared among the group peers. The distributed copy exists adding up all the pieces that each peer has achieved individually. Once the distributed copy is completed we can state that members of the Crowd will have guaranteed the completion of the shared content.

In the following sections we will describe the necessary changes on Piece Selection algorithm to implement our Group Piece Selection algorithm.

5.1.1. Group formation

The first step towards the implementation of the collaborative strategy is to determine how to establish the group of peers which we refer to as Crowd. To form the group we considered that a previous agreement between peers has been made. Thus, we assume that group has been pre-established and it will not change during the life cycle of a BitTorrent simulation.

Regarding the simulation, we need that each peer known the rest of the group. With this purpose in mind we have created a configuration file with the necessary information. An example of this file is showed in Figure 5-2. This configuration file only contains peers identifiers and indicates to which group they are linked.

```
#Crowd config file

#Crowd group id(number), BTPeer(id)
1,BTPeer(10)
1,BTPeer(11)
1,BTPeer(12)
1,BTPeer(13)
```

Figure 5-2. Crowd config File Example.

5.1.2. Gathering Distributed Copy information

In order to put in practice the Group Piece Selection strategy, we must be able to know which pieces the distributed copy has at any instant. In other words, any peer in a Crowd group should have a way of knowing which pieces the other members have. BitTorrent protocol has mechanisms to inform peers which pieces the others peers have or have acquired recently. These mechanisms consist in messages that the peers send in certain situations to inform about their state. The information about pieces is handled by BITFIELD and HAVE messages, more details about these messages can be found in the Background section.

Thanks to these messages, each peer inside a swarm can maintain a local database that stores which pieces other peers have. This database consists of a large set of boolean arrays, in which each one of these arrays contains information about the pieces that a single peer has. Obviously, this set of arrays is updated each time that a BITFIELD or HAVE message is received.

With these mechanisms we can easily find out which parts have each Crowd member. However, checking this set of arrays each time that a piece has to be selected can be costly and very inefficient. To solve this problem we have added a new structure to the array set of group members. This structure consists of a list of integers, in which each list position represents a distributed copy piece. Each time a Crowd member notifies that has new pieces

we add +1 to corresponding list piece position. So, having a zero in a certain position means that anyone in the group has this piece; a one means that one member has the piece; a two means that two members have the piece, and so on. We call this structure Crowd Piece Counter.

Using this process a Crowd peer will be able to know if the group has a certain piece in no time. In addition, a peer can also know if a piece of the distributed copy is owned by more than one member. With this information a peer may easily determine which pieces are less replicated in the distributed copy. This property will be of particular importance in further sections.

5.1.3. Modeling Group Piece Selection algorithm

In the previous section we have explained how to gather the information used by our algorithm. Now that we have the necessary data properly stored, we can proceed to detail how the Group Piece Selection policy is implemented.

Group Piece Selection is a variation of the original Piece Selection algorithm used by all BitTorrent clients. This can be easily seen by comparing the pseudocodes of each strategy contained in Figure 5-1 and Figure 5-3.

```
//First we check which pieces interest us
otherPiecesList = check which pieces the other peer has;
myPiecesList = check which pieces I have;
filteredPiecesList = otherPiecesList – myPiecesList;

//Choose among various pieces with rarest first strategy
Sort the pieces in filteredPieceList due its rarity;
RETURN First piece in the list;
```

Figure 5-3. Original Piece Selection pseudocode.

The original Piece Selection algorithm, which is showed in Figure 5-3, has two main steps:

- Identify which pieces the peer does not have.
- Choose one piece using the rarest first strategy.

In the first phase, the peer who calls the algorithm compares its pieces to the peer who is offering its bandwidth. This can be easily done because each peer keep stored which parts the other peers have. This way a peer identifies in which parts he is interested. During the second phase, peers decide which one of the interested pieces is going to choose. For this process they use rarest first algorithm, which selects the piece that has the least amount of copies. This algorithm is possible because each peer also maintains a counter of copies other peers have.

Our approach will maintain these two phases. Nevertheless, it changes the concept of which pieces are interesting for Crowd peers. Peers using this strategy are interested in download pieces that other group members does not have. In other words, interesting pieces are those that help to complete distributed copy. Therefore, to implement our approach we

must add some extra piece filtering that discard less interesting pieces for the Crowd and for the distributed copy as well. Finally, rarest first policy is used to choose a piece from all which are interesting. A Group Piece Selection pseudocode can be seen in Figure 5-4.

```
//First we check which pieces interest us
otherPiecesList = check which pieces the other peer has;
myPiecesList = check which pieces I have;
filteredPiecesList = otherPiecesList – myPiecesList;

//Get the most interesting pieces for the crowd.
crowdInterestPieceList = getCrowdInterestPieces(filteredPieces);

//Choose among various pieces with rarest first strategy
Sort the pieces in crowdInterestPieceList due its rarity;
RETURN First piece in the list;
```

Figure 5-4. Group Piece Selection pseudocode.

The significant part of our strategy is the method that determines which parts are most important for the group. In the pseudocode in Figure 5-4 we call it *getCrowdInterestPiece.* This method implements the policy of our Group Piece Selection strategy. This policy basically consists of selecting the pieces that other group components do not have. In other words, it selects those pieces that distributed copy is missing. However, it may happen that no piece meets this condition. If the group already has all the pieces that are available to download, peers then will focus on selecting less replicated pieces. This means that peers will look for pieces had by only one group member. If still there are no pieces meeting this condition, peers will look for pieces had by two members, and so on.

```
PROCEDURE getCrowdInterestPieces(LIST availablePieceList)

replicationLevel = 0;

WHILE crowdInterestPieceList.size == 0 DO
    FOR i=0 to availablePieceList.size DO
          pieceNum = availablePieceList.get(i);
          IF crowdPieceCounter[pieceNum] == replicationLevel THEN
                crowdInterestPieceList.add(pieceNum);
          END IF;
    END LOOP;
    replicationLevel++;
END LOOP;

RETURN crowdInterestPieceList;

END PROCEDURE;
```

Figure 5-5. Pseudocode of getCrowdInterstPieces method.

Figure 5-5 shows *getCrowdInterestPieces* method implementation. This method receives as input the available pieces to download, and returns a list of the most interesting pieces for the group. As we can see on Figure 5-5 pseudocode, we check how many group members have each one of these available pieces. On that basis, we establish levels according to how replicated a piece is (Replication Level). *CrowdPieceCounter* array is the one that gives us information about how many group members have each piece, this structure was introduced in Section 5.1.2 (Gathering distributed copy information). First we look for pieces with a level zero of replication, this means that distributed copy does not have this piece. We add all found pieces in the *crowdInterestPiece* list. If after checking all available pieces we have not added any to *crowdInterestPiece* list, we increase the replication level and perform the same process again. We'll keep doing this until we find at least one piece. Finally, the method will return *crowdInterestPiece* list containing all pieces with this replication level.

With this method we have just described all the improvements that incorporate the Group Piece Selection strategy. In the following sections we will focus on performance analysis and validation of system we just defined.

5.2. Group Piece Selection validation

We have designed and performed a series of experiments to evaluate the proper operation and effectiveness of our Group Piece Selection strategy. As in previous validations, these experiments were conducted in a discrete even-based peer-to-peer simulator GPS [23].

To perform the Group Piece Selection tests we are going to recreate a BitTorrent content life cycle. This is because we want to run our strategy on a real environment in which peers are entering the swarm with a certain frequency. In order to do this we are going to reproduce just born swarm scenarios. Our scenario will be a single seed and total of 30 peers will join the swarm following the equation 1 described by Guo et al. in [24].

In order to have realistic scenarios we have categorized our peer model using two upload capacities. These capacities are 0.2 Mb/s for regular peers and 1.0 Mb/s for high capacity peers. As it concerns download capacity, it is linked to upload capacity and its value it is expected to be ten times higher. Other simulation parameters are:

Torrent parameters:

- File size: 50 MB
- Pieces: 196
- Peers: 30
- Seeds: 1

Clients are configured as:

- 4 unchoke slots + 1 optimistic
- Upload: 0.20 / 1.00 Mb/s
- Download: 2.0 / 10.0 Mb/s

5.2.1. Test 1: A Crowd group in a homogeneous swarm

In this first scenario we want to show how our Group Piece Selection strategy behaves in a swarm with other peers. To provide a fair comparison between regular peers and crowd peers, we have set a homogeneous environment where all peers have the same upload capacity: 0.20Mb/s. However, the initial seed is an exception. We have assigned a greater capacity to this peer: 1.00 Mb/s. The purpose of this variation is to speed up the distribution of pieces in the earliest moments of the content life cycle and to achieve greater pieces diversity on the first joining peers. We should note that during these earliest moments initial seed is the only peer who distributes pieces.

We have run two different simulations under the same environment conditions for this first test, one with crowd group and the other without it. In the first simulation we have created a group of 3 peers who use the Group Piece Selection strategy. The remaining 27 swarm components are regular peers. In the second simulation we have not entered any group and the 30 swarm components are regular peers. The objective of this simulation is to achieve a basis for comparing the results obtained in other simulations.

Download times for both simulations are shown in the following tables. Table 5-1 contains the results of the simulation without crowd and Table 5-2 shows the results of three-member group simulation. In both tables peers are sorted by the time they joined the swarm. We want to emphasize the fact that peers who joined the swarm earlier have higher download times, while the times of late joining peers are lower. This is a consequence from peers finishing their download and become seeds. This higher presence of seeds is the cause of this decrease in download times.

