# Understanding BitTorrent: An Experimental Perspective

Arnaud Legout, Guillaume Urvoy-Keller, Pietro Michiardi

# Understanding BitTorrent: An Experimental Perspective

Arnaud Legout*, Guillaume Urvoy-Keller†, and Pietro Michiardi†
*I.N.R.I.A.
Sophia Antipolis, France
Email: arnaud.legout@sophia.inria.fr
†Institut Eurecom
Sophia Antipolis, France
Email: {Guillaume.Urvoy, Pietro.Michiardi}@eurecom.fr

*Technical Report, Under Submission*
*July 8, 2005*

***Abstract*— BitTorrent is a recent, yet successful peer-to-peer protocol focused on efficient content delivery. To gain a better understanding of the key algorithms of the protocol, we have instrumented a client and run experiments on a large number of real torrents. Our experimental evaluation is peer oriented, instead of tracker oriented, which allows us to get detailed information on all exchanged messages and protocol events. In particular, we have explored the properties of the two key algorithms of BitTorrent: the choke and the rarest first algorithms. We have shown that they both perform remarkably well, but that the old version of the choke algorithm, that is still widely deployed, suffers from several problems. We have also explored the dynamics of a peer set that captures most of the torrent variability and provides important insights for the design of realistic models of BitTorrent. Finally, we have evaluated the protocol overhead. We have found in our experiments a small protocol overhead and explain under which conditions it can increase.**

## I. Introduction

In few years, peer-to-peer applications have become among the most popular applications in the Internet [1], [2]. This success comes from two major properties of these applications: any client can become a server without any complex configuration, and any client can search and download contents hosted by any other client. These applications are based on specific peer-to-peer networks, e.g., eDonkey2K, Gnutella, and FastTrack to name few. All these networks focus on content localization. This problem has raised a lot of attention in the last years [3]–[6].

Recent measurement studies [1], [2], [7], [8] have reported that peer-to-peer traffic represents a significant portion of the Internet traffic ranging from 10% up to 80% of the traffic on backbone links depending on the measurement methodology and on their geographic localization. To the best of our knowledge, all measurements studies on peer-to-peer traffic consider traces from backbone links. However, it is likely that peer-to-peer traffic represents only a small fraction of the traffic on enterprises networks. The main reason is the lack of legal contents to share and the lack of applications in an enterprise context. Network administrators filter out the peer-to-peer ports in order to prevent such traffic on their network. However, we envision at short or mid term a widespread deployment of peer-to-peer applications in enterprises. Several critical applications for an enterprise require an efficient file distribution system: OS software updates, antivirus updates, Web site mirroring, backup system, etc. As a consequence, efficient content delivery will become an important requirement that will drive the design of peer-to-peer applications.

BitTorrent [9] is a new peer-to-peer application that has became very popular [1], [2], [7]. It is fundamentally different from all previous peer-to-peer applications. Indeed, BitTorrent does not rely on a peer-to-peer network federating users sharing many contents. Instead, BitTorrent creates a new peer-to-peer transfer session, called a torrent, for each content. The drawback of this design is the lack of content localization support. The major advantage is the focus on efficient content delivery.

In this paper, we perform an experimental evaluation of the key algorithms of BitTorrent. Our intent is to gain a better understanding of these algorithms in a real environment, and to understand the dynamics of peers. Specifically, we focus on the client local vision of the torrent. We have instrumented a client and run several experiments on a large number of torrents. We have chosen torrents with different characteristics, but we do not pretend to have reach completeness. Instead, we have only scratched the surface of the problem of efficient peer-to-peer content delivery, yet we hope to have done a step toward a better understanding of efficient delivery of data in peer-to-peer.

To the best of our knowledge this study is the first one to offer an experimental evaluation of the key algorithms of BitTorrent on real torrents. In this paper, we provide a sketch of answer to the following questions:

- Does the algorithm used to balance upload and download rate at a small time scale (called the choke algorithm, see section II-C.1) provide a reasonable reciprocation at the scale of a download? How does this algorithm behave with free riders? What is the impact of the algorithm differences when the peer is both a source and a receiver (i.e., a leecher), and the peer is a source only (i.e., a seed)?

- Does the content pieces selection algorithm (called rarest first algorithm, see section II-C.2) provide a good entropy of the pieces in the torrent? Does this algorithm solve the last pieces problem?
- How does the set of neighbors of a peer (called a peer set) evolve with time? What is the dynamics of the set of peers actively transmitting data (called active peer set)?
- What is the protocol overhead?

We present the terminology used throughout this paper in section II-A. Then, we give a short overview of the BitTorrent protocol in section II-B, and we give a detailed description of its key algorithms in section II-C. We present our experimentation methodology in section III, and our detailed results in section IV. Related work is discussed in section V. We conclude the paper with a discussion of the results in section VI.

## II. Background

### A. Terminology

The terminology used in the peer-to-peer community and in particular in the BitTorrent community is not standardized. For the sake of clarity, we define in this section the terms used throughout this paper.

Files transfered using BitTorrent are split in *pieces*, and each piece is split in *blocks*. Blocks are the transmission unit on the network, but the protocol only accounts for transfered pieces. In particular, blocks cannot be served by a peer, only pieces can, i.e., if a piece is partially received, the peer cannot serve that piece.

Each peer maintains a list of other peers it can potentially send pieces to. We call this list the *peer set*. This notion of peer set is also known as neighbor set. A peer can only send data to a subset of its peer set. We call this subset the *active peer set*. The choke algorithm (described in section II-C.1) is in charge of determining the peers being part of the active peer set.

We call local peer, the peer with the monitored BitTorrent client, and remote peers, the peers that are in the peer set of the local peer.

The local peer maintains four flags for each peer in the peer set: *interested*, *interesting*, *choked*, and *choking*. The flags *interested* and *interesting* are set based only on the pieces the local peer and the remote peer have. The flag *interested* means that the remote peer has at least one piece that the local peer does not have, i.e., the local peer is interested in the remote peer. The flag *interesting* means that the local peer has at least one piece that the remote peer does not have, i.e., the local peer is interesting for the remote peer. The flags *choked* and *choking* are set based on the choke algorithm. The flag *choked* means that the local peer will not send any data to the remote peer as long as the flag is set. The flag *choking* means that the remote peer will not send any data to the local peer as long as the flag is set.

Only peers that are unchoked and interested are part of the active peer set.

A peer has two states: the leecher state, when it is downloading a content, but does not have yet all pieces; and the seed state when the peer has all the pieces of the content.

### B. BitTorrent Overview

BitTorrent is a P2P application that capitalizes on the *bandwidth* of peers to efficiently replicate large contents on large sets of peers. Clients involved in a torrent, i.e., the download of a single file, cooperate to replicate the file among each other using *swarming* techniques. In particular, the file is split in pieces of typically 256 KB. Other piece sizes are possible.

A user joins an existing torrent by downloading a *.torrent* file usually from a Web server, which contains meta-information on the file to be downloaded (e.g., the number of pieces and the SHA-1 hash values of each piece) and the IP address of the so-called *tracker* of the torrent. The tracker is not involved in the actual distribution of the file; instead, it only keeps track of the peers currently active in the torrent. The tracker is the only centralized component of BitTorrent.

Active clients report their state to the tracker every 30 minutes in steady-state regime, or when disconnecting from the torrent, indicating each time the amount of bytes they have uploaded and downloaded since they joined the torrent. When joining a torrent, a new client obtains from the tracker a list of IP addresses of active peers to connect to and cooperate with (typically 50 peers chosen at random in the list of active peers). A torrent can thus be viewed as a collection of interconnected sets of peers. If ever the number of neighbors a peer is connected to falls below a predefined threshold (typically 20 peers), this peer will contact the tracker again to obtain a new list of IP addresses of clients.

Each time a client obtains a new piece, it informs all the peers it is connected to. Interactions between clients are based on two principles. First, the choke algorithm that encourages cooperation among peers and limits the number of peers a client is sending simultaneously data to. Second, the rarest first algorithm that controls the pieces a client will actually request in the set of pieces currently available for download. Those algorithms are further detailed in the next section.

### C. BitTorrent Algorithms Description

We focus here on the two most important algorithms of BitTorrent: the choke algorithm and the rarest first piece selection algorithm. We will not give all the details of these algorithms, but we will explain the main ideas behind them.

*1) Choke Algorithm:* The choke algorithm was introduced to guarantee a reasonable level of upload and download reciprocation. As a consequence, free riders, i.e., peers that never upload, should be penalized. The choke algorithm makes an important distinction between the leecher state and the seed state. This distinction is very recent in the BitTorrent protocol and appeared in version 4.0.0 of the *mainline* client [10]. We do not see it documented, neither implemented elsewhere. In the following, we describe the algorithm with the default parameters. By changing the default parameters it is possible to increase the size of the active peer set and the number of optimistic unchoke.

