Master Thesis

Free Riding in BitTorrent and Countermeasures

Patrick Moor

Distributed Computing Group
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology Zurich

Summer 2006

Hosts:
Prof. Dr. Roger Wattenhofer
Thomas Locher
Stefan Schmid

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing Group

Abstract

We show that, contrary to common belief, free riding is indeed possible in BitTorrent. We present a BitTorrent client implementation in Java which successfully downloads shared files without ever contributing a single byte of user data to the network. Surprisingly, this client often achieves the same download speed as the original BitTorrent client. This client is of particular interest for people living in countries where sharing of copyrighted material is forbidden by law but acquiring it is legal, no matter from what source.

We also present possible modifications of BitTorrent to effectively reduce free riding. The key concept is a strict tit for tat algorithm paired with ideas from network coding. The new scheme results in a robust, scalable and fair peer-to-peer file sharing system.

Contents

Chapter 1

Introduction

In this thesis we take a close look at the popular peer-to-peer (P2P) network *BitTorrent*[1] which is nowadays considered the most used network for file sharing. The goal is to find ways to free ride in BitTorrent, i.e. downloading the shared files without contributing to the network. After showing that this is indeed feasible we investigate new mechanisms to enforce collaboration and thus inhibit free riding almost entirely.

1.1 Motivation

BitTorrent has long been considered a robust P2P protocol with incentives to render cheating unattractive, i.e. minimize free riding. Free riding, however, can be attractive because of several reasons: Upstream bandwith for residential access is usually more limited than downstream and thus a precious resource. But as network links are becoming faster every year, this will not be a limiting factor in the near future anymore. Another advantage for free riders is the legal aspect of uploading: In Switzerland for example, it is legal to download copyrighted media while uploading and sharing it is forbidden. A client that free rides can therefore be used legitimately to download music and movies for free without the risk of being sued. Software, on the other hand, cannot be downloaded legally using a free riding client.

Once the questions on free riding has been answered, we try to enhance the protocol and invent mechanisms to render free riding infeasible. This includes a *real* tit for tat mechanism and ideas from network coding.

1.2 Contents

In the next chapter we are going to give a brief introduction to BitTorrent. We take a look at the protocol and the mechanisms behind it as well as some important details and flaws. Following that is a chapter about BitThief, our own BitTorrent client implementation in Java. We describe different attacks we tried to apply and discuss their effectiveness. The last two chapters introduce extension mechanisms to enforce collaboration which could effectively prevent free riding and an outlook on further work in the area of BitTorrent.

Acknowledgements

First and foremost I would like to thank my tutors, Thomas Locher and Stefan Schmid, for their ongoing support during the whole six months of working on this thesis. My thanks go further to Prof. Dr. Roger Wattenhofer for letting me work on this thesis at his Distributed Computing group here at ETH Zürich. I also would like to thank Mr. Hans Dubach, head of student affairs, for helping me remember all the important deadlines and for assisting me with various administrative tasks. A special thank goes to the “Informatik-bar” crew for providing me with food, drinks and dozens of cosy hours at the cafeteria during the last five years. Last but not least I would like to thank the Federal Institute of Technology Zurich (ETH Zürich) and especially the Computer Science Department for the valuable education I was able to enjoy.

[1] http://bittorrent.com/

Chapter 2

The BitTorrent Protocol

BitTorrent was invented and implemented by Bram Cohen[1] in 2001. The goal of BitTorrent is to distribute rather large files in an efficient manner. Old client/server models usually impose a huge load on the servers. With BitTorrent, some of the load is distributed among all the peers requesting a file as all peers uploads to other peers while downloading which results in an increased overall bandwith.

Bram's implementation, commonly referred to as the *mainline* client, is written in Python and serves as the de-facto standard for other clients. The most used client nowadays is Azureus[2], which has a lot of features and extensions not available in the original client.

2.1 Terminology

If a user wants to share some files using BitTorrent she first needs to create a *torrent metafile*. The metafile contains essential information about the files being shared:

- names/paths of the files
- file lengths
- piece length
- piece hashes
- tracker announce URL

The data of all files is concatenated in the order the files are specified in the metafile and then split into equal sized *pieces*. Each piece is 2^N bytes long, usually between 64KB and 1MB. BitTorrent then operates on these pieces only and the files are reconstructed *after* all pieces have been fetched.