In both tables we have bolded the three peers involved in the group strategy. Comparing their simulation download times, we note that average times of group members have decreased by 7.4 seconds. This represents an improvement of 0.43% which is negligible. We expected this result because Group Piece Selection strategy is focused on achieving the distributed copy as soon as possible, and do not cover individual improvement for the peers that compose it. Individual peer improvement for crowd members will be addressed in next section with the Crowd Members First strategy.

```
   Time elapsed BTPeer(2): 1640,69 sec      Time elapsed BTPeer(17): 1693,51 sec
   Time elapsed BTPeer(3): 1659,92 sec      Time elapsed BTPeer(18): 1671,23 sec
   Time elapsed BTPeer(4): 1638,76 sec      Time elapsed BTPeer(19): 1708,2 sec
   Time elapsed BTPeer(5): 1785,36 sec      Time elapsed BTPeer(20): 1666,7 sec
   Time elapsed BTPeer(6): 1710,3 sec       Time elapsed BTPeer(21): 1639,94 sec
   Time elapsed BTPeer(7): 1766,39 sec      Time elapsed BTPeer(22): 1629,31 sec
   Time elapsed BTPeer(8): 1728,49 sec      Time elapsed BTPeer(23): 1585,27 sec
   Time elapsed BTPeer(9): 1649,18 sec      Time elapsed BTPeer(24): 1567,48 sec
   Time elapsed BTPeer(10): 1749,73 sec     Time elapsed BTPeer(25): 1506,08 sec
   Time elapsed BTPeer(11): 1746,67 sec     Time elapsed BTPeer(26): 1457,72 sec
   Time elapsed BTPeer(12): 1685,50 sec     Time elapsed BTPeer(27): 1386,52 sec
   Time elapsed BTPeer(13): 1767,52 sec     Time elapsed BTPeer(28): 1281,54 sec
   Time elapsed BTPeer(14): 1687,5 sec      Time elapsed BTPeer(29): 1077,67 sec
   Time elapsed BTPeer(15): 1797,69 sec     Time elapsed BTPeer(30): 711,78 sec
   Time elapsed BTPeer(16): 1757 sec        Time elapsed BTPeer(31): 207,2 sec

Total 30 downloads, average download time 1552.02 sec.
Peers 10, 11 and 12 average download time 1727.30 sec.
```

Table 5-1. Peers download times: simulation without a Crowd group.

```
   Time elapsed BTPeer(2): 1667,36 sec      Time elapsed BTPeer(17): 1744,85 sec
   Time elapsed BTPeer(3): 1584,62 sec      Time elapsed BTPeer(18): 1697,92 sec
   Time elapsed BTPeer(4): 1758,60 sec      Time elapsed BTPeer(19): 1703,12 sec
   Time elapsed BTPeer(5): 1780,31 sec      Time elapsed BTPeer(20): 1648,21 sec
   Time elapsed BTPeer(6): 1778,73 sec      Time elapsed BTPeer(21): 1659,21 sec
   Time elapsed BTPeer(7): 1741,14 sec      Time elapsed BTPeer(22): 1658,68 sec
   Time elapsed BTPeer(8): 1647,94 sec      Time elapsed BTPeer(23): 1590,78 sec
   Time elapsed BTPeer(9): 1703,00 sec      Time elapsed BTPeer(24): 1583,42 sec
   Time elapsed BTPeer(10): 1714,83 sec     Time elapsed BTPeer(25): 1498,69 sec
   Time elapsed BTPeer(11): 1822,90 sec     Time elapsed BTPeer(26): 1461,34 sec
   Time elapsed BTPeer(12): 1622,66 sec     Time elapsed BTPeer(27): 1359,83 sec
   Time elapsed BTPeer(13): 1678,02 sec     Time elapsed BTPeer(28): 1268,58 sec
   Time elapsed BTPeer(14): 1659,13 sec     Time elapsed BTPeer(29): 1095,8 sec
   Time elapsed BTPeer(15): 1757,78 sec     Time elapsed BTPeer(30): 704,29 sec
   Time elapsed BTPeer(16): 1728,21 sec     Time elapsed BTPeer(31): 207,2 sec
                                            Time of DC           : 917,14 sec

Total 30 downloads, average download time 1550.91 sec.
Crowd members average download time 1719.90 sec.
```

Table 5-2. Peers download times: simulations with 3 members Crowd

In table 5-2 we have also bolded the download time of distributed copy (DC). This indicates how much time has needed the Crowd to complete the distributed copy. We can appreciate that distributed copy is 802 seconds faster than the average download time of group members. That means an improvement of 46.67% over the average time of the Crowd. With these results we can state that, in this scenario, crowd peers have ensured the completion of their downloads when they reach about 54% of their download progress. This is because, at this point, among them they already have all the pieces of the digital content.

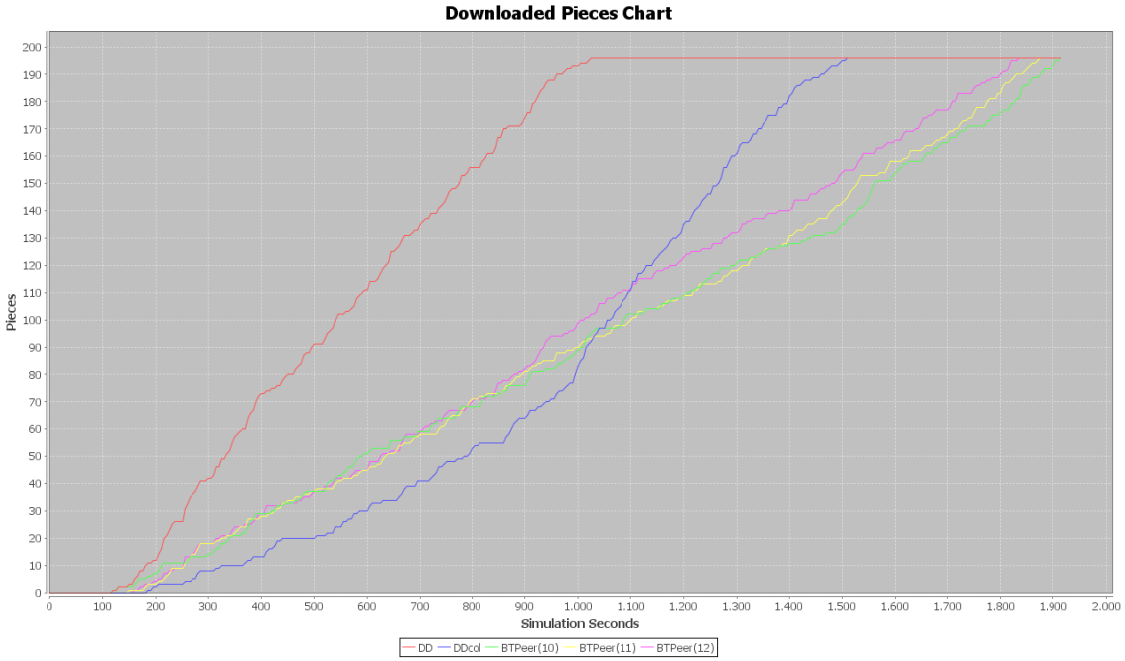

Figure 5-6. Crowd group download progress.

In Figure 5-6 we can see the download progress of all elements involved in our strategy. On one hand we have peers 10 (green), 11 (yellow) and 12 (pink) piece download progress. As we can appreciate, their pieces download rate is very similar, at least until the distributed copy finishes. This leads us to assume that they are contributing equally to the distributed copy achievement. Besides that, we cannot extract more information from group components.

On the other hand, we have information about distributed copy (red and blue lines). Distributed copy download progress (red) confirms the improvement of 46.67% over group member's average times showed in Table 5-2. However, the most interesting fact is the behavior of collisions (blue). This line increases every time a peer downloads a piece that already has one of the members of the crowd. As we can see, the number of collisions remains low while the distributed copy is incomplete. But when it reaches a certain download percentage amount (around 90%), the number of collisions increases greatly. This behavior means that peers of the Crowd are trying to avoid collisions by downloading pieces the other members do not have. Of course, at a certain point this is not longer possible because distributed copy is completed or nearly completed. Then peers have no choice but to get repeated pieces. This behavior proves that our strategy works properly and focuses the group's efforts to complete the distributed copy.

5.2.2. Test 2: Influence of piece availability over crowd groups

In the first test we focused on checking the proper operation of our Group Piece Selection strategy. For the following experiments we will focus on evaluate situations that may affect the download time of the distributed copy. In this second test, we are going to study how affects to the Crowd group a low availability of pieces in the swarm. As it was in the previous simulation, we have set a homogeneous environment where all peers have 0.20 Mb/s as

upload capacity. Once again the initial seed is the exception. This time, we will use different upload capacities for each performed simulation.

For this second test, we have performed three simulations under the same scenario. These simulations include a group of 3 peers who use the Group Piece Selection strategy, and the 27 remaining swarm components are regular peers. However, we have used initial seeds with different upload capacities for each simulation: 0.20 Mb/s, 0.60 Mb/s and 1.00 Mb/s. By modifying upload capabilities we can limit the piece distribution capacity of initial seeds. Thus, we can control the amount and diversity of pieces present in the swarm when Crowd members join.

When the initial seed has only an upload capacity of 0.2 Mb/s the distributed copy is so negatively affected that makes our collaborative strategy useless. As we can see in Figure 5-7, Crowd peers finish their downloads almost at the same time as distributed copy. To be more precise, distributed copy is 45 seconds faster than group members' average. On the current simulation, this represents only an improvement of 1.38%. This poor performance is not only caused by low availability of pieces. The fact that initial seed has the same bandwidth as the rest of peers in the swarm causes a bottleneck effect. With so many peers struggling to download from initial seed is very difficult to get rare pieces. Therefore, peers end up downloading the most replicated pieces because they are the only available ones. The high collisions level (blue line) in Figure 5-7 is proof of that.

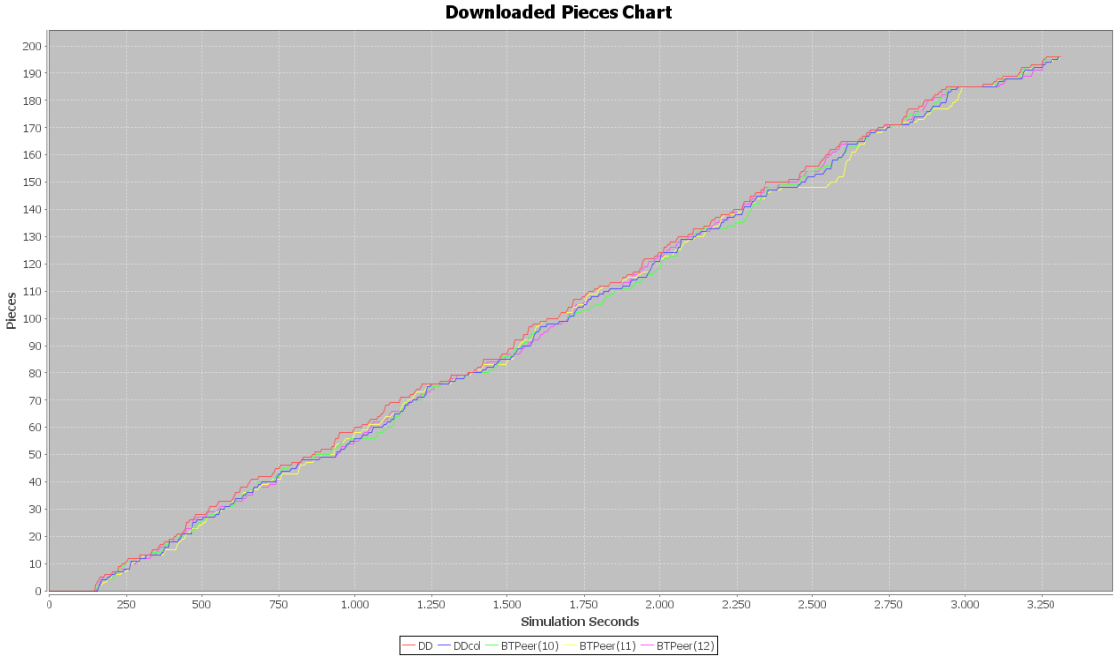

Figure 5-7. Seed Upload 0.2 Mb/s.