In order to do not confuse the reader, we do not use the flags interested, interesting, choked, and choking to describe the state of the remote peers. In this section, interested always

means interested in the local peer, and choked always means choked by the remote peer.

When in leecher state, the choke algorithm is the following. The algorithm is called every ten seconds and each time a peer joins, leaves the peer set, or becomes interested. As a consequence, the choke period can be much shorter than 10 seconds. Each time this step is executed, we say that the choke algorithm starts a new round, and the following steps are executed.

1) At the beginning of every three round, i.e., every 30 seconds, the algorithm chooses one peer at random that is choked and interested. This peer is called the optimistic unchoked peer.
2) The algorithm orders peers that are interested and have sent at least one block in the last 30 seconds according to their download rate (to the local peer). A peer that has not sent any block in the last 30 seconds is called *snubbed*. Snubbed peers are excluded in order to guarantee that only active peers are unchoked.
3) The three fastest peers are unchoked.
4) If the optimistic unchoked peer is not part of the three fastest peers, it is unchoked and the round is completed.
5) If the optimistic unchoked peer is part of the three fastest peers, another peer is chosen at random.
    a) If this peer is interested, it is unchoked and the round is completed.
    b) If this peer is not interested, it is unchoked and a new peer is chosen at random. Item 5a is repeated with the new randomly chosen peer. As a consequence, more than 4 peers can be unchoked by the algorithm. However, only 4 interested peers can be unchoked in the same round. Unchoking *not interested* peers allows to call the choke algorithm as soon as one of such unchoked peer becomes interested.

In the following, we call the three peers unchoked in step 3 the regular unchoked (RU) peers, and the peer unchoked in step 5a the optimistic unchoked (OU) peer. The optimistic unchoke described in step 1 has two purposes. It allows to evaluate the download capacity of new peers in the peer set, and it allows to bootstrap new peers that do not have any piece to share by giving them their first piece.

In previous versions of the BitTorrent protocol, the choke algorithm was the same in leecher state and in seed state except that in seed state the ordering performed in step 2 was based on upload rates (from the local peer). The current choke algorithm in seed state is somewhat different. The algorithm is called every ten seconds, and each time a peer joins, leaves the peer set, or becomes interested. Each time this step is executed, we say that the choke algorithm starts a new round, and the following steps are executed.

1) Only the peers that are unchoked and interested are considered in the following.
2) The algorithm orders the peers according to the time they were last unchoked (most recently unchoked peers first) for all the peers that were unchoked recently (less than 20 seconds ago) or that have pending requests for blocks. The upload rate is then used to decide between peers with the same time, giving priority to the highest upload.
3) The algorithm orders the other peers according to their upload rate, giving priority to the highest upload, and put them after the peers ordered in item 2.
4) For two rounds over three, the algorithm keeps unchoked the first 3 peers, and unchoke another interested peer selected at random. For the third round, the algorithm keeps unchoked the first four peers.

In the following, we call the three or four peers that are kept unchoked in step 4 the seed kept unchoked (SKU) peers, and the unchoked peer selected at random the seed regular unchoked (SRU) peer. Step 2 is the key of the new algorithm in seed state. Peers are no more ordered according to their upload rate from the local peer, but using the time of their last unchoke. As a consequence, the peers in the active peer set are changed frequently.

The previous version of the algorithm, unlike the new one, favors peers with a high download rate. This has a major drawback: a single peer can monopolize all the resources of a seed, provided it has the highest download capacity. This drawback can adversely impact a torrent. A free rider peer, i.e., a peer that does not contribute anything, can get a high download rate without contributing anything. This is not a problem in a large torrent. But in small torrents, where there are only few seeds, a free rider can monopolize one or all the seeds and slow down the whole torrent by preventing the propagation of rare pieces that only seeds have. In the case the torrent is just starting, the free rider can even lock the seed and significantly delay the startup of the torrent. This drawback can even be exploited by an attacker to stop an incipient torrent, by requesting continuously the same content.

We will show in section IV how the new algorithm avoids such a drawback.

*2) Rarest First:* The rarest first algorithm is very simple. The local peer maintains the number of copies in its peer set of each content piece. It uses this information to define a rarest pieces set. Let $m$ be the number of copies of the rarest piece, then the ID of all the pieces with $m$ copies in the peer set are added to the rarest pieces set. The rarest pieces set is updated each time a copy of a piece is added to or removed from the peer set of the local peer.

If the local peer has downloaded strictly less than 4 pieces, it chooses the next piece to request at random. This is called the *random first policy*. Once it has downloaded at least 4 pieces, it chooses the next piece to download at random in the rarest pieces set. BitTorrent also uses a *strict priority policy*, which is at the block level. When at least one block of a piece has been requested, the other blocks of the same piece are requested with the highest priority.

The aim of the random first policy is to permit a peer to download its first pieces faster than with a the rarest first policy. The aim of the strict priority policy is to complete the download of a piece as fast as possible. However, we will see in section IV-B.2 that some pieces take a long time to be downloaded entirely.

A last policy, not directly related to the rarest first algorithm,

is the *end game mode* [9]. When a peer has sent a request for all blocks, it requests all blocks not yet received to all peers in its peer set. Each time a block is received, it cancels the request for the received block to all peers in its peer set. As a peer has a small buffer of pending requests, all blocks are effectively requested close to the end of the download. Therefore, the *end game mode* is used at the very end of the download, thus it has little impact on the overall performance (see section IV-B.2).

## III. Experimentation Methodology

### A. Choice of the BitTorrent client

Several BitTorrent clients are available. The first BitTorrent client has been developed by Bram Cohen, the inventor of the protocol. This client is open source and is called *mainline*. As there is no well maintained and official specification of the BitTorrent protocol, the *mainline* client is considered as the reference of the BitTorrent protocol. It should be noted that, up to now, each Bram Cohen's improvement to the BitTorrent protocol was replicated to all the other clients.

The other clients differ from the *mainline* client on two points. First, the *mainline* client has a rudimentary user interface. Other clients have a more sophisticated interface with a nice look and feel, realtime statistics, many configuration options, etc. Second, as the *mainline* client defines the BitTorrent protocol, it is de facto a strict implementation of the BitTorrent protocol. Other clients offer experimental extensions to the protocol.

As our intent is an evaluation of the strict BitTorrent protocol, we have decided to restrict ourselves to the *mainline* client. We instrumented version 4.0.2 of the *mainline* client released at the end of May 2005[1].

We also considered the *Azureus* client. This client is the most downloaded BitTorrent client at SourceForge[11] with more than 70 million downloads[2]. This client implements a lot of experimental features and we will discuss some of them when relevant.

### B. Experimentations

We performed a complete instrumentation of the *mainline* client. The instrumentation comprises: a log of each BitTorrent message sent or received with the detailed content of the message (except the payload for the PIECE message), a log of each state change in the choke algorithm, a log of the rate estimation used by the choke algorithm, a log of important events (end game mode, seed state), some general informations.

All our experimentations were performed with the default parameters of the *mainline* client. It is outside of the scope of this study to evaluate the impact of each BitTorrent parameters variation. The main default parameters are: maximum upload rate (default to 20 KB/s), minimum number of peers in the peer set before requesting more peers to the tracker (default to 20), maximum number of connections the local peer can initiate (default to 40), maximum number peers in the peer set (default to 80), number of peers in the active peer set including the optimistic unchoke (default to 4), block size (default to $2^{14}$ Bytes), number of pieces downloaded before switching from random to rarest piece selection (default to 4).

In our experiments, we uniquely identify a peer by its IP address and peer ID. The peer ID is a string composed of the client ID and a randomly generated string. This random string is regenerated each time the client is restarted. The client ID is a string composed of the client name and version number, e.g., M4-0-2 for the *mainline* client in version 4.0.2. We are aware of around 20 different BitTorrent clients, each client existing in several different versions. When in a given experiment, we see several peer IDs on the same IP address (between 0% to 16% of the IP addresses depending on the experiments, with a mean around 7%), we compare the client ID of the different peer IDs. In case the client ID is the same for all the peer IDs on a same IP address, we deem that this is the same peer. The pair (IP, client ID) does not guarantee that each peer can be uniquely identified, because several peers beyond a NAT can use the same client in the same version. However, considering the large number of client IDs, it is common in our experiments to observe 15 different client IDs, the probability of collision is reasonably low for our purposes. Unlike what was reported by Bhagwan et al. [12], we did not see any problem of peer identification due to NATs. In fact, BitTorrent has an option, activated by default, to prevent accepting multiple incoming connections from the same IP address. The idea is to prevent peers to increase their share of the torrent, by opening multiple clients from the same machine.