Once the metafile has been created, the user has to start a BitTorrent client to *seed* the torrent. A BitTorrent node that has already downloaded the whole torrent is commonly referred to as a *seed* or *seeder*. The client will announce itself to the tracker URL specified in the metafile and the *tracker* will store the address and the *info hash* (SHA1 hash of the metafile) for later retrieval. Metafiles can be published as any ordinary file: On a website, by Email, on a CD-ROM or even a DVD and are usually only a couple of KB in size. Most of the metafiles however end up on public torrent sites among dozens of thousands other ones where they can be easily searched and retrieved.

A user who is interested in downloading the torrent just needs to fetch the metafile found on the web. She then starts a BitTorrent client that contacts the *tracker* first by sending the *info hash* of the torrent metafile. The tracker responds with a set of addresses which the client contacts afterwards. The client's address will also be remembered by the tracker to tell later joining peers about the existence of it. Clients keep querying the tracker once in a while to refresh their view of the other peers. The client has enough information now to start downloading the files from the other peers. At this stage the client is called a *leecher* because it is

[1] http://bitconjurer.org/
[2] http://azureus.sourceforge.net/

still in progress of fetching the files. As soon as it finishes downloading all the pieces it will turn into a seeder. It is up to the client to determine how long to stay in the network as a seeder before leaving.

A tracker, a metafile and the peers registered at that tracker for downloading the specific torrent form a so called torrent *swarm*. Swarms are independent of each other and a client can be part of multiple swarms. This is a notable difference compared to other p2p systems such as Gnutella or eMule where clients are all part of the same network, regardless of what files they share.

2.2 On The Wire

BitTorrent was designed to work over the Internet and thus uses TCP for communication. A connection with a remote peer is initialized by connecting to it on TCP level and then sending a handshake message. The format of the handshake message can be seen in Table A.1.

The message is 68 bytes long and contains, among others, three important parts:

- Reserved Bytes: This is a 64 bit long bitfield which contains information about the protocol extensions supported by the client. The first version of BitTorrent had all bits set to zero, while newer versions already use a couple of bits. An unofficial list of reserved bits can be found on the BitTorrent specification wiki page.[3]

- Info Hash: The SHA1 hash of the metafile is used to signal to the remote peer for which torrent download this connection is planned to be used.

- Peer ID: A 160 bit long bitstring which is chosen at random by each client instance.

After both peers have sent their handshake messages their connection is set up properly. What follows is a stream of messages starting with a 4 byte length prefix each. Before sending any further messages each peer needs to communicate its download progress. The bitfield message as seen in Table A.2 is used for this purpose. It contains a bit for each piece, where 0 is used for a piece yet to be downloaded and 1 marks a piece that has already completely been downloaded and also verified against the hash in the metafile. If a peer does not have any pieces yet it may ommit the bitfield message. When a peer downloads a piece later on, it informs all neighbors by sending a "have" message as listened in Table A.3. If a connection has been idle for two minutes, a peer should send a keep-alive message in order to prevent the remote peer from closing the link. Keep-alive messages are only 4 bytes in size, all set to zero (See Table A.4). The remaining message types will be explained in more detail during the next section. There exists the choke message (Table A.5), unchoke message (Table A.6), interested message (Table A.7), not-interested message (Table A.8), request message (Table A.9), cancel message (Table A.10) and piece message (Table A.11).

2.3 Mechanisms

A BitTorrent connection is stateful and both peers act according to the rules that will be explained below. A typical sequence of messages sent on a connection is shown in Figure 2.1.

2.3.1 Interested/Not Interested

A peer is either interested or not interested in the opposite peer. A new connection starts with both peers being *not* interested in the remote side. Once a peer discovers that the neighbor possesses a piece it is still missing it sends an interested message (Table A.7) to the remote peer. Similarly, if a peer downloads a new piece and discovers that the remote side does not offer any new pieces anymore, it will send a not-interested message (Table A.8).

The interested state is important for the unchoking decisions: The remote can use the unchoke slots more efficiently by knowing in advance which peers will start requesting pieces once unchoked.