When we perform simulations where initial seeds have a higher upload capacity this effect is dissipated. In Figure 5-8 initial seed has a bandwidth three times greater (0.60 Mb/s). In this case the improvement is a 14.32% over the group average. Finally, in Figure 5-9 we can see the results of a test with an initial seed with a much greater capacity than the rest: 1.00 Mb/s Figure 5-9 shows that distributed copy (red line) have recovered its normal performance.

Checking at the simulation results, we can appreciate that the improvement over the group averages has amounted to 47.67%. Regarding the number of collisions (blue line), it remains stable until the completion of the distributed copy. In this simulation, the pieces availability problem has disappeared and the strategy works properly.

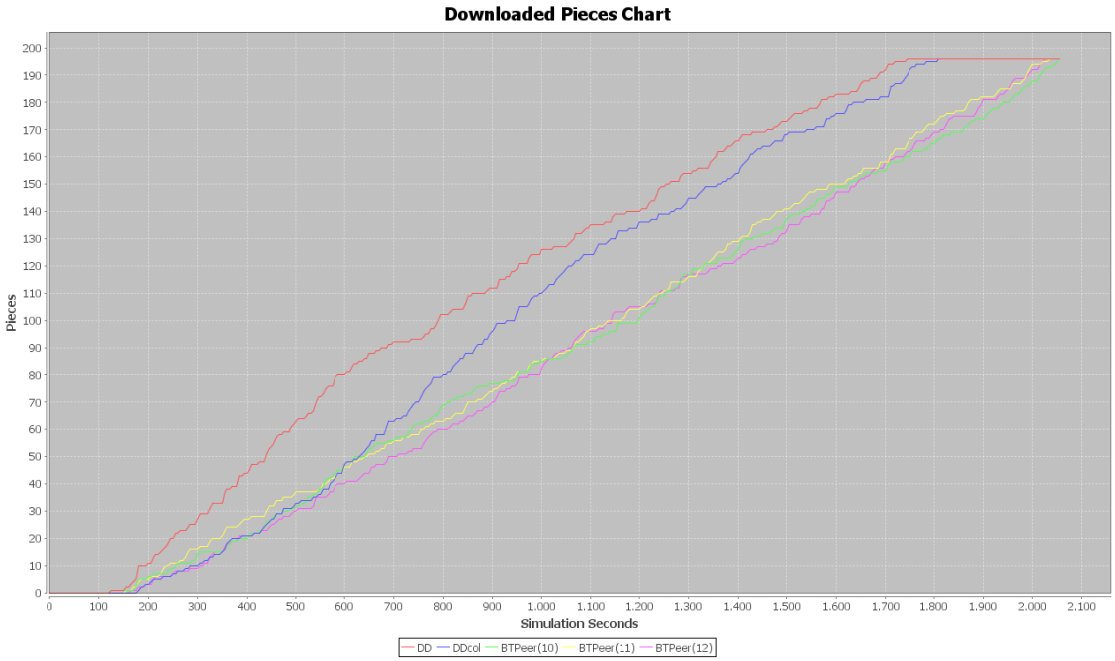

Figure 5-8. Seed Upload 0.6 Mb/s.

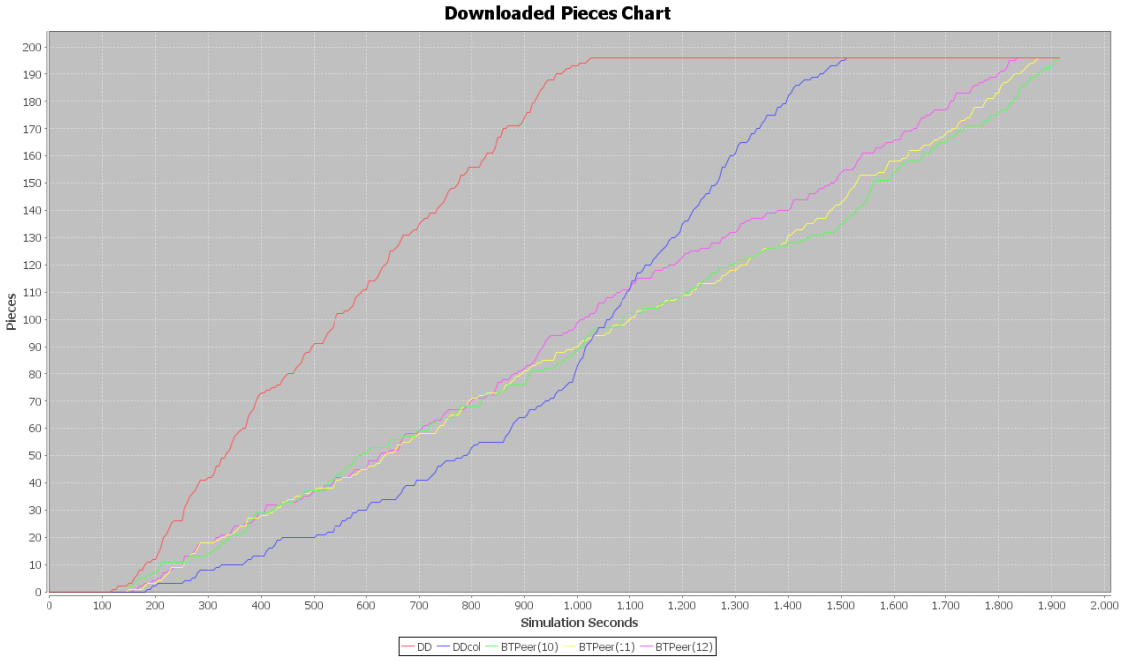

Figure 5-9. Seed Upload 1.0 Mb/s.

In this second test we have seen that Group Piece Selection strategy works fine. However, it is necessary that Crowd peers can opt to a certain pieces variety for the strategy brings results. If there is no diversity peers can only download pieces that their companions already have. Although that generate collisions, is a better option than staying idle waiting for a piece that can benefit the group.

5.2.3. Test 3: Group size influence over distributed copy

In the previous experiment we started assessing which factors affect the downloading time of the distributed copy. In the current experiment we will continue this process by focusing on other variables. This third test is focused on studying how the size of Crowd groups affect to the distributed copy performance. Like in other simulations, we want to provide a fair comparison between regular peers and crowd peers. We have set a homogeneous environment where all peers have the same upload capacity: 0.20Mb/s. The initial seed is an exception; in order to be able to supply the demand of pieces to the swarm, we have assigned a greater bandwidth to it: 1.00 Mb/s.

In this third test we have run five simulations. For each simulation we have introduced a group with a different size. We are investigating groups with sizes of: 2, 3, 4, 5 and 6 members. Regarding the regular peers, if the group size is *k*, there will be 30 – *k* regular peers. In Figure 5.10 we show crowd components download times for each of these configurations.

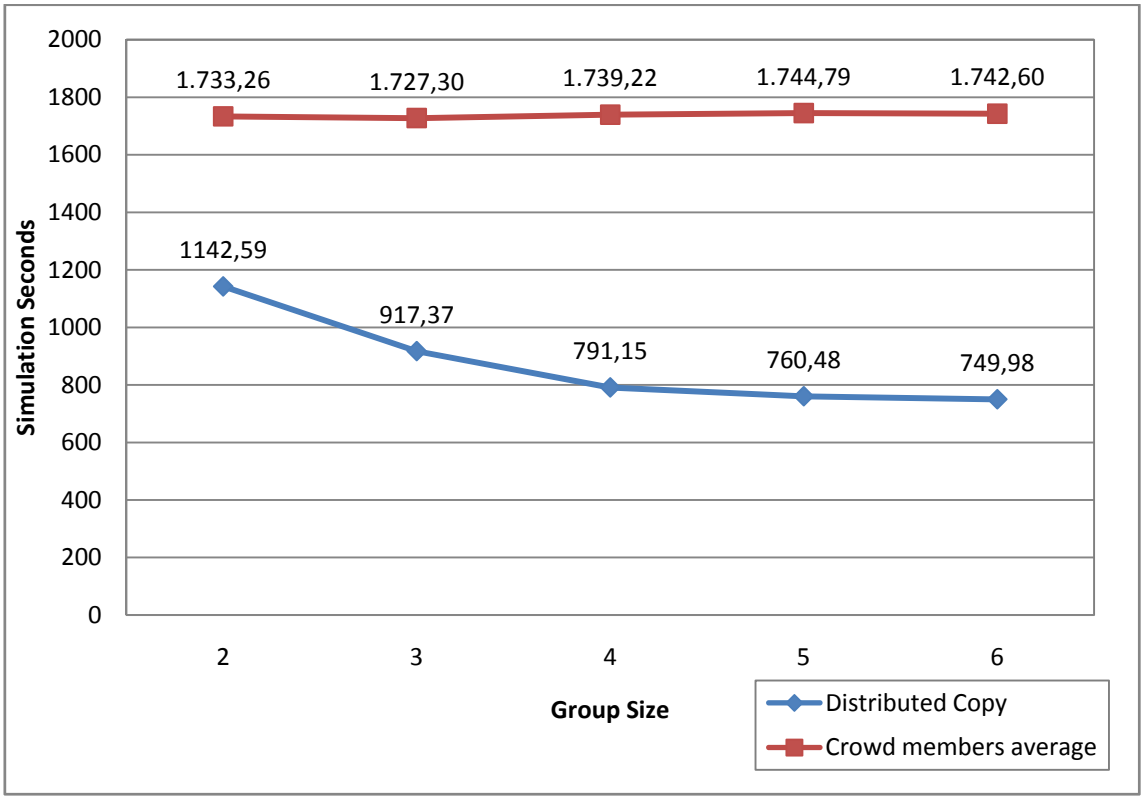

Figure 5-10. Crowd download times.

We can see significant variations in distributed copy download time when there is an increment in Crowd group size. The greater the size of the group is, the better the performance of the distributed copy. For a group of 2 members the download time is 1142.59 seconds. However, a group of 6 members can complete the distributed copy 392.61 seconds faster. This represents an improvement of 34.32%. If we take the average download time of all Crowd peers as a reference, the distributed copy with better performance (group with 6 peers) finishes 992.62 seconds faster. This means that group members are on average at 42.86% of their progress when the distributed copy finishes. This decrease in the download time of the distributed copy is perfectly logical. This phenomenon is due to the Group Piece Selection strategy: since there are more peers in the group, each one downloads a smaller portion of the distributed copy. As all peers download in parallel, distributed copy is achieved sooner.