We did all our experimentations from a machine connected to a high speed backbone. However, the upload capacity is limited by default by the client to 20 KB/s. There is no limit to the download capacity. We obtained effective download speeds ranging from 20 KB/s up to 1500 KB/s depending on the experiments.

The experimental evaluation of the BitTorrent protocol is complex. Each experiment in not reproducible as it heavily depends on the behavior of peers, the number of seeds and leechers in the torrent, and the subset of peers randomly returned by the tracker. However, by considering a large variety of torrents and by having a precise instrumentation, we were able to identify fundamental behaviors of the BitTorrent protocol.

We ran between 1 and 3 experiments on 12 different torrents. Each experiment lasted for 8 hours in order to make sure that each client became a seed and to have a representative trace in seed state. We give the characteristic of each torrent using the notation (t,s,l,f), where t is the torrent id, s (resp. l) is the number of seeds (resp. leechers) in the torrent at the beginning of the experiment, and f is the size in Megabytes of the file distributed using the torrent. Here is the list of the torrents we studied: (1,50,18,600), (2,1,40,800), (3,1,2,580), (4,115,19,430), (5,160,5,6), (6,102,342,200), (7,9,30,350), (8, 1,29,350), (9, 12612, 7052, 140), (10, 462, 180, 2600), (11,

[1]Another branch of development called 4.1.x was released in parallel. It does not implement any new functionality to the core protocol, but enables a new tracker-less functionality. As the evaluation of the tracker functionality was outside the scope of this study we focused on version 4.0.2.

[2]The *mainline* is the second most downloaded BitTorrent client at SourceForge with more than 42 million downloads

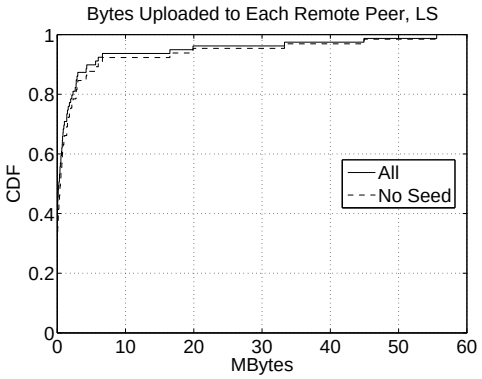


Fig. 1. Local peer in leecher state.

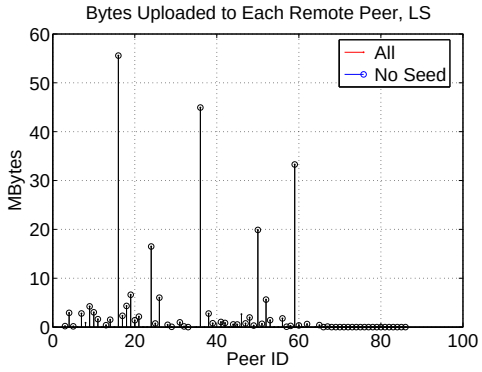


Fig. 2. Local peer in leecher state.

1, 130, 820), (12, 30, 230, 820). As the number of seeds and leechers are given at the start of each experiment, these numbers can be much different at the end.

While these torrents have very different characteristics, we found surprisingly a very stable behavior of the protocol. Due to space limitation we cannot present the results on each experiment. Instead, we illustrate each important result with a figure representing the behavior of a representative torrent, and we discuss the differences of behavior for the other torrents.

## IV. Experimentation Results

### A. Choke Algorithm

In the following figures, the legend *all* represents all peers that were in the peer set during the experiments, and the legend *no seed* represents all peers that were in the peer set in the experiment, but that were not seed the first time they joined the peer set. All the figures in this section are given for torrent 7. The local peer spent 562 minutes in the torrent. It stayed 228 minutes in leecher state and 334 minutes in seed state.

*1) Leecher State:* Fig. 1 represents the CDF of the number of bytes uploaded to each remote peer, when the local peer is in leecher state. The solid line represents the CDF for *all* peers and the dashed line represents the CDF for all the *no seed* peers. We see that most of the peers receive few bytes, and few peers receive most of the bytes. Around 20% of the *no seed* peers seem to receive no byte at all. In fact, when the local peer switched from leecher to seed state he discovered only around 80% of all the peers discovered during the entire experiment. Therefore, most of the peers discovered before the seed state received at least a few bytes during the leecher state.

Fig. 2 represents the aggregate amount of bytes uploaded to each remote peer, when the local peer is in leecher state. Peer IDs are ordered according to the time the peer is discovered by the local peer, first discovered peer with the lowest ID. We see that few peers receive most of the bytes, most of the peers receive few bytes, and all the peers with an ID higher than 65 do not receive any bytes because they were not yet discovered by the local peer.

This result is a direct consequence of the choke algorithm in leecher state. Fig. 3 shows the number of time each peer is unchoked either as regular unchoke (RU) or as optimistic

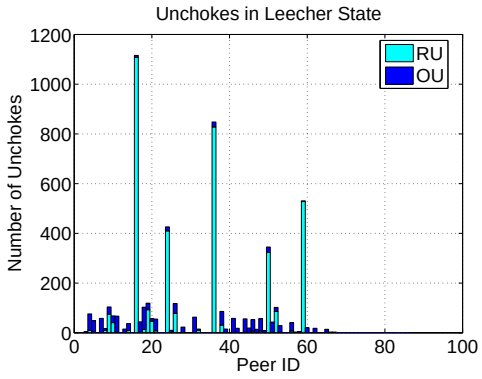


Fig. 3. Number of time each peer is unchoked when the local peer is in the leecher state.

unchoke (OU). We see that most of the peers are optimistically unchoked, and few peers are regularly unchoked a lot of time. The peers that are not unchoked at all are either seeds, or peers that do not stay in the peer set long enough to be optimistically unchoked. After 600 seconds of experiment and up to the end of the leecher state, a minimum of 18 and a maximum of 28 peers are interested in the local peer. In leecher state, the local peer is interested to a minimum of 19 and a maximum of 37 remote peers. Therefore, the result is biased neither due to a lack of peers, nor to a lack of interest.

The observed behavior of the choke algorithm in leecher state is the exact expected behavior. The optimistic unchoke gives a chance to all peers, and the regular unchoke keeps unchoked the remote peers from which the local peer gets the highest download rate.

While the choke algorithm takes its decisions based on the current download rate of the remote peers, it does not achieve a perfect reciprocation of the amount of bytes downloaded and uploaded. Fig. 4 shows the relation between the amount of bytes downloaded from leechers, the amount of bytes uploaded, and the time each peer is unchoked. Peers are ordered according to the amount of bytes downloaded, the same order is kept for the two other subplots. We see that the peers from which the local peer download the most are also the peers the most frequently unchoked and the peers that receive the most uploaded bytes. Even if the reciprocation is not strict, the correlation is quite remarkable.

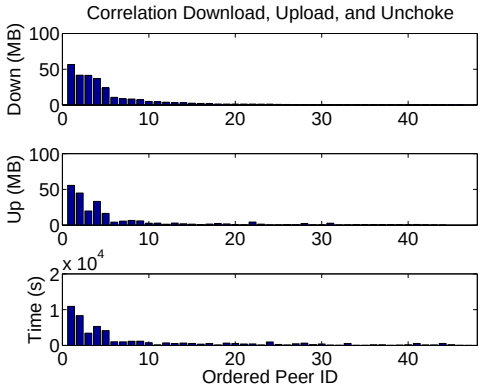


Fig. 4. Correlation between the uploaded and downloaded bytes in leecher state.

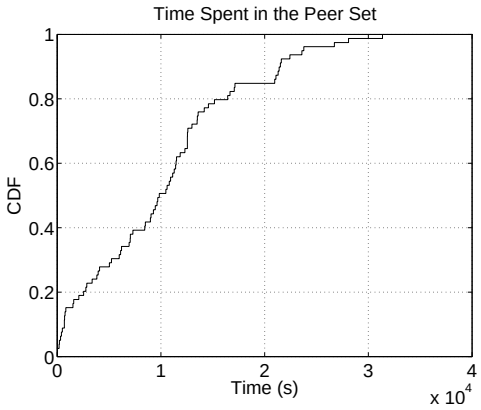


Fig. 6. CDF of the time spent in the peer set.