[3]http://wiki.theory.org/BitTorrentSpecification#Reserved_Bytes

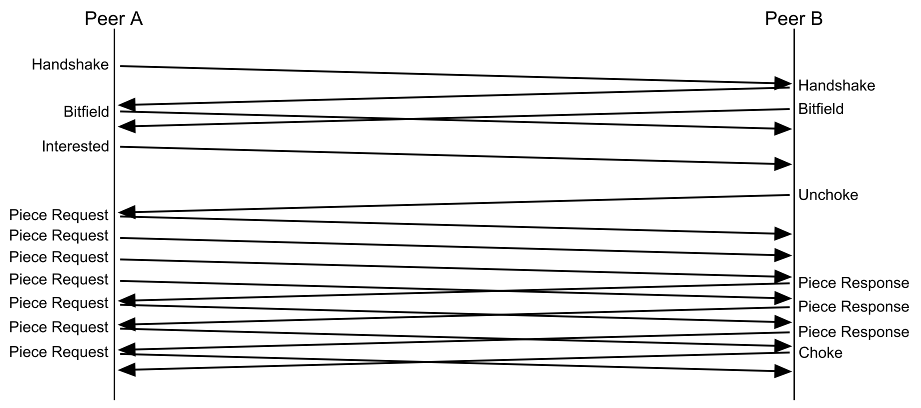


Figure 2.1: Peer A connects to peer B, after which they exchange handshake and bitfield messages. B offers pieces A does not yet have which results in A sending an interested message along the connection. Once B decides to unchoke A it sends an unchoke message to peer A which then immediately starts sending piece requests. The piece requests are answered and new ones sent until B chokes A again.

2.3.2 Choking/Unchoking

Because TCP congestion control is not adequate, Bram Cohen implemented a mechanism based on choking and unchoking to manage bandwith more efficiently. Peers are either choked or unchoked by a remote peer. A connection starts with both peers in choked mode. A choked peer is not allowed to send any piece requests and will be banned immediately if doing so. Once a peer becomes unchoked and is interested it will start sending piece request messages (Table A.9) which are answered by sending the part of the piece requested (Table A.11). Choking and unchoking is performed using the messages shown in Table A.5 and A.6 respectively.

The most important thing about BitTorrent is the algorithm for choosing which peers to choke and unchoke. The mainline client reconsiders its choking decisions once every 10 seconds. Peers are sorted by their recent upload rate starting with the fastest uploaders first. Then, the client unchokes peers starting with the fastest uploader until it has unchoked k peers that are interested. Not interested peers are also unchoked, but they are not considered for the k unchoke slots as they will not be able to request any pieces immediately after being unchoked. If an unchoked but not interested peer becomes interested, the client recalculates the unchoked set for this new situation. Apart from the k unchoked peers there is one *optimistic unchoke* taking place every 30 seconds in order to give recently joined peers an opportunity to download a starting set. This intial set enables them to exchange pieces with other peers using the regular choking mechanisms.

Often people talk about tit for tat when it comes to BitTorrent, but this is simply not appropriate. A peer that has less than k connections into the swarm will *always* upload to any peer that is interested, no matter whether that neighbor contributes anything. Also, while the optimistic unchoke slot is good for fresh peers, it violates tit for tat as well because it enables peers to download without having contributed anything. This clearly is a weakness of the BitTorrent protocol and we will see later on how effectively this can be exploited.

2.3.3 Endgame

Implementations usually refrain from requesting the same sub-piece from multiple peers simultaneously in order not to download data twice. This mechanism is preventing a client to reach fast download rates when most of the pieces have already been downloaded. Once every remaining sub-piece is pending as a request and there are no more left to request, the mainline client switches into *endgame* mode. It now starts requesting sub-pieces from multiple neighbors at the same time which increases overall throughput at the risk of redundant data being downloaded. To reduce this risk Bram introduced a cancel message (Table A.10) used to cancel sub-piece requests as soon as they are downloaded from a different peer. The client ends up downloading only a couple of sub-pieces twice which is a good compromise.

2.4 Extensions

Over the years, the original BitTorrent protocol has been extended in various ways. The people behind the Azureus client are one of the driving forces that push the protocol forward. Naturally, the Azureus client already supports most of the extensions mentioned below.

2.4.1 Distributed Tracker Protocol (DHT)[4]

The tracker represents a single point of failure in the original BitTorrent protocol. Popular torrents often attract thousands of peers and thus impose a heavy load on the tracker resulting in complete denial of service at worst. In order to circumvent the problems of a centralized instance they developed a distributed tracker protocol based on a distributed hash table. The specification draft can be found on bittorrent.org. Both Azureus and the mainline client have implemented a DHT but unfortunately not the same one: Azureus was first in implementing it in a very general way but then Bram decided to create a slightly different (read: incompatible) version for the mainline client.