However, the performance increase in the distributed copy is not proportional to the size of the group. The increase when we use a group of three members instead of two is a 19.71%. If we include one more member (group size 4) the performance increases 11.05%. From this point on the time of the distributed copy does not show any significant improvement. By using groups of 5 and 6 members distributed copy time is only reduced 30.67 seconds (2.71%) and 10.5 seconds (0.92%). To analyze this phenomenon we are going to see the download progress in crowds of 3, 4 and 5 members (Figures 5-9, 5-11 and 5-12).

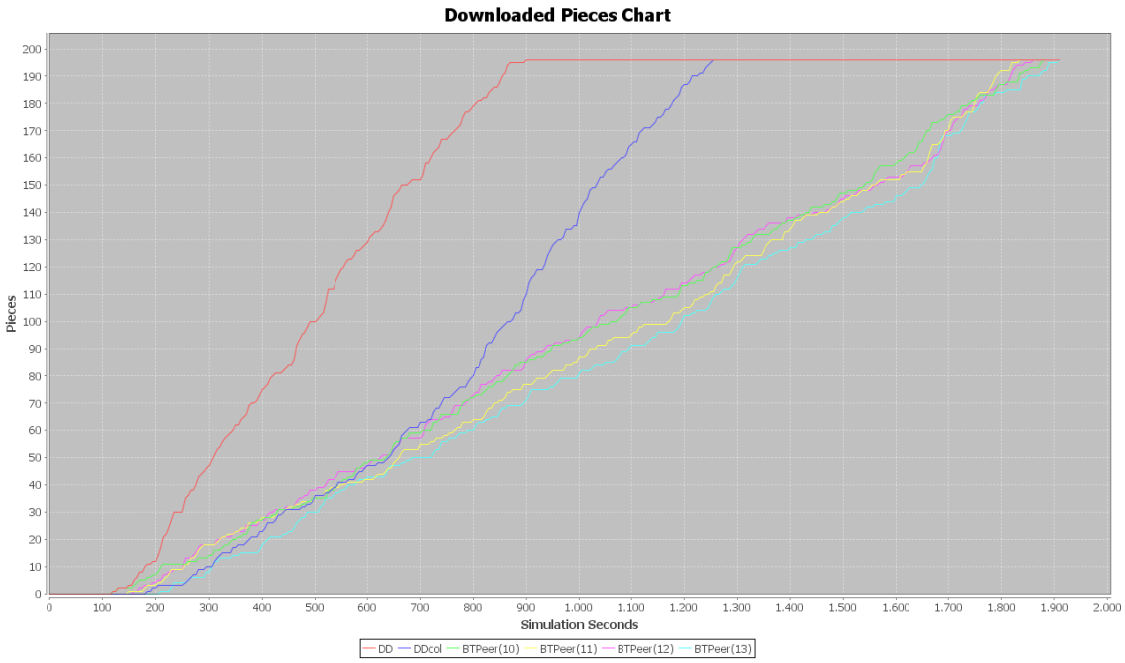

Figure 5-11. Crowd of four members

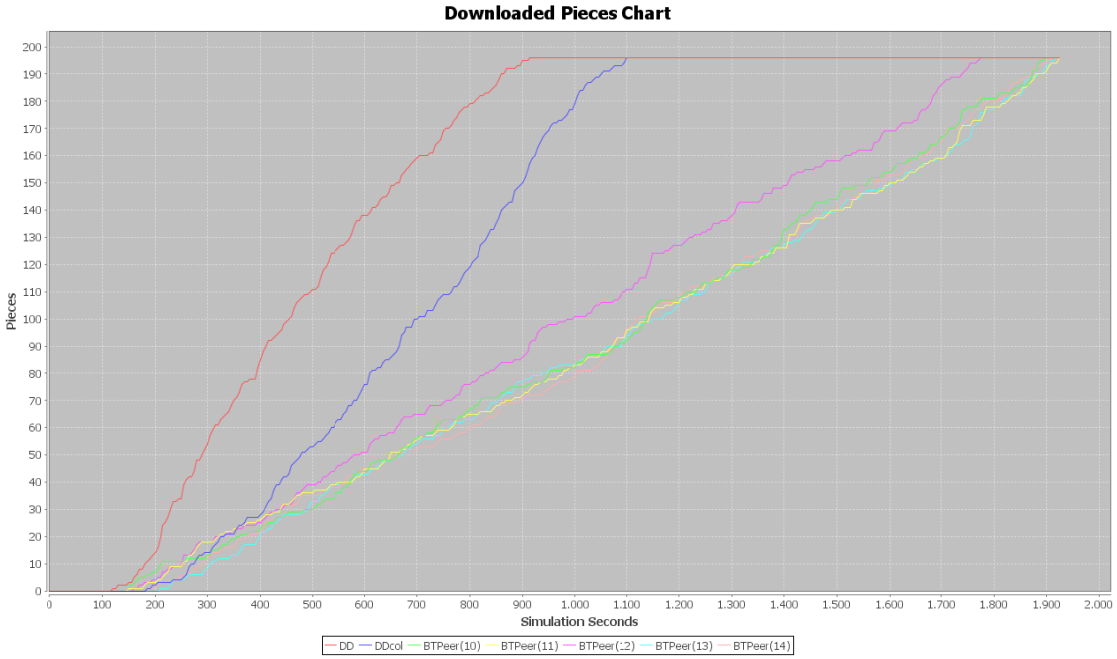

Figure 5-12. Crowd of five members

Regarding the download progress of Crowd components, we can appreciate a large increase in collisions (blue lines) each time we increase the group members. The number of collisions becomes critical when we form groups bigger than 5 members (Figure 5-12). In groups of this size, collisions number grows almost at same rate as distributed copy (red line). However, this phenomenon begins to show when crowds of 4 members are used (Figure 5-11). According to previously analyzed performance (Figure 5-10), the distributed copy barely improve when we use groups bigger than four members. These results lead us to affirm that this lack of improvement in the distributed copy is caused by the increase of collisions.

We can study in detail which Crowd peers are generating this increase of collisions. Thus, we can rule out any abnormalities in the strategy operation or in the peers' behavior. In Figure 5-13 we can see each peer contribution to the completion of the distributed copy and which percentage of downloaded pieces are collisions.

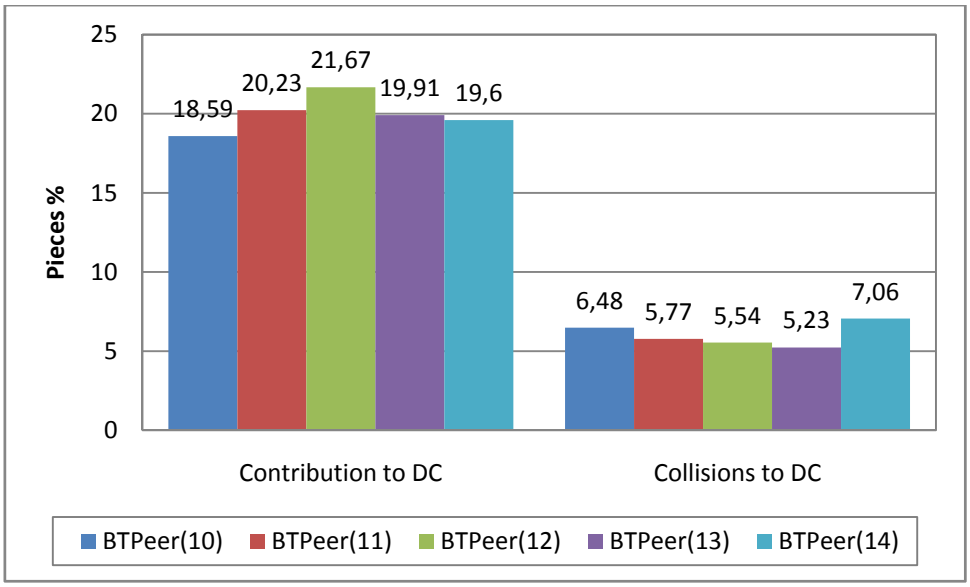

Figure 5-12. Peers contribution to Distributed Copy

Figure 5-13 shows, for a five members group, that each Crowd peer contributes about 20% to the distributed copy (first column). Furthermore, each member does a similar percentage of collisions (about 6%). This means that all Crowd peers are contributing the same in the achievement of the distributed copy, and there is not a single peer who is taking advantage of their companions.

The collision increase that we have seen in this test occurs because not all peers in the swarm have the pieces that distributed copy misses. As group members should also consider their own download completion, they are forced to download pieces that the group has in order not to waste their bandwidth. This situation worsens every time we increase the size of the group. This is because exist the same piece availability but there are more Crowd members looking for pieces for the distributed copy. Hence, there is a greater probability of downloading repeated pieces. In an ideal environment, where other peers in the swarm always had the pieces that distributed copy misses (e.g. seeds), we could increase the group size without producing this stagnation in distributed copy performance. However, for the present scenario it is evident that the best relation between size and distributed copy download time is a four members group.

5.2.4. Test 4: A Crowd group saving the swarm

All BitTorrent swarms have a life cycle. A birth, in this period swarm is created and a single initial seed has all the content pieces. A boom period when the content is popular and many peers join the swarm to download it. At some point the swarm size becomes constant (Stable period). Fall period, in which the content is no longer interesting. During this period very few peers join and seeds, little by little, leave the swarm. Finally, when a piece totally disappears because the seed which have this piece last copy leaves, we can state that swarm has dead. At this stage any peer in the swarm will not be able to complete its content. That is because at least one of the pieces is unreachable. Figure 5-13 shows the representation of a swarm life cycle.

As we have seen in preceding tests, distributed copy can be completed when Crowd peers are about 43% of their download progress. This can be used in many ways, like ensure the content completion for all Crowd members in a relative short time. In this test we want to probe that using of a Crowd can help to restore a swarm that it is already dying.

For this test we are going to simulate the whole life cycle of a BT swarm. As we indicated at the beginning of this section, peers join the swarm following the equation 1 described by Guo et al. in [24]. This grants that peers will join with a frequency according to the swarm lifecycle. Moreover, the peers will leave the swarm at the time they complete their downloads and become seeds. In this way, we can simulate an accelerated fall and death of the swarm. Initial seed is a special case. It will leave the swarm at second 1000 of simulation time. This will give enough time to joining peers to get enough pieces.