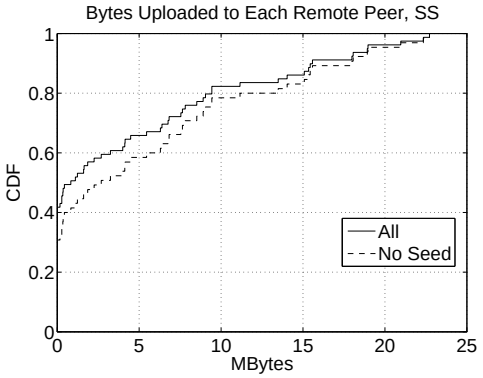


Fig. 5. Local peer in seed state.

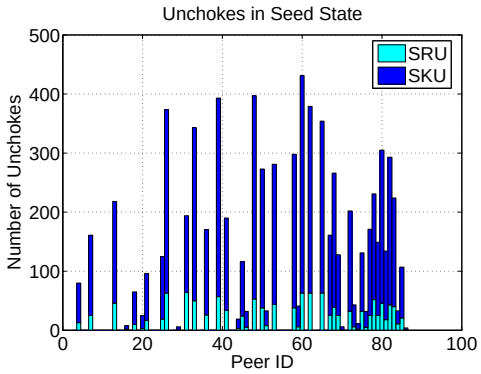


Fig. 7. Number of time each peer is unchoked when the local peer is in the seed state.

We observed a similar behavior of the choke algorithm in leecher state for all the experiments we performed.

*2) Seed State:* Fig. 5 represents the CDF of the number of bytes uploaded to each remote peer, when the local peer is in seed state. The solid line represents the CDF for *all* peers and the dashed line represents the CDF for all the *no seed* peers. We see that the shape of the curve significantly differs from the shape of the curve in Fig. 1. The amount of bytes uploaded to each peer is uniformly distributed among the peers. This uniform distribution is not due to the choke algorithm, but to the time spent by each peer in the peer set. The choke algorithm gives roughly to each peer the same time of service. However, when a peer leaves the peer set, it receives a shorter service time than a peer that stays longer. Fig. 6 shows the CDF of the time spent in the peer set. We observe that the curve can be linearly approximated with a given slope between 0 and 13,680 seconds (228 minutes), i.e., when the peer is in leecher state, and with another slope from 13,680 seconds up to the end of the experiment, i.e., when the local peer is in seed state. The change in the slope is due to seeds leaving the peer set when the local peer becomes a seed. Indeed, when a leecher becomes a seed, it removes from its peer set all the seeds. The time spent in the peer set in seed state follows a uniform distribution as the shape of the CDF is linear. Therefore, the number of bytes uploaded to each remote peer shall follow the same distribution, which is confirmed by Fig. 5.

The uniform distribution of the time spent in the peer set is not a constant in our experiments. For some experiments, the CDF of the time spent in the peer set is more concave, indicating that some peers spend a longer time in the peer set, while most of the peers spend a shorter time compared to a uniform distribution. However, we observe the same behavior of the choke algorithm for all other experiments. In particular, the shape of the CDF of the number of bytes uploaded to each remote peer closely match the shape of the CDF of the time spent in the peer set. Therefore, the service time given by the choke algorithm depends linearly on the time spent in the peer set.

Fig. 7 shows the number of time each peer is unchoked either as seed regular unchoke (SRU) or as seed keep unchoke (SKU). We see that the number of unchokes is well balanced among the peers in the peer set. SRUs account for a small part of the total number of unchokes. Indeed, this type of unchoke is only intended to give a chance to a new peer to enter the active peer set. Then this peer remains a few rounds in the active peer set as SKU. The peer leaves the active peer set when four new peers are unchoked as SRU more recently than itself. It can also prematurely leave the active peer set in case it does not download anything during a round.

We see that with the new choke algorithm in seed state, a peer with a high download capacity can no more lock a seed. This is a significant improvement of the algorithm. However, as only the *mainline* client implements this new algorithm, it is

not yet possible to evaluate its impact on the overall efficiency of the protocol on real torrents.

We will now discuss the impact of the choke algorithm in seed state on a torrent. A client can download from leechers and from seeds, but the choke algorithm cannot reciprocate to seeds, as seeds are never interested. As a consequence, a peer downloading most of its data from seeds will correctly reciprocate to the leechers it downloads from, but its contribution to the torrent will be lower than in the case it is served by leechers only. Indeed, when the local peer has a high throughput capacity and it downloads most of its data from seeds, this capacity is only used to favor itself (high download rate), and not to contribute to the torrent (short upload time). Without such seeds, the download time would have been longer, thus a longer upload time, which means a higher contribution to the torrent in leecher state. Moreover, as explained in section II-C.1, when the local peer has a high download capacity, it can attract and monopolize a seed implementing the old choke algorithm in seed state.

In our experiments we found several times a large amount of data downloaded from seeds for torrents with a lot of leechers and few seeds. For instance, for an experiment on torrent 12, the client downloaded more than 400 MB from a single seed for a total content size of 820 MB. The seed was using a client with the old choke algorithm in seed state. For few experiments, we found a large amount of download from seeds with a version of the BitTorrent client with the new choke algorithm (*mainline* client 4.0.0 and higher). These experiments were launched on torrents with a higher number of seeds than leechers, e.g., torrent 1 or torrent 10. In such cases, the seeds have few leechers to serve, thus the new choke algorithm does not have enough leechers in the peer set to perform a noticeable load balancing. That explains the large download from seeds even with the new choke algorithm.

For one experiment on torrent 5, the local peer downloaded exclusively from seeds. This torrent has a lot of seeds and few leechers. The peer set of the local peer did not contain any leecher. Even if in such a case, the local peer cannot contribute, as there is no leecher in its peer set, it can adversely monopolize the seeds implementing the old choke algorithm in seed state.

In summary, the choke algorithm in seed state is as important as in leecher state to guarantee a good reciprocation. Only the new choke algorithm is robust to free riders and to misbehaviors.

*3) Summary of the Results on the Choke Algorithm:* The choke algorithm is at the core of the BitTorrent protocol. Its impact on the efficiency of the protocol is hard to understand, as it depends on many dynamic factors that are unknown: remote peers download capacity, dynamics of the peers in the peer set, interested and interesting state of the peers, bottlenecks in the network, etc. For this reason, it is hard to extrapolate, without doing a real experiment, that the short time scale reciprocation of the choke algorithm can lead to a reasonable reciprocation in a time scale spanning the whole torrent experiment.

We found that in leecher state: i)All leechers get a chance to join the active peer set; ii)Only a few leechers remain a significant amount of time in the active peer set; iii)For those leechers, there is a good reciprocation between the amount of bytes uploaded and downloaded.

We found that in seed state: i)All leechers get an equal chance to stay in the active peer set; ii) The amount of data uploaded to a leecher is proportional to the time the leecher spent in the peer set; iii) The new choke algorithm performs better than the old one. In particular it is robust to free riders, and favors a better reciprocation.

One fundamental requirement of the choke algorithm is that it can always find interesting peers and peers that are interested. A second requirement is that the set of peers interesting and the set of peers interested is quite stable, at a time scale larger than a choke algorithm round. If these requirements are not fulfilled, we cannot make any assumption on the outcome of the choke algorithm. For this reason, a second algorithm is in charge to guarantee that both requirements are fulfilled: the rarest first algorithm.

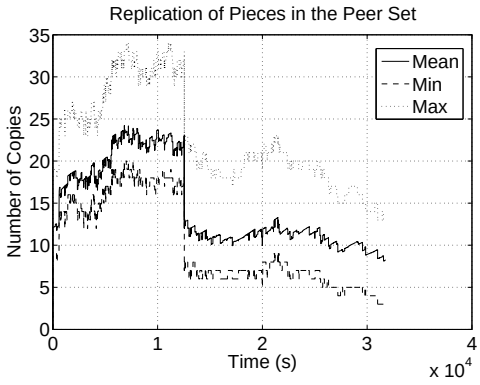


Fig. 8. Evolution of the number of copies of pieces in the peer set.

### *B. Rarest First Algorithm*

The rarest first algorithm objective is to maximize the entropy of the pieces in the torrent, i.e., the diversity of the pieces among peers. In particular, it should prevent the apparition of rare pieces, i.e., pieces with only one copy, or significantly less copies than the mean number of copies. A high entropy is fundamental to the correct behavior of the choke algorithm [9].

The rarest first algorithm should also prevent the last pieces problem, in conjunction with the end game mode. We say that there is a last pieces problem when the time to download the last pieces is significantly larger than for the rest of the pieces.

*1) Rarest Piece Avoidance:* The following figures are the results of an experiment on torrent 7. The content distributed in this torrent is split in 1395 pieces.

Fig. 8 represents the evolution of the number of copies for each piece in the peer set with time. The dotted line represents the number of copies of the most replicated piece in the peer set at each instant. The solid line represents the average number of copies over all the pieces in the peer set at each instant. The dashed line represents the number of copies of the least replicated piece in the peer set at each instant. The most and least replicated pieces change over time.