2.4.2 Peer Exchange (PEX)[5]

Another extension allows the clients to gossip about their known peers and thus exchange active peer information rapidly. Azureus among others has implemented this extension but the mainline client has not. The usefulness can be questioned as this does not help new peers find any neighbors and they still need a tracker of some sort for bootstrapping.

2.4.3 Fast Extension[6]

This extension is officially supported by the mainline client and introduces a couple of modifications to the original BitTorrent protocol:

- Have All/Have None Messages: Instead of sending a bitfield filled with all ones it is simpler to send a have all message. Similarly, instead of sending an all empty field one can send a have none message. The bandwith saved by this mechanism can be questioned though. We do not think it matters a lot.
- Suggest Piece Message: Peers, especially seeders, can send this message to improve overall network efficiency by distributing the sent pieces uniformly among all neighbors and to avoid a piece being downloaded several times while another one is not downloaded at all.
- Reject Request Message: If a request message is known not to be handled, peers now reply with a reject request message. This can happen if a choke message is sent to a remote peer and in the meantime another (valid) request message arrives. The peer should then respond with a reject message in order to inform the remote peer that the request will not be processed.

The fast extension has one more interesting feature: It specifies an algorithm to calculate a piece subset based on the Class C IP-Network the remote peer is in. The pieces in that set can be downloaded by the remote peer “free of charge”, i.e. piece requests will be fulfilled regardless whether the peer is currently choked. This can improve bootstrapping of newly arriving peers by giving them a pseudo-random piece set to start with. The number of pieces available for free is by default set to 10.

2.4.4 Connection Encryption[7]

More and more Internet service providers (ISPs) are starting to throttle or even block BitTorrent traffic. In order to circument these measures Azureus started implementing connection encryption. The encryption key is negotiated using a diffie-hellmann key exchange at the beginning of the connection. There exists a

[4] http://bittorrent.org/Draft_DHT_protocol.html
[5] http://en.wikipedia.org/wiki/Peer_exchange
[6] http://bittorrent.org/fast_extensions.html
[7] http://www.azureuswiki.com/index.php/Message_Stream_Encryption

less useful but more compatible version where the diffie-hellmann key exchange happens *after* the inital handshake sending. The drawback of this solution is that a packet sniffer can still easily detect BitTorrent handshake messages and then throttle or block the corresponding TCP connection.

2.4.5 Cache Discovery Protocol

This official extension has been announced by BitTorrent, Inc. and CacheLogic on August 7 2006.[8] There exist estimates that BitTorrent might account for up to 35% of total Internet traffic[9] and clearly ISPs have an interest in curbing this. With the connection encryption extension this task got more difficult if not impossible. The idea of the cache discovery protocol seems smart: ISPs install special cache machines (similar to proxy servers probably) in their network which will capture and store BitTorrent traffic running through the network. BitTorrent clients will then query the cache servers for pieces before downloading them from remote clients. This keeps (potentially) a lot of traffic within the ISPs local network which is cheap.

Unfortunately, as of this writing, there is not a single bit of documentation or source code available that would explain the details of this protocol. We are very interested in learning more about it, as this could be a *great* opportunity for free riders! Our guess is that BitTorrent, Inc. was not able to find enough ISPs interested in this solution and so the whole thing will probably remain vaporware.

2.5 Quirks

During development of our client we discovered a couple of annoyances in the BitTorrent protocol:

- Length Prefixed Messages: Having a four byte length prefix for each message is a very good strategy, but why is there no length prefix for the handshake message? Implementations could be written much more elegantly if there was one.
- Tracker NAT (Network Address Translation) Check: Some trackers perform NAT checks to verify if a newly joined node can actually be connected to from the outside. The NAT Check is performed by sending a handshake message *without the peer id part* and then waiting for a handshake message from the peer. The problem with this is that peers receiving the first 48 bytes of a handshake message cannot decide whether this is a NAT check or a regular peer connection and the missing 20 bytes will be delivered in an instant. The mainline client "fixes" this by replying with a handshake message already after the first 48 bytes. This just feels wrong. Instead, they should have chosen a NULL peer id for NAT checks in our opinion, which would improve the consistency. Alternatively, the handshake message could start with a length prefix as mentioned above which would make this NAT check more elegant.

[8] http://www.cachelogic.com/home/pages/news/pr070806.php
[9] http://in.tech.yahoo.com/041103/137/2ho4i.html

Chapter 3

BitThief: A Free Riding Client

In this chapter we introduce our own BitTorrent client *BitThief* and the key concepts and ideas. We also analyze the effectiveness of our client and give an overview on the concrete implementation in Java. Our client can be downloaded from `http://dcg.ethz.ch/projects/bitthief/`.