For this forth test we are going to perform two experiments. In the first experiment all peers in the swarm will be regular peers. In the second, there will be a group of four members and the rest will be regular peers. We have choose a four members Crowd because in previous tests have achieved better results. The goal of these experiments is to study in which instants peers cannot finish their downloads because of swarm's decay.

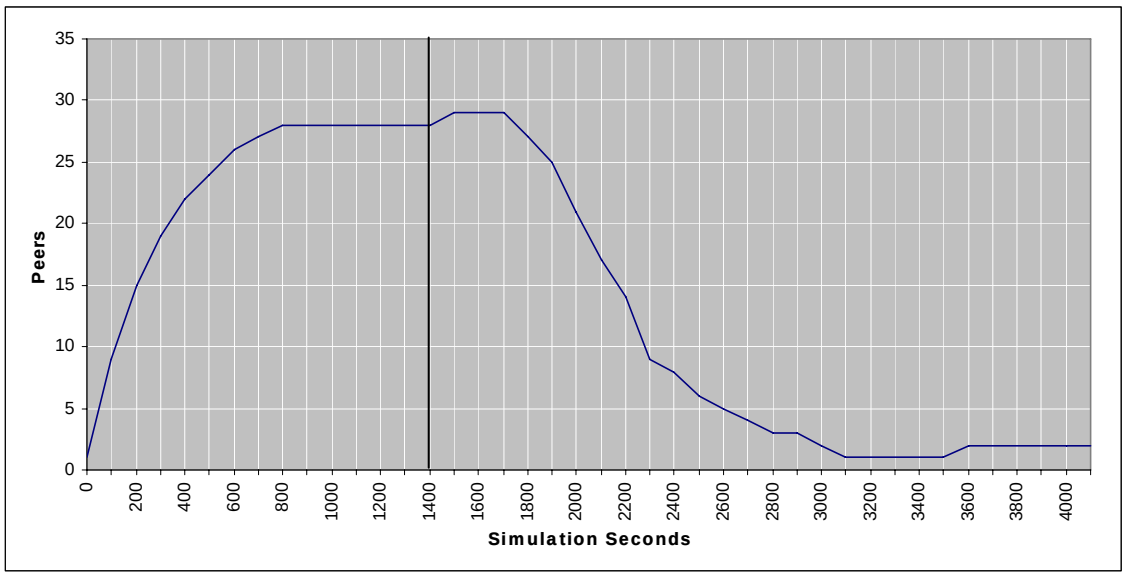

Figure 5-13. Swarm lifecycle.

In the first experiment we performed a simulation in which all nodes are regular peers. In this simulation we are going to observe at which point peers cannot finish their downloads because of seed departure. Figure 5-13 shows swarm lifecycle evolution. The black line is the deadline for new joining peers. This means if a peer joins the swarm after 1395.00 simulation seconds, he will not be able to complete his download because some pieces will disappear before he can get them.

For the second experiment, we have introduced a Crowd group in the swarm. Our aim is to study if Group Piece Selection strategy can help to restore it. To determine whether a Crowd can help to save a swarm and lengthen its life expectancy, we have run different simulations.

In each simulation the group enters a bit later. We repeat this process until we find a new deadline. In Figure 5-14 we can see the new deadline position (red line). It shows that if a group joins the swarm, any peer that enters before instant 1980.00 can complete its download. Beyond this point even the Crowd could not recover the swarm.

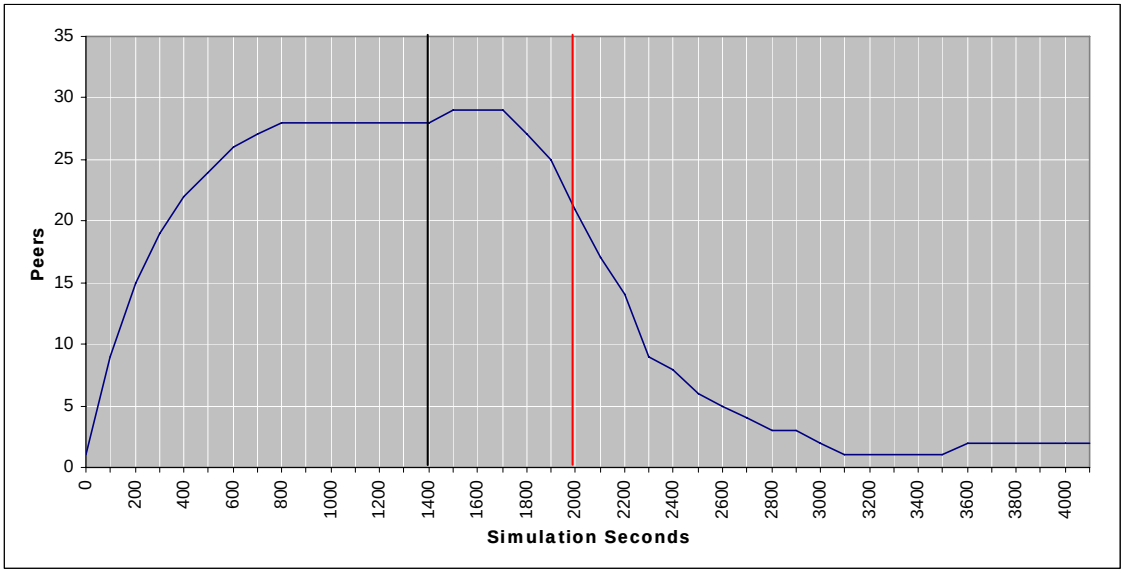

Figure 5-14. Deadline with a Crowd in the swarm.

In these two performed tests, seeds leave the swarm very fast. This causes that some pieces can become unreachable quite easily. This situation affects negatively to peers, which cannot finish their content if they join beyond second 1395. However, thanks to the Group Piece Selection strategy this deadline point can be fixed 585 seconds later. This phenomenon is caused by distributed copy that Crowd members build. As we have seen in previous tests, a crowd group can get all content pieces 57% faster than a peer working alone. As distributed copy is achieved faster, crowd members can end their downloads joining much later. This also allows regular peers to complete their content.

5.3. Crowd Members First strategy

In section 5.2 we have performed some tests to check the proper running of Group Piece Selection strategy. These tests have showed that this strategy is focused on achieving a good performance for the distributed copy. However, it does not bring any improvement for group members download times. In order to achieve some improvement over that field, we have designed an expansion of rechoking algorithm.

All peers in a swarm offer their upload bandwidth through four slots. Each one of this upload slots receives the same bandwidth amount. Periodically, these four slots are auctioned among the peers in the swarm. The algorithm which is responsible for deciding which peers take these slots is the Rechoking Algorithm.

Crowd Members First expansion seeks to habilitate certain privileges between group members when they need to exchange pieces. This privileges consist in habilitate an additional upload slot (beside from the four rechoking algorithm slots) which is dedicated exclusively to Crowd peers. An important point of this algorithm is to decide which one of the Crowd members will be assigned to this extra slot. For taking this decision we will use the same strategy as rechoking algorithm. The slot will be assigned to the faster uploader of the Crowd.

However, this strategy and the Group Piece Selection proposal pursue opposing objectives. One seeks that peers get pieces that other group members does not have to complete a distributed copy of the content. The other wants to facilitate pieces exchange between members to increase download speed. For these two techniques can coexist, Crowd Members First strategy will only be active when the distributed copy is complete or nearly complete.

This strategy is an expansion of the original rechoking algorithm. In further sections we will explain what additions are necessary to implement Crowd Members First strategy.

5.3.1. Gather information

As we have previously stated, the extra upload slot will be assigned to the Crowd member who is giving more upload rate to the peer who is rechocking. This means that to assign this extra slot we are using a tit-for-tat strategy. To implement this process, we need to count an historical of what upload is giving each group member in order to discover who the faster uploader is. Nevertheless, this information is not difficult to obtain. That is because BitTorrent already stores this information in order to perform rechoking operations. As our strategy is an expansion of rechoking we use this data for our approach.

5.3.2. Modeling Crowd Members First strategy

Crowd Members First algorithm is an expansion of the original rechoking algorithm. For building our algorithm, we will maintain the code implemented by the rechoking algorithm and we add our extension once the normal rechoking operations have finished.

```
Execute original Rechoking algorithm code
//Once the Rechoking process is finished

IF strategy is active THEN
    fast_list = Get fast uploaders sorted list
    fast_list = filter peers no members of Crowd
    //Now we have list of crowd members sorted by gived uploads
    actual_owner = get the actual extra-slot owner

    //Now we have to get first(most giver) interested peer of list
    FOR each crowd_peer in fast_list DO
          IF crowd_peer is state.CHOKED THEN
                //First finded is a choked peer
                //Means new peer on the slot
                Unchoke crowd_peer
                Choke actual_owner
                actual_owner = crowd_peer
                Exit FOR loop(break)

          ELSE
                //First finded is unchoked peer
                IF crowd_peer = actual owner THEN
                      //The faster member already in the slot
                      //so, do nothing
                      Exit FOR loop(break)
                END IF
          END IF
    END LOOP
END IF
```

Figure 5-15. Crowd Members First pseudocode.

In figure 5-1 we can see how the Crowd Members First strategy is performed. Our addition to the algorithm starts when the normal rechoking operations have finished and the four upload slots have been decided.

The auction for deciding which Crowd member will be assigned to the extra slot is quite simple. That is because the original rechoking algorithm has already done the hard work. During its running, it has built a sorted list of peers which are giving more upload bandwidth to the rechoking peer. We are going to use this list for our own purpose. But before using it, we must filter the peers that are not Crowd members. Once this filtering is done, we obtain a sorted list with Crowd faster uploaders.

At this stage, we only have to select the first peer on the list and assign it to the extra slot. However, when we select this peer it can be in one of the following states. Depending on its state we will proceed differently.

1. The selected peer is choked. This means that peer is the fast uploader and does not have any slot for upload. In this situation, we move out the old assigned peer and assign this new one to the additional slot.
2. The selected peer is unchoked and is the actual owner of the slot. That means that peer won the additional slot in the last rechoke process and still is the fast uploader on

the actual one. No changes are required for this situation because fast crowd uploader is already assigned to the additional slot.

3. Peer is unchoked and is not the owner of the additional slot. This means that peer is the fast uploader but it has get unchoked during the normal rechoking process (wining one of the four default slots). When this happens, we will choose the next peer on the list and we will repeat the process.

With this addition to the rechoking algorithm we have described all Crowd Members First strategy modifications. In next section we will check how the two collaborative strategies perform together and how the inclusion Crowd Members First affects to the Group Piece Selection strategy.

5.4. Crowd Members First validation

In this section we will continue with the experiments to evaluate the collaborative strategy that we began in section 5.2 with the validation of the Group Piece Selection strategy. As in that case, these experiments were conducted in a discrete even-based peer-to-peer simulator GPS [23].