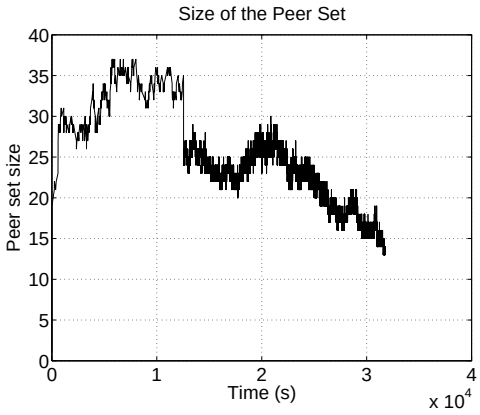


Fig. 9. Evaluation of the peer set size.

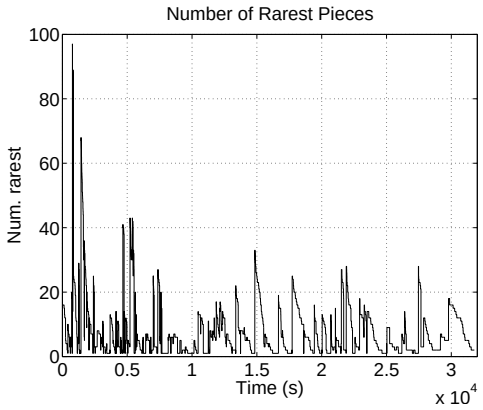


Fig. 10. Evolution of the number of rarest pieces in the peer set.

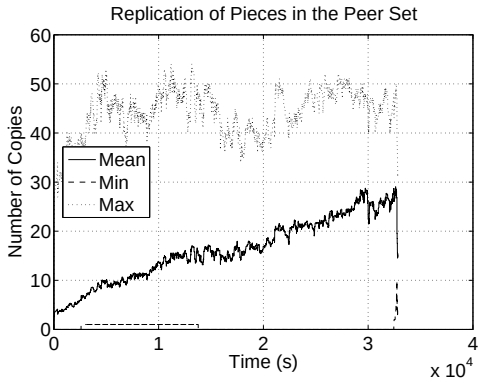


Fig. 11. Evolution of the number of copies of pieces in the peer set.

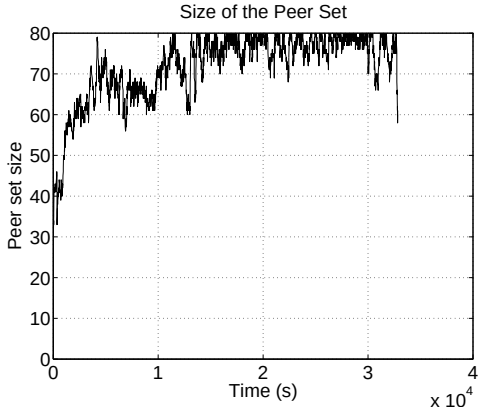


Fig. 12. Evolution of the peer set size for torrent 11.

Despite a very dynamic environment, the mean number of copies is well bounded by the number of copies of the most and least replicated pieces. In particular, the number of copies of the least replicated piece remains close to the average. The decrease in the number of copies 13680 seconds (228 minutes) after the beginning of the experiment corresponds to the local peer switching to seed state. Indeed, when a peer becomes seed, it closes its connections to all the seeds, following the BitTorrent protocol.

The evolution of the number of copies closely follows the evolution of the peer set size as shown in Fig. 9. This is a hint toward the independence between the number of copies in the peer set and the identity of the peers in the set. Indeed, with a high entropy, any subset of peers shall have the same statistical properties.

We see in Fig. 8 that even if the min curve closely follows the mean, it does not significantly get closer. However, that does not mean that the rarest first algorithm does a poor job at increasing the number of copies of the rarest pieces. To support this claim we have plotted over time the number of rarest pieces, i.e., the set size of the pieces that are equally rarest. Fig. 10 shows this curve. We have removed from the data the first second, as when the first peer joins the peer set and it is not a seed, the number of rarest pieces can reach high values.

We see in this figure a sawtooth behavior that clearly shows the effect of the rarest first algorithm. Each peer joining or leaving the peer set can alter the set of rarest pieces. However, as soon as a new set of pieces becomes rarest, the rarest first algorithm quickly duplicates them as shown by a consistent drop of the number of rarest pieces in Fig.10.

We observed the same behavior for almost all our experiments. However, for a few experiments, we encountered periods with some pieces missing in the peer set. To illustrate this case, we now focus on results of an experiment performed on torrent 11. The file distributed in this torrent is split in 1657 pieces. We run this experiment during 32828 seconds. At the beginning of the experiment there were 1 seed and 130 leechers in the torrent. After 28081 seconds, we probed the tracker for statistics and found 1 seed and 243 leechers. After 32749 seconds we found 16 seeds and 222 leechers in the torrent. As a consequence, this torrent had only one seed for the duration of most of our experiment. Moreover, in the peer set of the local peer, there was no seed in the intervals [0,2594] seconds and [13783,32448] seconds.

Fig. 11 represents the evolution of the number of copies for each piece in the peer set with time. We see some major differences compared to Fig. 8. First, the number of copies of the least replicated piece is often equal to zero. This means that some pieces are missing in the peer set. Second, the mean number of copies is significantly lower than the number of copies of the most replicated piece. Unlike what we observed in Fig. 8, the mean curve does not follow a parallel trajectory

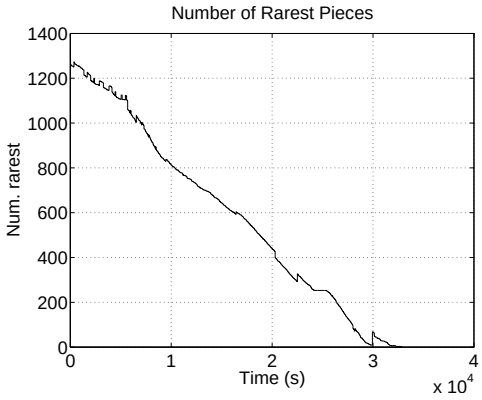


Fig. 13. Evolution of the number of rarest pieces in the peer set.

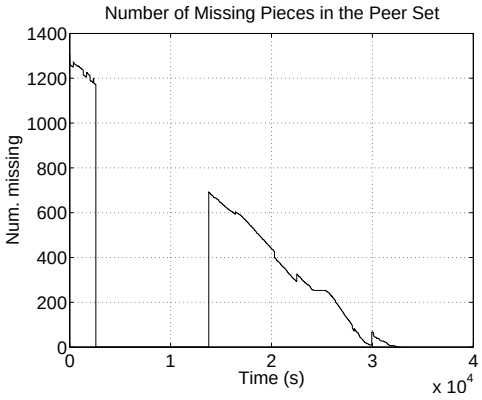


Fig. 14. Evolution of the number of missing pieces in the peer set.

to the max curve. Instead, it is continuously increasing toward the max curve, and does not follow the same trend as the peer set size shown in Fig. 12. As the mean curve increase is not due to an increase in the number of peers in the peer set, this means that the rarest first algorithm increases the entropy of the pieces in the peer set over time.

In order to gain a better understanding of the replication process of the pieces for torrent 11, we plotted in Fig. 13 the evolution of the number of rarest pieces over time, and in Fig. 14 the evolution of the number of missing pieces over time. While the former simply shows the replication of the pieces within the peer set of the local peer without any indication on the origin of the pieces, the later shows the impact of the peers outside the peer set on the replication of the missing pieces within the peer set.

We see in Fig. 13 that the number of missing pieces decreases linearly with time. As the size of each piece in this torrent is 524288 bytes, a rapid calculation shows that the rarest pieces are duplicated at a rate close to 20 KB/s. The exact rate is not the same from experiment to experiment, but the linear trend is a constant in all our experiments. As we only have the peer set view, but not the torrent view, we cannot take into account the peers outside the peer set contributing pieces to the peers in the peer set. For this reason it is not possible to give the exact reason of this linear trend. Our guess is that, as the entropy is high, the number of peers that can serve the rarest pieces is stable. For this reason, the capacity of the torrent to serve the rarest pieces is constant, whatever the peer set is.

Fig. 14 complements this result. We see that each time the seed leaves the peer set, a high number of pieces is missing. We see in Fig. 13 that the slope of the curve remains constant even when the seed is in the peer set. However, only peers outside the peer set, probably the seed itself, can serve the missing pieces, whereas any peers in the torrent can serve the rarest pieces when they are not missing. That confirms that the capacity of the torrent to serve the rarest pieces is stable, whatever the peer set for each peer is.