3.1 Concepts and Results

We came up with a list of different attacks, mostly inspired by [7]. The authors of the paper sketch a couple of individual attacks but fail to use them concurrently. Also, their interest lies in the stability and robustness of the bittorrent swarm under the influence of free riders: While this can be an interesting question, we are *not at all* concerned with the stability of the swarm but we only care about our individual, selfish client's performance.

What follows is a short summary of the different attacks and results obtained. For a more detailed discussion we would like to refer to our paper "Free Riding in BitTorrent is Cheap" [8]. The paper can also be found in Appendix B.

3.1.1 Agressive Connection Opening

The mainline client has a default limit of 100 open connections which can be extended using command line arguments. Our client does not impose a limit on the number of active connections; in fact, it tries very agressively to open as many connections as possible. This is achieved by querying the tracker more often for peer addresses. A tracker normally reports no more than 50 peer addresses per query which is only a fraction in a huge swarm with thousands of peers. Our frequent tracker queries improve our knowledge about the swarm rapidly. The distributed tracker protocol (DHT) could also be used for this task but we found ordinary tracker queries to be efficient enough.

Figure 3.1 shows a comparison between BitThief and the mainline client regarding their number of open connections. After just 6 minutes BitThief was able to open almost three times as many connections as the mainline client. The advantage becomes less as time continues but remains notable until the end.

3.1.2 Downloading from Seeders

The seeders are the greatest weakness of BitTorrent: They have already downloaded the complete file and are therefore not interested in downloading anything from their peer neighbors. Thus, they have no better algorithm than round-robin to decide to which neighbor they send data to next. This results in all its neighbors receiving the same share of its upload bandwith, regardless of whether they upload data to other leechers.

A torrent swarm starts with a single seeder and a lot of leechers which will then gradually convert into seeders. These seeders stay online for an unpredictable amount of time: It is basically up to the user to decide when to stop "seeding" a torrent. This results in a torrent swarm consisting of mostly seeders after a while and only a handful of leechers. Our client profits a lot in these situations as trading with other leechers

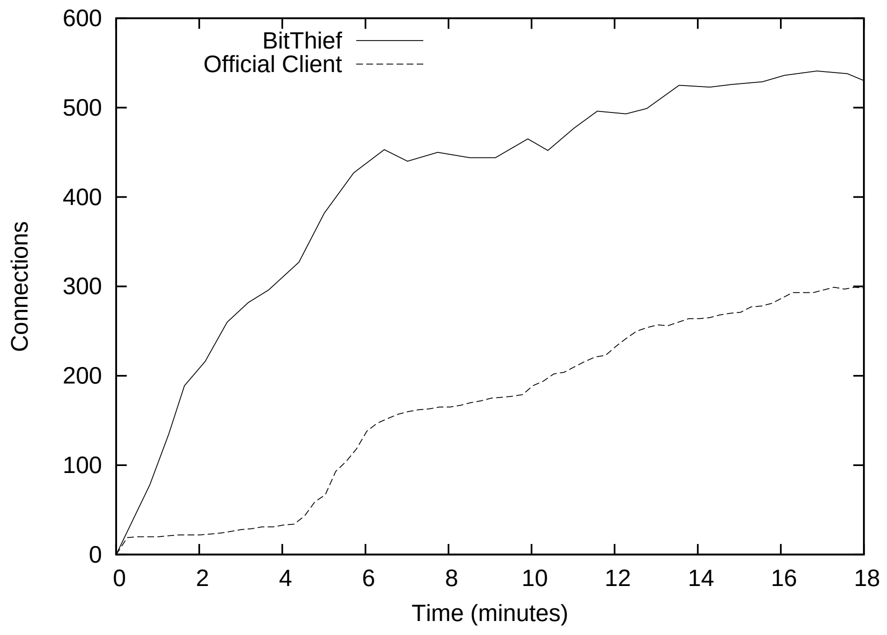


Figure 3.1: Number of open connections over time: In comparison to the official client, BitThief opens connections much faster.

does not boost performance that much. The effect of this surplus of seeders can especially be observed in closed BitTorrent communities with sharing ratio enforcements: Users are very eager to get a high sharing ratio and remain in a torrent swarm as seeders for a very long time.