In this validation process we want to evaluate de proper working of Crowd Members First strategy. To perform this process we should focus on two main aspects. Firstly, we should keep in mind that this strategy has been designed to improve the download times of group members. Secondly, the two collaborative strategies we have presented are complementary. So, Crowd Members First strategy should have minimal impact over the distributed copy build by Group Piece Selection strategy.

The experiments for validating Crowd Members First strategy will be performed in a heterogeneous environment. In this environment we have categorized two peer models with different upload capacities. These capacities are 0.2 Mb/s for regular peers and 1.0 Mb/s for high capacity peers. As in previous validations download capacities are ten times higher: 2.0 Mb/s for regular peers and 10.0 Mb/s for high capacity peers. The swarm composition will be the same for every simulation. At the start there will be only one initial seed. A total of 30 peers will join the swarm following the equation 1 described by Guo et al. in [24]. From these 30 peers, five will be high capacity peers and the remaining 25 will be regular peers. Other significant parameters are:

Torrent parameters:

- File size: 50 MB
- Pieces: 196
- Peers: 30
- Seeds: 1

Clients are configured as:

- 4 unchoke slots + 1 optimistic
- Upload: 0.20 / 1.00 Mb/s
- Download: 2.0 / 10.0 Mb/s

In order to validate the Crowd Members First strategy we have run two different simulations. For both simulations we have introduced a Crowd group into the swarm. This group is formed by four members, three regular peers and one high capacity peer. However, in each simulation we have activated different collaborative strategies. In the first simulation only Group Piece Selection strategy is active. During its validation, we have seen that this strategy does not affect group members download times. Therefore, we can use this simulation as a model to see how Crowd Members First affects to group peers and the distributed copy. In the second simulation we have activated the two collaborative strategies. We have already mentioned that Crowd Members First strategy only activates in specific situations. In this simulation it will activates when the distributed copy reach a progress of 95%.

Download times for both simulations are shown in Tables 5-3 and 5-4. Table 5-3 shows the results of Group Piece Selection simulation. Table 5-4 contains the results of the simulation with both collaborative techniques enabled. Both tables' results are sorted by the peers joining time. We recall the fact that peers who joined the swarm earlier have higher download times, while the times of late joining peers are lower. This is a consequence from peers finishing their download and become seeds. This higher presence of seeds is the cause of this decrease in late peers download times.

The first important fact we can appreciate is that distributed copy has the same download times in both simulations. This means that the two collaborative strategies do not interfere with one another. Thus, we can confirm that Crowd Members First strategy does not activate until distributed copy is almost complete.

If we compare the average download times of the four group members, we can see that their time have fallen from 1071.13 to 1003.21 seconds. This represents a decrease of 67.92 seconds (6.34%). This improvement may seem small. However, we have to consider that Crowd Members First strategy is only activated when distributed copy is almost complete (95% progress). As the distributed copy is completed in 724.14 seconds, it would have taken 687.93 seconds to reach the 95%. Therefore, Crowd members have only activated our strategy and average time of 1071.13 - 687.93 = 383.2 seconds. This means that the average decrease of 67.92 seconds has been achieved during the 383.2 seconds that our strategy has been active. This means an improvement of 17.72%. So, we can state that Crowd Member First strategy works quite well; at least for the short time it is active.

```
   Time elapsed BTPeer(2): 1049,1 sec       Time elapsed BTPeer(17): 1138,91 sec
   Time elapsed BTPeer(3): 1028,34 sec      Time elapsed BTPeer(18): 683,55 sec
   Time elapsed BTPeer(4): 1270,08 sec      Time elapsed BTPeer(19): 1171,13 sec
   Time elapsed BTPeer(5): 1246,25 sec      Time elapsed BTPeer(20): 1120,54 sec
   Time elapsed BTPeer(6): 865,53 sec       Time elapsed BTPeer(21): 1041,74 sec
   Time elapsed BTPeer(7): 1122,73 sec      Time elapsed BTPeer(22): 1052,83 sec
   Time elapsed BTPeer(8): 1196,2 sec       Time elapsed BTPeer(23): 954,52 sec
   Time elapsed BTPeer(9): 1048,16 sec      Time elapsed BTPeer(24): 891,45 sec
   Time elapsed BTPeer(10): 1117,92 sec     Time elapsed BTPeer(25): 939,18 sec
   Time elapsed BTPeer(11): 794,95 sec      Time elapsed BTPeer(26): 880,2 sec
   Time elapsed BTPeer(12): 1198,53 sec     Time elapsed BTPeer(27): 838,19 sec
   Time elapsed BTPeer(13): 1173,12 sec     Time elapsed BTPeer(28): 700,47 sec
   Time elapsed BTPeer(14): 1209,5 sec      Time elapsed BTPeer(29): 520,65 sec
   Time elapsed BTPeer(15): 1134 sec        Time elapsed BTPeer(30): 218,27 sec
   Time elapsed BTPeer(16): 1095,53 sec     Time elapsed BTPeer(31): 207,2 sec
                                            Time of DC           : 723,37 sec

Total 30 downloads, average download time 963.625 sec
Crowd members average download time 1071.13 sec.
```

Table 5-3. Time results with normal rechoking.

```
   Time elapsed BTPeer(2): 1007,72 sec      Time elapsed BTPeer(17): 1174,69 sec
   Time elapsed BTPeer(3): 1087,55 sec      Time elapsed BTPeer(18): 770,23 sec
   Time elapsed BTPeer(4): 1314,85 sec      Time elapsed BTPeer(19): 1111,16 sec
   Time elapsed BTPeer(5): 1163,24 sec      Time elapsed BTPeer(20): 1098,57 sec
   Time elapsed BTPeer(6): 893,18 sec       Time elapsed BTPeer(21): 1045,11 sec
   Time elapsed BTPeer(7): 1138,49 sec      Time elapsed BTPeer(22): 1012,91 sec
   Time elapsed BTPeer(8): 1163,18 sec      Time elapsed BTPeer(23): 991,96 sec
   Time elapsed BTPeer(9): 1104,73 sec      Time elapsed BTPeer(24): 900,15 sec
   Time elapsed BTPeer(10): 1082,56 sec     Time elapsed BTPeer(25): 961,08 sec
   Time elapsed BTPeer(11): 784,99 sec      Time elapsed BTPeer(26): 862,65 sec
   Time elapsed BTPeer(12): 1071,95 sec     Time elapsed BTPeer(27): 836,6 sec
   Time elapsed BTPeer(13): 1079,35 sec     Time elapsed BTPeer(28): 723,67 sec
   Time elapsed BTPeer(14): 1214,43 sec     Time elapsed BTPeer(29): 528,44 sec
   Time elapsed BTPeer(15): 1106,84 sec     Time elapsed BTPeer(30): 223,18 sec
   Time elapsed BTPeer(16): 1225,9 sec      Time elapsed BTPeer(31): 207,2 sec
                                            Time of DC           : 724,14 sec

Total 30 downloads, average download time 962.875 sec.
Crowd members average download time 1003.21 sec.
```

Table 5-4. Time results with Crowd Members First rechoking.

So far, we have studied the average download time of group peers. If we focus on the individual download time of each group member (highlighted in bold), we can observe that a group member has improved much lower than the other peers. This negatively affects the average improvement of the Crowd. Peer 11 only improves 9.96 seconds when uses Crowd Members First strategy. Other peers in the group have decreased their download times in 35.36 (peer 10), 93.77 (peer 13) and 126.58 seconds (peer 12). Interestingly, the group high capacity peer is who presents the worst improvement ratio. It is also the peer who less time receives the benefits of our strategy. If we calculate how long each peer has been subjected to Crowd Members First strategy, we obtain these results: *peer(11)* 70 seconds, *peer(10)* 439 seconds, *peer(13)* 485 seconds and *peer(12)* 511 seconds. We can observe that peers who use more time Crowd Members First strategy are which receive the greatest improvement.

We must take into account that peer 11 is a special case. Given that it is a high capacity peer, it has five times more capacity than the rest of group members. This causes that, at the moment the distributed copy finishes, peer 11 has most of its parts (about 48% of distributed

copy pieces). This can be seen in Figures 5-16 and 5-17. When the Crowd Members First strategy activates, this high capacity peer is just at 70 seconds of finishing its content. As it has most of content pieces, the other group members have few pieces to offer. Hence it obtains little benefit from our strategy.

The opposite effect occurs with weak Crowd peers. The presence of a strong peer in the group supposes a great benefit for them. As the high capacity peer has most the pieces, it can help weaker peers of the group. We can see this phenomenon if we observe the number of collisions of each simulation. In Figures 5-16 and 5-17 we can compare collisions levels between both simulations until the completion of distributed copy. We can appreciate that the group using Crowd Members First strategy has increased the number of collisions by approximately 20%. This increase is elevated because it occurs in the last 37 seconds of the distributed copy progress (last 5% of DC progress). Such behavior is stronger on the weaker peers. This fact confirms that the strongest group peers are helping weaker ones thanks to Crowd Members First strategy.

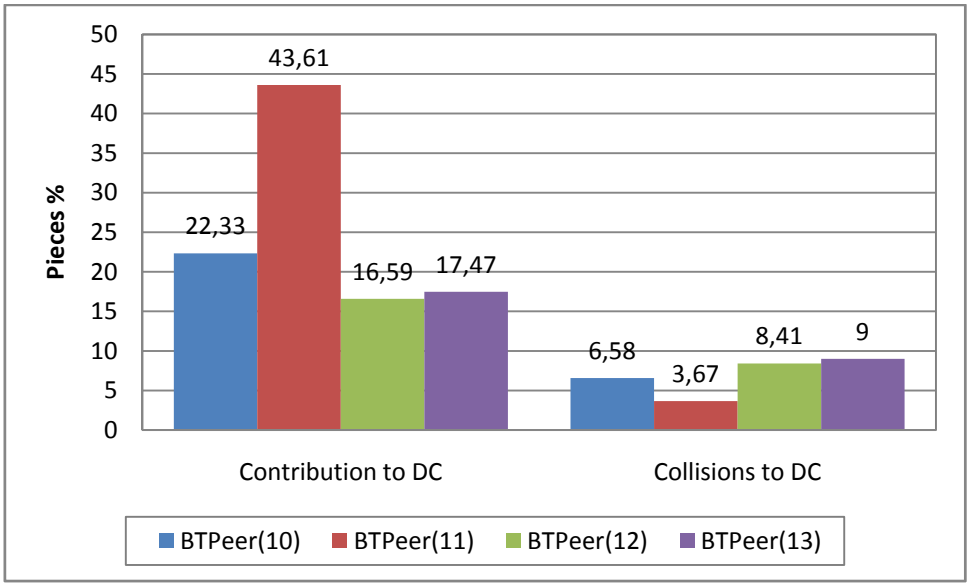

Figure 5-16. Peers contribution with normal rechoking.