Fig. 11 shows that the least replicated piece (min curve) has a single copy in the peer set when the seed is in the peer set and is missing when the seed leaves the peer set. Thus, the missing and rarest pieces can only be provided by the seed.

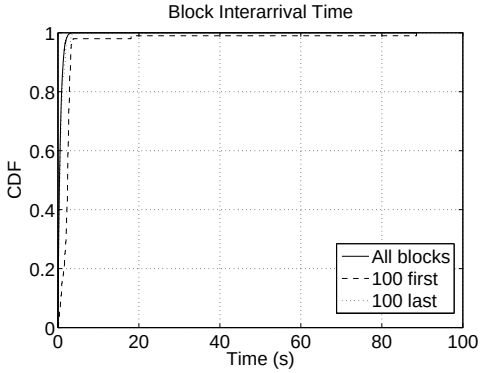


Fig. 15. CDF of the block interarrival time.

Fig. 14 shows that the presence of the seed in the peer set is not necessary to propagate those missing pieces within the peer set. Moreover, the speed of propagation is independent of the presence of the seed in the peer set.

In conclusion, our guess is that with the rarest first policy, the pieces availability in the peer set is independent of the peers in this set. All our experimental results tend to confirm this guess. However, a global view of the torrent is needed to confirm this guess. While it is beyond the scope of this paper to evaluate globally a torrent, this is an interesting area for future research.

*2) Last Piece Problem:* The rarest first algorithm is sometimes presented as a solution to the last piece problem [13]. We evaluate in this section whether the rarest first algorithm solves the last piece problem. We give results for an experiment on torrent 7 and discuss the differences with the other experiments.

Fig. 15 shows the CDF of the block interarrival time. The solid line represents the CDF for all blocks, the dashed line represents the CDF for the 100 first downloaded blocks, and the dotted line represents the CDF for the 100 last downloaded blocks. We first see that the curve for the last 100 blocks is very close to the one for all the blocks. The interarrival time for the 100 first blocks is larger than for the 100 last blocks. For a total of 22308 blocks[3] around 83% of the blocks have

[3]There are 1395 pieces and 16 blocks per pieces, but the last piece can consists of less than 16 blocks.

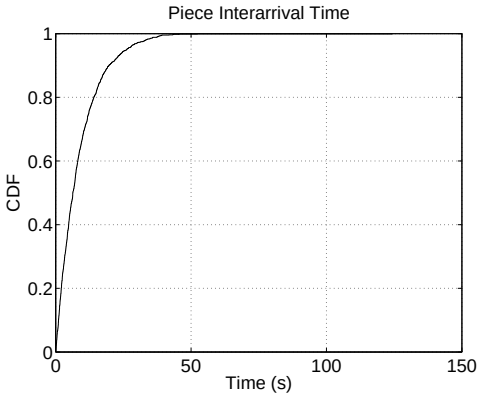


Fig. 16. CDF of the piece interarrival time.

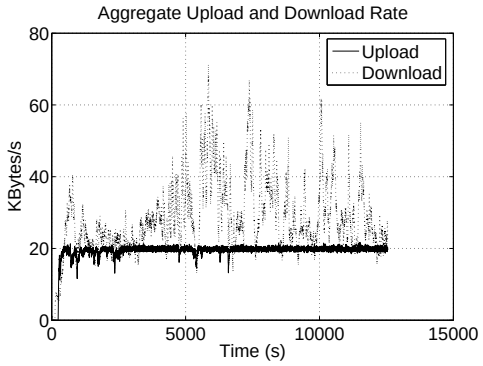


Fig. 17. Upload and download rate of the local peer before the seed state.

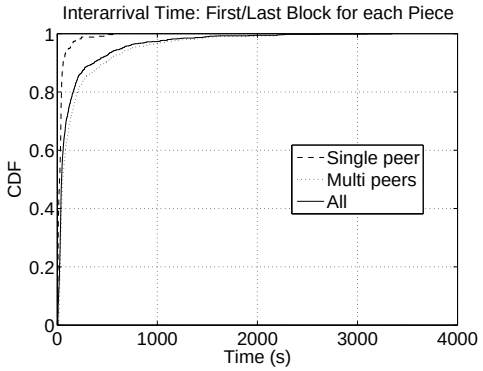


Fig. 18. CDF of the interarrival time of the first and last received blocks for each piece.

a interarrival time lower than 1 second, 98% of the blocks have an interarrival time lower than 2 seconds, and only three blocks have an interarrival time higher than 5 seconds. The highest interarrival time among the last 100 blocks is 3.47 seconds. Among the 100 first blocks we find the two worst interarrival time.

We have never observed a last blocks problem in all our experiments. As the last 100 blocks do not suffer from a significant interarrival time increase, the local peer did not suffer from a slow down at the end of the download. However, we found several times a first blocks problem. This is due to the startup phase of the local peer, which depends on the set of peers returned by the tracker, and the moment at which the remote peers decide to *optimistically unchoke* or *seed regular unchoke* the local peer. We discuss in section IV-B.3 how experimental clients improve the startup phase.

A block is the unit of data transfer. However, a block cannot be retransmitted by a BitTorrent client, only pieces can. For this reason it is important to study the piece interarrival time, which is representative of the ability of the local peer to upload pieces. Fig. 16 shows the CDF of piece interarrival time. The interarrival time is lower than 10 seconds for 68% of the pieces, it is lower than 30 seconds for 97% of the pieces, and only three pieces have an interarrival time larger than 50 seconds. In this torrent the piece size is 256 KB. As a consequence, the worst case upload is 25.6 KB/s for 68% of the pieces and 8.53 KB/s for 97% of the pieces. It appears that the piece interarrival time can be a bottleneck for the upload speed of the local peer. However, as we do not know the interarrival time pattern of the pieces with time, this problem is perhaps not significant. Also, the piece arrival rate cannot be faster than the download rate of the local peer. In order to get a better understanding of this problem, we have plotted the upload and download rate of the local peer.

Fig.17 represents the upload and download rate computed as the average of bytes uploaded and downloaded over a 20 seconds sliding window. We plot a point every two seconds. We do not represent the upload and download rate during the seed state. During this state, the download rate is 0 KB/s and the upload rate stays at 20 KB/s. We see that after a startup phase, the upload rate stays around 20 KB/s. The download rate, despite some spikes, remains close to the upload rate. As a consequence, it appears that the piece interarrival time is mainly constrained by the download rate for this experiment.

Fig. 18 shows the CDF of the interarrival time between the first and last received blocks of a piece for each piece. The solid line shows the CDF for all the pieces, the dashed line represents the CDF for the pieces served by a single peer, the dotted line represents the CDF for the pieces served by at least two different peers. We see that pieces served by more than one peer have a higher first to last block piece interarrival time than pieces served by a single peer. The maximum interarrival time in the case of the single peer download is 556 seconds. In case of a multi peers download, 7.6% of the pieces have a first to last received block interarrival time larger than 556 seconds, and the maximum interarrival time is 3343 seconds. As the local peer cannot upload pieces partially received, a large interarrival time between the first and last received blocks is suboptimal.

Fig. 19 represents the CDF of the pieces served by $n$ different peers. The solid line represents the CDF for all pieces, the dashed line represents the CDF for all pieces downloaded before the end game mode, and the dotted line represents the CDF of the pieces downloaded once in end game mode. We see that a significant portion of the pieces are downloaded by more than a single peer. Some pieces are downloaded by 7 different peers. We note that the end game mode does not lead to an increase in the number of peers that serve a single piece. As the end game mode is activated for the last few blocks to download, the percentage of pieces

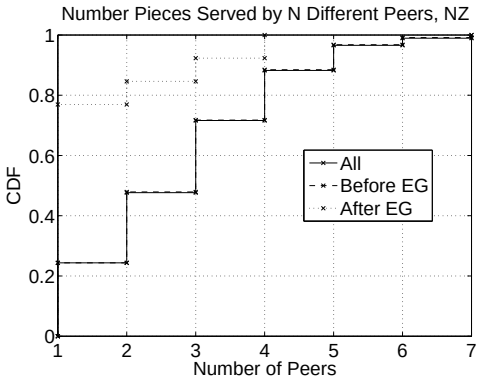


Fig. 19. CDF of the number of pieces served by a different number of peers.