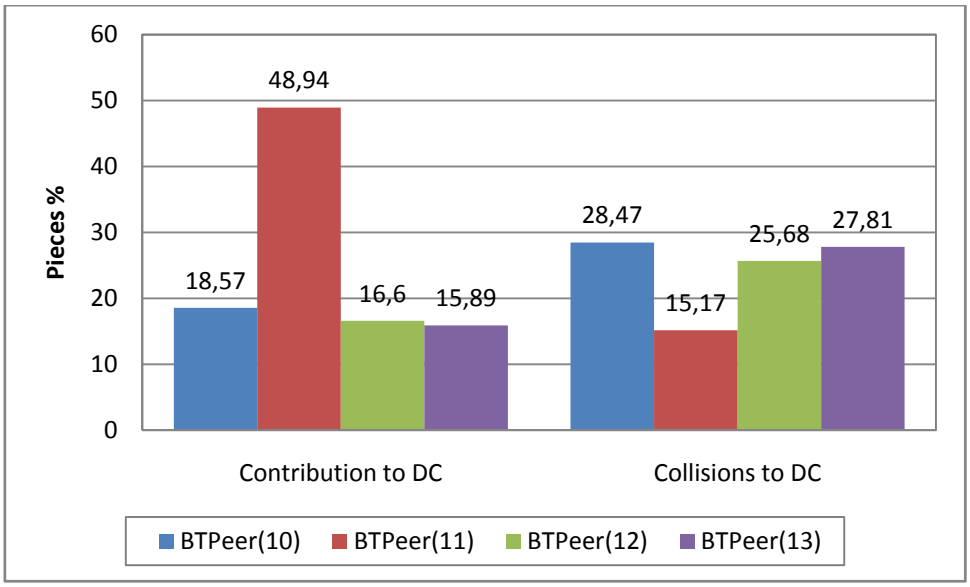

Figure 5-17. Peers contribution with Crowd Members First rechoking.

We can conclude that Crowd Members First strategy can provide good improvements in group members download times. However, to achieve optimal performance it should be activated only when distributed copy is close to completion. Otherwise it can have a negative impact on both collaborative strategies. An optimal environment for this strategy is when one group member has a greater bandwidth than the rest. The assistance of a strong peer causes a great increase in weaker group members' download times.

6. Combined strategies validation

Throughout this project we evaluated the operation of the strategies that we have proposed. In section 4 we evaluated the performance of Download Price Prediction system when it is incorporated to Optimistic Unchoke algorithm of BitTorrent. In section 5 we analyzed the performance and benefits offered by the proposed collaborative strategies: Group Piece Selection and Crowd Members First. However, the operation of these strategies has only been evaluated separately. In this validation process we want to analyze which effects have the combination of our strategies over the members of the group. We expect that Crowd group can get the benefits of our AI strategy and perform even better. In this validation process we want to focus on how the Download Price predictor can strengthen collaborative approaches. So, we will focus in analyze its effects over the distributed copy. As in other validations we will use GPS, a discrete event-based peer-to-peer simulator [23].

To validate how this combination of strategies performs, we will repeat some of the experiments that we conducted when we studied the operation of the Crowd group. For this purpose, we have rebuilt the same scenario that we used in those experiments. We have set a homogeneous scenario where all peers have the same upload capacity: 0.20Mb/s. As usual, initial seed have a greater capacity: 1.00 Mb/s. Regarding the download capacity, like in all performed validations, is ten times higher than upload bandwidth: 2.0 Mb/s. The swarm is composed of 31 peers. At the start there is only one initial seed. A total of 30 peers will join following the equation 1 described by Guo et al. in [24]. Other significant simulation parameters are:

Torrent parameters:
- File size: 50 MB
- Pieces: 196
- Peers: 30
- Seeds: 1

Clients are configured as:
- 4 unchoke slots + 1 optimistic
- Upload: 0.20 Mb/s
- Download: 2.0 Mb/s

Download Price Prediction strategy achieves an average decrease of 16% in peers' download time. This can be seen in section 4 validations. The combination of this strategy with the Crowd group approach will have an effect on the download time of group members. This, at the same time, has a direct impact on distributed copy performance. In this experiment we want to evaluate what impact has on the distributed copy a Crowd group in which all members use a Download Price Predictor. We have run several simulations under the same environment with a crowd group which combines our collaborative and price prediction strategies. For each one of these simulations we have used groups with different sizes. In this way we can study various factors of distributed copy behavior.

Figure 6-1 shows the results of previous distributed copy tests together with the ones obtained in this experiment. On the one hand, the figure shows a study of Crowd group size using only collaborative strategies (red and blue lines). These results were obtained in tests

performed on section 5.2.3. On the other hand, the figure also shows the results of the same test when we are combining our strategies (purple and green lines).

As we expected, there is a decrease in Crowd members download times for the peers using combined strategies. On average, each member of the group finishes its content 237.45 seconds faster. This average improvement in Crowd members download time is caused by Download Price Prediction strategy. How this strategy achieves such peer individual improvement is explained in section 4.

Thanks to the price prediction strategy, Crowd members complete their contents much faster. Thus, they should be able to complete the distributed copy in a shorter time. Green line in Figure 6-1 shows that fact. We can appreciate that, for the same group size, Crowds which use combined strategies achieve the distributed copy sooner. Regardless of the group size, the average improvement is 92.48 seconds over the groups who only use collaborative strategies. This represents approximately an improvement of 13%.

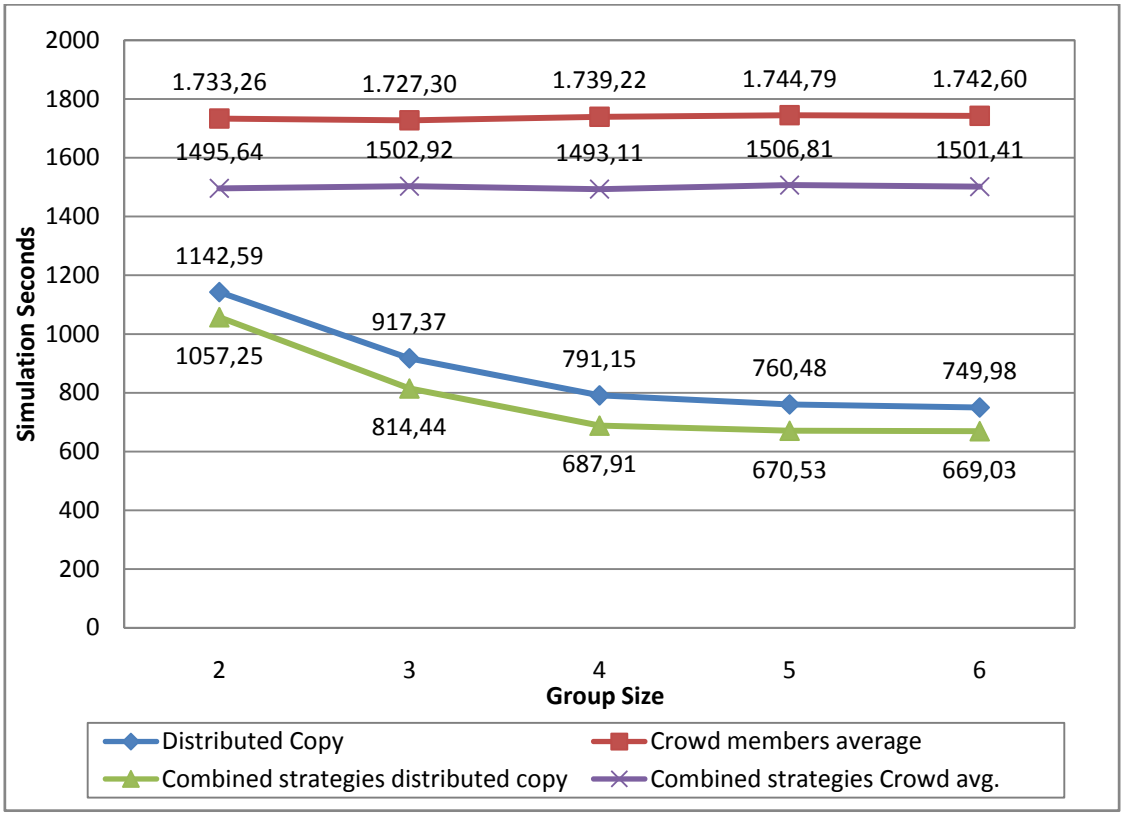

Figure 6-1. Crowd members and distributed copy download times with combined strategies.

When it came to the group size study, the two distributed copies (blue and green lines) show the same behavior. They decrease their download completion time when we increase the Crowd size. However, this performance increase is not proportional to the size of the group. For the Crowd which uses combined strategies, the performance increase when we use a group of three members instead of two is a 22.96%. If we use a four members group the distributed copy is achieved 126.53 seconds faster (11.96%). We should note that four

members group has the best relation between size and distributed copy completion time. From this point on, the distributed copy does not show any significant improvement. Groups of five or six members barely decrease the distributed copy download times. Distributed copies of Crowds who only use the collaborative strategies also show this stagnation in its performance. The underlying reasons of this behavior in groups of more than four members were already explained in section 5.2.3 (Group size influence over distributed copy).

As we have seen in the previous simulation, if we combine collaborative and download price prediction strategies group members can achieve the distributed copy sooner. This means that group members can ensure the completion of their contents in a shorter time. In "test 4: A crowd group saving the swarm" from Group Piece Selection validations, we proved that distributed copy can be used to recover a swarm that is almost dead. These new results can strengthen this property even more. To demonstrate this fact we will repeat "saving the swarm" test we performed in section 5.2.4.

In that test we simulated the death of a swarm by making peers leave as soon as they became seeds. We observed that some regular peers could not complete their contents because of seeds departure. The objective of this experiment was to determine at what instant a joining peer could not finish its download. To find this deadline we ran different simulations. In each simulation the peer joined the swarm a little later. We repeated this process until we found the point where the peer could not finish its download. This operation was performed for a regular peer and for a four members Crowd. This way we found up how many seconds later the Crowd group deadline was placed.

In this experiment, we have performed the same test as in section 5.2.4. This time, however, we have introduced a four members Crowd which combine Collaborative and Download Price Prediction strategies.