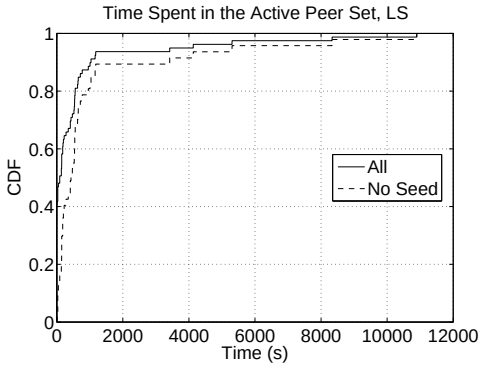


Fig. 20. CDF of the time spent by remote peers in the active peer set in leecher state.

served by a single peer in end game mode is not significant. For some experiments, we got a lower percentage in end game mode than before it. However, none of our experiments shows that the end game mode leads to a piece downloaded by more peers than before the end game mode.

All the results in this section are given for an experiment on torrent 7. However, we did not observe any fundamental differences in the other experiments. The major difference is the absolute interarrival time that decreases for all the plots when the download speed of the local peer increases.

We have not evaluated the respective merits of the rarest first algorithm and of the end game mode. We do not expect to see a major impact of the end game mode. This mode is useful in case of pathological cases, when the last pieces are downloaded from a slow peer. While a user can tolerate a slowdown during a download, it can be frustrating to see it at the end of the download. The end game mode acts more as a psychological factor than as a significant improvement of the overall BitTorrent download speed.

*3) Summary of the Results on the Rarest First Algorithm:* The rarest first algorithm is at the core of the BitTorrent protocol, as important as the choke algorithm. The rarest first algorithm is simple and based on local decisions.

We found that: i)The rarest first algorithm increases the entropy of pieces in the peer set, but also in the torrent; ii)The rarest first algorithm does a good job at attracting missing pieces in a peer set; iii)The last pieces problem is overstated, but the first pieces problem is underestimated.

We saw that multi peer download of a single piece leads to sub optimality. With the rarest first algorithm, rarest pieces are downloaded first. Thus, in case the remote peer stops uploading data to the local peer, it is possible that the few peers that have a copy of the piece do not want to upload it to the local peer. The strict priority policy tries to mitigate this drawback of the rarest first algorithm. This problem can be a starting point to an improvement to the rarest first algorithm.

Finally, we have seen that while we did not observe a last pieces problem, we observed a first pieces problem. The first pieces take time to download, when the initial peer set returned by the tracker is too small. In such a case the Azureus client offers a significant improvement. Indeed, with Azureus, peers can exchange their peer set during the initial peer handshake performed to join the peer set of the local peer. This results in a very fast growth of the peer set as compared to the *mainline* client. We have not evaluated in detail this improvement, but it is an interesting problem for future research.

### C. Peer Set and Active Peer Set Evolution

In this section, we evaluate the dynamics of the peer set and of the active peer set. This dynamics is important as it captures most of the variability of the torrent. These results provide also important insights for the design of realistic models of BitTorrent. All the figures in this section are given for torrent 7.

*1) Active Peer Set:* The dynamics of the active peer set depends on the choke algorithm, but also on the dynamics of the peers, and on the pieces availability. In this section, we study the dynamics of the active peer set on real torrents.

In the following figures, *all* represents all peers that were in the peer set during the experiments, and *no seed* represents all the peers that were in the peer set in the experiment, but that were not seed the first time they joined the peer set.

Fig. 20 represents the CDF of the time spent by the remote peers in the active peer set of the local peer when it is in leecher state. We observe a CDF similar to the one of Fig. 1. It is indeed expected that the longer a peer is in the active peer set, the more it uploads from the local peer. The main difference is that whereas all the *no seed* peers are at least for some time in the active peer set, some *no seed* peers do not upload any data from the local peer. In fact, the choke algorithm optimistically unchoke all the *no seed* peers (Fig. 3), but they are not all able to receive data during the optimistic unchoke period. This is perhaps due to a slow connection or to an overloaded machine.

In Fig. 3, we see that few peers stay a long time in the active peer set as regular unchoke, and the optimistic unchoke gives all the *no seed* peers a chance to join the active peer set. In summary, the active peer set is stable for the three peers that are unchoked as regular unchoke, the additional peer unchoked as optimistic unchoke changes frequently and can be approximated as a random choice in the peer set. This conclusion is consistent for all our experiments.

Fig. 21 represents the CDF of the time spent by remote peers in the active peer set of the local peer when it is in seed

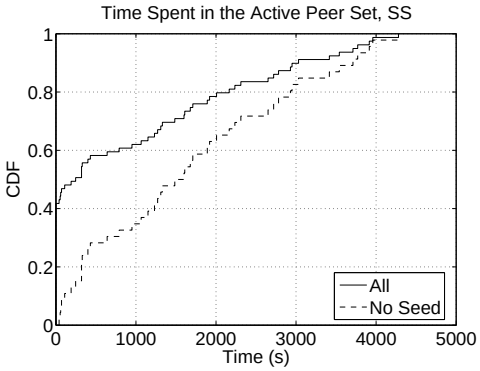


Fig. 21. CDF of the time spent in the active peer set in seed state.

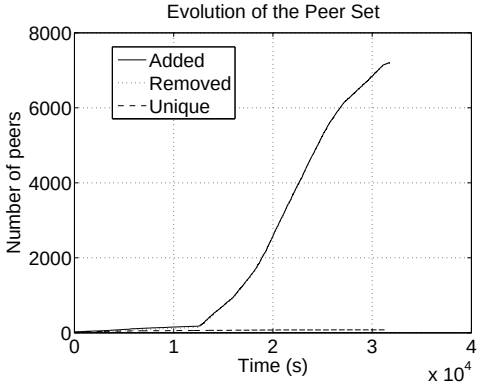


Fig. 22. Evolution of the peer set population for torrent 2.

state. We see a CDF similar to the one of Fig. 5 with the same explanation as for the leecher state.

As explained in section IV-A.2, the distribution of the time spent in the active peer set in seed state is similar to the distribution of the time spent in the peer set in seed state. An analogous behavior has been observed in all our experiments. In particular, the CDF of the time spent by remote peers in the active peer set of the local peer when it is in seed state (Fig. 21), the CDF of the number of bytes uploaded to each remote peer when the local peer is in the seed state (Fig. 5), and the CDF of the time spent in the peer set (Fig. 6) have the same shape. In the next section we evaluate the dynamics of the peer set in our experiments.

*2) Peer Set:* Fig. 9 represents the evolution of the peer set size with time. We see that the peer set size has a lot of small variations. The peer set size decreases 13680 seconds after the start of the experiment, which corresponds to the local peer switching to seed state. Fig. 22 represents the cumulative number of peers joining and leaving in the peer set with time. The solid line is the cumulative number of times a peer joins the peer set, the dotted line is the cumulative number of times a peer leaves the peer set, and the dashed line is the cumulative number of times a unique peer (identified by its IP address and client ID) joins the peer set.

The solid and dotted lines are on top of each other. We first note the huge difference between the cumulative number of joins and the cumulative number of unique peer joining the peer set. The difference grows when the local peer switches to seed state. This difference is due to a misbehavior of a popular BitTorrent client: BitComet[4]. When a local client becomes a seed, it disconnects from all the seeds in its peer set. The *mainline* client reacts to such a disconnect by dropping the connection. Instead, the BitComet clients tries to reconnect. While this behavior could make sense when the local peer is in leecher state, it is a useless when the remote peer becomes a seed. As the frequency of the reconnect is short, this behavior generates a large amount of useless messages. However, compared to the amount of regular messages, these useless messages are negligible.

Fig. 22 shows a slow increase of the unique peers joining the peer set. At the end of the experiment, 79 different peers have joined the peer set. This result is consistent for all the other experiments. Moreover, the increase of the unique peers joining the peer set follows a linear trend with time. The only exceptions are when the number of different peers reaches the size of the torrent. In this case, the curve flattens. This result is important as it means that the amount of new peers injected in the peer set is roughly constant with time. We do not have any convincing explanation for this trend, and we intend to further investigate this result in the future.

### *D. Protocol Overhead*

We have evaluated for each experiment the protocol overhead. We count as overhead the 40 bytes of the TCP/IP header for each message exchanged plus the BitTorrent message overhead. We count as payload the bytes received or sent in a PIECE message without the PIECE message overhead. The upload overhead is the ratio of all the sent messages overhead over the total amount of bytes sent (overhead + payload). The download overhead is the ratio of all the received messages overhead over the total amount of bytes received (overhead + payload).

Overall, the protocol download and upload overhead is lower than 2% in most of our experiments. The messages that account the most for the overhead are the HAVE, REQUEST, and BITFIELD messages[5]. The contribution to the overhead of all other messages can be neglected in our experiments. For three experiments, torrent 5, 9, and 6, we got a download overhead of respectively 23%, 7%, and 3%. This overhead is due to the small size of the contents in these torrents (6 MB, 140 MB, 200 MB), and to a long time in seed state (8 hours). The longer the peer stays in seed state, the higher its download overhead. Indeed, in seed state a peer does not receive anymore payload data, but it continues to receive BitTorrent messages. However, even for a small content and several hours in seed state, the overhead remains reasonable.

For some experiments, we observed an upload overhead up to 15%. Several factors contribute to a large overhead. A small time spent in the seed state reduces the amount of pieces contributed, whereas all the HAVE and REQUEST messages are sent in leecher state. In case the download speed is high

[4]We have observed the same behavior with Azureus.

[5]Due to space limitation, we do not detail all the messages in the BitTorrent protocol. The interested reader is referred to the documentation available on the BitTorrent Web site[10].

and the upload speed is low, then the local peer will contribute even less, but its amount of HAVE and REQUEST messages will remain the same. This is the main reason for the observed overhead of 15%. For very large contents, e.g., torrent 10, the BITFIELD message will be large, thus a larger overhead in particular in seed mode.

The download overhead increases moderately with the time spent in seed state, and it is inevitable to have a download overhead that increases while in seed state. The upload overhead increases, as peers contribute less. Thus, only selfish peers will experience a higher upload overhead. In conclusion, the BitTorrent protocol overhead can be considered as small.

## V. Related Work

While BitTorrent can be considered as one of the most successful peer-to-peer protocol, there are few studies on it.

In [14], [15], and [16], the authors come up with specific *analytical models*. While somehow limited by their assumptions, they provide good insights on a selected subset of the basic properties of BitTorrent-like systems. In [16] the authors propose a deterministic and static analysis of three content distribution architectures: a linear chain, a tree, and a forest of trees. In [14] the authors extend the initial work presented in [15] by providing an analytical solution to a fluid model of BitTorrent. Their results show the high efficiency in terms of system capacity utilization of BitTorrent, both in a steady state and in a transient regime. Furthermore, the authors concentrate on a game-theoretical analysis of the built-in incentive used by BitTorrent to stimulate node cooperation. A common limitation of the analytical models presented in [14], [15], and [16] is that the authors impose a global view of the system state available to all peers, which is in contradiction with the reality since each peer has only a limited knowledge of the torrent, which is defined by its peer set.

Bharambe et al. [13] present a simulation-based study of BitTorrent using a discrete-event simulator that supports up to 5000 peers. The authors concentrate on the evaluation of system performance by tackling salient features of the BitTorrent protocol, such as the local rarest first strategy, and the impact of basic parameters that specify the size of the peer set and the active peer set. Even if the results are very useful in understanding the basic principles that drive BitTorrent, the authors fail to investigate a real world setting for a peer set cardinality larger than 15 peers. In [17], the authors investigate on different peer and piece selection strategies that are compared to the legacy BitTorrent implementation. However, the assumptions of a global knowledge of all peers' algorithms is unrealistic.

The work that is more closely related to our study is [18]. In this paper, the authors provide seminal insights on BitTorrent based on data collected from a "tracker" log for a *single* yet popular torrent, even if a sketch of a local vision from a client perspective is presented. Their results provide information on peers behavior, and show a correlation between uploaded and downloaded amount of data. Our work differs from [18] in that we provide a thorough measurement-based analysis of the fundamental mechanisms that constitute BitTorrent from a local perspective and in that we do not limit our attention only to a very particular torrent typology. Moreover, without pretending to answer all possible questions that arise from a simple yet powerful protocol as BitTorrent we incisively provide the mean of understanding the basic functioning of the core algorithms that constitute BitTorrent. Another work that presents an overview of the BitTorrent protocol is provided by [19]. We cite this work even if the motivation that drives the measurements carried out by the authors is not in line with our objective of understanding how the BitTorrent protocol performs from a client perspective. Indeed, the authors focus on a high-level description of BitTorrent while their results focus on file popularity, file availability, download performance, content lifetime and pollution level.

## VI. Discussion

In this paper, we have evaluated through experimentations the efficiency of the two cardinal algorithms of BitTorrent, namely the choke and rarest first algorithms. By observing a consistent number of torrents with varying characteristics in terms of number of leechers, number of seeds, and content sizes to name a few, we have obtained a clear picture of the actual behavior of the BitTorrent protocol that results from the aggregate behavior of the peers involved in a torrent. We believe that most of the conclusions we drawn out from our study are general, i.e., not simply limited to the very torrents we observed. We give hereafter a summary of our main findings:

- Both algorithms are jointly responsible for an efficient content replication;
- The choke algorithm gives a fair chance to each client to be serviced by a given peer;
- The choke algorithm achieves a good reciprocation with respect to the amount of data exchanged between leechers;
- The new version of the choke algorithm in seed mode solves the free-rider problem that affects previous versions of the protocol, by evenly sharing the capacity offered by a seed among all candidate leechers;
- The rarest first algorithm, *independently* executed by each peer, exhibits the good property of consistently increasing with time the diversity (entropy) of the information spread in the *whole* system under stationary conditions;
- In a highly dynamic environment, the rarest first algorithm is good at balancing the piece distribution as peers joins or leaves the peer set;
- The overhead of the protocol is, in general, very low;

We believe that this work sheds a new light on two new algorithms that enrich previous content distribution techniques in the Internet. BitTorrent is the only existing peer-to-peer replication protocol that exploits these two promising algorithms in order to improve system capacity utilization. We deem that an exhaustive understanding of these two algorithms is of fundamental importance for the design of future peer-to-peer content distribution applications. However, our study should not be considered as a mere interpretation of the characteristics of the piece and peer selection algorithms implemented in BitTorrent. The results and discussions presented

in this paper could be considered as a seed for future research, for example, toward the definition of analytical models based on *realistic* assumptions that can only find their roots in a thorough experimental study.

## REFERENCES


[1] T. Karagiannis, A. Broido, M. Faloutsos, and K. C. Claffy, "Transport layer identification of p2p traffic," in *Proc. ACM IMC'04*, Taormina, Sicily, Italy, October 2004.

[2] T. Karagiannis, A. Broido, N. Brownlee, and K. C. Claffy, "Is p2p dying or just hiding?" in *Proc. IEEE Globecom'04*, Dalla, Texas, USA, Nov. 29-Dec. 3 2004.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. ACM SIGCOMM'01*, San Diego, California, USA, August 27-31 2001.

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proc. ACM SIGCOMM'01*, San Diego, California, USA, August 27-31 2001.

[5] Y. Chawathe, S. Ratnasamy, L. Breslau, and S. Shenker, "Making gnutella-like p2p systems scalable," in *Proc. ACM SIGCOMM'03*, Karlsruhe, Germany, August 25-29 2003.

[6] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The impact of dht routing geometry on resilience and proximity," in *Proc. ACM SIGCOMM'03*, Karlsruhe, Germany, August 25-29 2003.

[7] A. Parker, "The true picture of peer-to-peer filesharing," http://www.cachelogic.com/, July 2004.

[8] CAIDA, "Characterization of internet traffic loads, segregated by application," http://www.caida.org/analysis/workload/byapplication/, June 2002.

[9] B. Cohen, "Incentives build robustness in bittorrent," in *Proc. First Workshop on Economics of Peer-to-Peer Systems*, Berkeley, USA, June 2003.

[10] http://www.bittorrent.com/.

[11] http://sourceforge.net/.

[12] R. Bhagwan, S. Savagen, and G. Voelker, "Understanding availability," in *International Workshop on Peer-to-Peer Systems*, Berkeley, CA, USA, February 2003.

[13] A. R. Bharambe, C. Herley, and V. N. Padmanabhan, "Analysing and improving bittorrent performance," Microsoft Research, Microsoft Corporation One Microsoft Way Redmond, WA 98052, USA, Tech. Rep. MSR-TR-2005-03, February 2005.

[14] 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.

[15] X. Yang and G. de Veciana, "Service capacity in peer-to-peer networks," in *Proc. IEEE Infocom'04*, Hong Kong, China, March 2004, pp. 1–11.

[16] E. W. Biersack, P. Rodriguez, and P. Felber, "Performance analysis of peer-to-peer networks for file distribution," in *Proc. Fifth International Workshop on Quality of Future Internet Services (QofIS'04)*, Barcelona, Spain, September 2004.

[17] 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.

[18] M. Izal, G. Urvoy-Keller, E. W. Biersack, P. Felber, A. A. Hamra, and L. Garcés-Erice, "Dissecting bittorrent: Five months in a torrent's lifetime," in *Proc. PAM'04*, Antibes Juan-les-Pins, France, April 2004.

[19] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips, "The bittorrent p2p file-sharing system: Measurements and analysis," in *Proc. 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, Ithaca, New York, USA, February 2005.