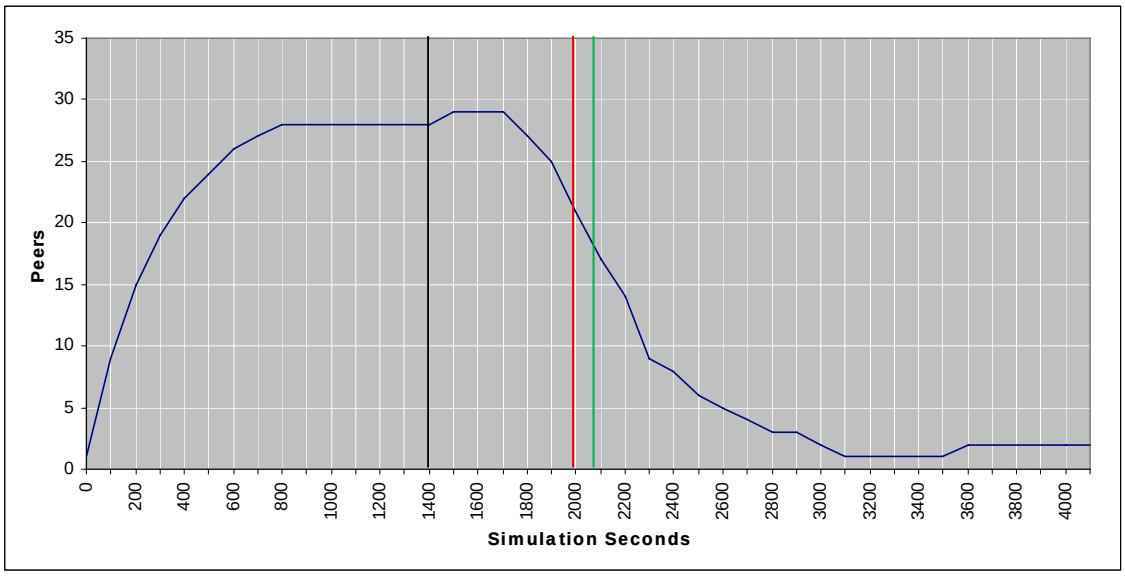

Figure 6-2. Deadline with a Crowd using combined strategies.

Figure 6-2 shows the result of this experiment. Black and Red deadlines were calculated during the previous "saving the swarm" test detailed in section 5.2.4. Black line indicates the

deadline for regular peers. This means that beyond this point any joining peer cannot finish its content because of piece disappearance. Red line shows the deadline for a regular Crowd of four members. Members of a Crowd group can get all content pieces between them (distributed copy) much faster than a peer working alone. For that reason they can join much later and still finish their contents. Green line shows the results of this experiment. When we use a four members Crowd with combined strategies, its deadline is fixed at second 2065. This means an improvement of 85 seconds over a regular Crowd. This result is reasonable, as distributed copy on these Crowds is about 100 seconds faster. Therefore, it is impossible for this kind of Crowds to increase its deadline position more than these 100 seconds.

Seeing these results, we can state that a combination of our strategies can increase the ability of a Crowd of saving a swarm. However, this improvement is limited by the distributed copy completion time. If we find ways of decreasing its download time, we will be able to strengthen this property even more.

7. Conclusions and future work

In this work, we have presented a set of BitTorrent (BT) approaches in order to conduct the problems of global knowledge difficulty acquisition and the common selfish orientation of BT strategies. To do so, we have proceeded as follows: (i) we have designed a Download Price prediction system based on Artificial Intelligence techniques in order to acquire more knowledge about the swarm, (ii) and we have defined a collaborative strategy which focuses on improving the overall state of the swarm.

In the first part of the paper we have presented a system based on Artificial Intelligence techniques that can be trained to perform numerical predictions. The Download Price Prediction system uses AI algorithms to look for patterns in large amounts of information. This information is used as the experience set to foresee any relevant value of the BitTorrent environment. We have specialized this system in predicting the download price of peers. Download Price is a value that indicates how profitable it is to download pieces from a certain peer. Our validation has shown that, embedded in the Optimistic Unchoke algorithm, download price value can achieve completion time improvements of 25.48%. However, even the most well trained of these systems always has a little prediction error which cannot be eliminated. Therefore, this utopian 25.58% is impossible to reach. Tests have proved that our trained system has an error of 5% in its predictions. Even with this error, presence results have shown an improvement of 16.46% over regular peers. This only represents a 9.02% below the best possible performance. We can conclude that prediction systems can be used in BitTorrent environments to achieve significant improvements over regular peer download times.

It is worth mentioning that download price value can be exploited in many other ways. Like the upload bandwidth adjusting system that BitTyrant proposes. However, these kinds of optimizations are based on free-riding behavior and have a very negative impact over the whole swarm.

The second part or the work involves the creation of a reduced group of peers (Crowd) which focus their efforts on improving the swarm condition through collaborative techniques. We have proposed two main approaches regarding this collaborative approach. The first approach, Group Piece Selection, lead the group collaborative efforts on quickly obtaining all content pieces among all group members (distributed copy). The second one, Crowd Members First, defines a priority system and how its members should work together to decrease its individual download completion time. Our validations have proven that Crowd groups can achieve the distributed copy much faster than any other peer on its own. Results indicate that group members are on average at 42.86% of their progress when the distributed copy completes. This behavior proves that Crowd members can ensure its content completion in a rather short time. Furthermore, when distributed copy is reached, Crowd Members First strategy encourages pieces exchange among group members. It grants an improvement of 17.72% in their remaining pieces download time. We have also noted that this fast download copy completion time can be used in a selfless manner. Other performed tests have proved that a crowd could help to restore a swarm which is already dying.

We should note that previous related works like Helper nodes rely on cross-swarm reciprocation. Furthermore, Helper nodes must download content they are not interested in. On the contrary, our approach works inside the same swarm and all group nodes are interested in the downloaded content.

Finally, we have demonstrated that Download Price Prediction and Collaborative strategies can work together in order to increase the distributed copy performance even further. Results show that using these strategies combined represents an 11.96% decrease in distributed copy completion time. This improvement also strengthens the ability of the distributed copy of restoring weak swarms.

We consider this work as an initial step for the artificial intelligence to be present into the BitTorrent universe. We can only hope that these obtained results encourage other researchers to open the protocol to a whole new world of possibilities.

Future work

We should note that the strategies on this work have been conceived in a simulation environment. As a future work, it would be a major step to deploy our system in a real world scenario. It would open AI optimizations to BitTorrent's world as well.

Regarding to artificial intelligence techniques, we have discussed several other approaches to improve the performance and usability for the predictive system. We believe Neural Networks could be a good fit for solving this problem. As of collaborative strategies, a multi-agent system approach should be considered to improve Crowd members connectivity.

8. References

[1] Real-Time internet traffic statistics. http://www.internetobservatory.net

[2] Cisco Systems Inc, Cisco Visual Networking Index: Forecast and Methodology. June 1, 2011

[3] IPOQUE. Internet Study 2009: Data about P2P, VoIP, Skype, file hosters like RapidShare and streaming services like YouTube, November 2009.

[4] A. Legout, G. Urvoy-Keller, and P. Michiardi. “Rarest first and choke algorithms are enough”. In Internet Measurement Conference, 2006, pp. 203–216.

[5] D. Qiu and R. Srikant. “Modeling and performance analysis of bittorrent-like peer-to-peer networks”. In Proc. ACM SIGCOMM'04, Portland, Oregon, USA, Aug. 30 Sept. 3 2004.

[6] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang, “Exploiting bittorrent for fun (but not profit)”. In The 5th International workshop on Peer-To-Peer Systems (IPTPS), 2006.

[7] B. Cohen, “Incentives build robustness in BitTorrent”, in First Workshop on Economics of Peer-to-peer Systems, Berkeley, USA, June 2003.

[8] Raymond Lei Xia and Jogesh K. Muppala, “A Survey of BitTorrent Performance”, in IEEE Communications Surveys & Tutorials, vol. 12, no. 2, 2010.

[9] P. Felber and E. W. Biersack. “Self-scaling networks for content distribution”. In Proc. International Workshop on Self-Properties in Complex Information Systems, Bertinoro, Italy, May-June 2004.

[10] R. Bindal, P. Cao, W. Chan, J. Medved, G. Suwala, T. Bates, and A. Zhang, “Improving traffic locality in BitTorrent via biased neighbor selection,” in ICDCS ’06: Proc. 26th IEEE International Conference on Distributed Computing Systems. Washington, DC, USA: IEEE Computer Society, 2006.

[11] S. Yamazaki, H. Tode, and K. Murakami, “CAT: A cost-aware BitTorrent”, in 32nd IEEE Conference on Local Computer Networks (LCN 2007), Oct 2007, pp. 226–227.

[12] Chi-Jen Wu, Cheng-Ying Li, and Jan-Ming Ho, “Improving the Download Time of BitTorrent-like Systems”, in ICC 2007 proceedings.

[13] N. Laoutaris, D. Carra, and P. Michiardi, “Uplink allocation beyond choke/unchoke: or how to divide and conquer best”, in CoNEXT ’08: Proceedings of the 2008 ACM CoNEXT Conference, 2008, pp. 1–12.

[14] X. Yang and G. de Veciana. “Service capacity in peer-to-peer networks”. In Proc. IEEE Infocom'04, pages 1-11, Hong Kong, China, March 2004.

[15] M. Piatek, T. Isdal, T. E. Anderson, A. Krishnamurthy, and A. Venkataramani, “Do incentives build robustness in bittorrent?” in NSDI, 2007.

[16] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. “Free riding in BitTorrent is cheap. In HotNets”, 2006.

[17] T. Locher, S. Schmid, and R. Wattenhofer, “Rescuing Tit-for-Tat with Source Coding,” in P2P’07.

[18] J. H. T. Wong, “Enhancing Collaborative Content Delivery with Helpers”, MSc thesis, University of British Columbia, September 2004.

[19] Paweł Garbacki and Alexandru Iosup and Dick Epema and Marten van Steen, “2Fast: Collaborative downloads in P2P networks,” in P2P’06.

[20] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. J. T. Reinders, M. van Steen, and H. J. Sips, “Tribler: a social-based peer-to-peer system,” Concurrency and Computation: Practice and Experience, vol. 20, no. 2, pp. 127–138, 2008

[21] BTSlave protocol page. [Online]. Available: http://btslave.sourceforge.net

[22] J. Wang, C. Yeo, V. Prabhakaran, and K. Ramchandran, “On the role of helpers in peer-to-peer file download systems: design, analysis, and simulation,” in IPTPS’07.

[23] W. Yang and N. Abu-Ghazaleh, “Gps: a general peer-to-peer simulator and its use for modeling bittorrent,” in Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on, 2005, pp. 425–432.

[24] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, and X. Zhang, “A performance study of bittorrent-like peer-to-peer systems,” IEEE Journal on Selected Areas in Communications, vol. 25, no. 1, pp. 155–169, 2007.