3.1.3 Uploading Garbage

Piece sizes in BitTorrent are usually between 64KB and 1MB, depending on the choice of the user who created the torrent metafile. The metafile contains SHA1 hashes for each piece, so that a piece can be verified as soon as it has been downloaded completely. In order to more effectively and successfully exchange data between peers, these pieces are broken up into sub-pieces of 16KB in size each, as uploading a 1MB piece would take too long for slow peers. The sub-piece granularity has one big flaw: A peer which receives a sub-piece cannot verify the integrity of it, as the torrent metafile contains only the hashes for whole pieces. The peer therefore has to fetch the whole piece first and can then decide whether the piece is valid or not. In the later case it usually cannot determine which peer it received bad data from. This makes the protocol very unstable when there are peers uploading random garbage.

This serves as an idea for an attack: Instead of pure free riding without uploading any data we consider a more relaxed form of free riding without uploading any *valid* data. By uploading random garbage we can cheat the remote peer into thinking that we are a good neighbor who uploads data quickly and are thus unchoked more frequently which enables us to download at a high speed in return. Unfortunately, a lot of BitTorrent clients try to download all sub-pieces of a piece from the same peer and hence discover quickly that we are cheating. This usually results in a block of our IP address for a couple of hours or even days, depending on the implementation.

In trying to improve this idea we implemented a mechanism which prevents us from uploading a whole piece to a remote peer. That way, the peer needs to fetch the remaining sub-piece from another node and will in the end not be able to detect which peer sent the corrupted data. Unfortunately, this did not work out either, as the peer connection stalled because we did not answer certain sub-piece requests. We conclude that uploading garbage is not an interesting option anymore as most client implementations are good at handling this exploit.

Nevertheless, a hash tree (Merkle Tree) for torrent content verification could be a helpful extension to be able to verify the content at a 16KB block level. That way a malicious peer could be identified and blocked on the spot. There has been a proposed protocol extension[1] but it also acts on piece level which does not make a lot of sense.

3.2 Implementation

We chose to implement our client in Java based on the official protocol definition and various other resources on the Internet. The official (mainline) client implementation[2] (Python) and Azureus[3] (Java as well) were used as a reference once the documentation was too imprecise or simply outdated. We started implementing everything from scratch which turned out to be a great lesson in terms of software engineering. Two screenshots of BitThief in action can be seen in Figure 3.2 and 3.3.

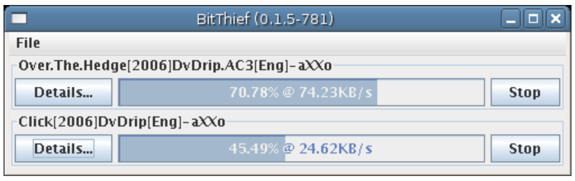


Figure 3.2: The main window of BitThief showing two concurrent torrent downloads.

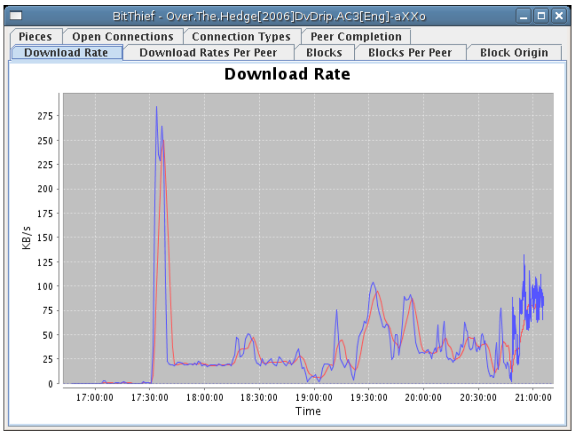


Figure 3.3: A detailed view of one of the torrent downloads. The graph displayed shows the actual download rates in KB/s (blue) as well as the average over a 5 minute window (red).

Technologies that were used include:

- *Log4j:* For all things related to logging as it offers very flexible configuration options.
- *JUnit:* For unit testing. The state of the art testing framework for the Java platform. We have written well over 350 test methods which cover the most important aspects of BitThief.
- *JFreeChart:* This library was used for all graphs visible in the UI and for some internal monitoring and statistics tools.
- *EasyMock:* A library which enormously faciliates writing mock and stub objects for test cases. It also causes a lot of dependency injection to be coded into the application for making tests easier to write.
- *Jakarta Commons:* Some method from this package were used to take over various little tasks: Command line parsing and string conversions for example are being used from this package.

[1]http://tribler.org/developers/?layer0=101
[2]http://www.bittorrent.com/
[3]http://azureus.sourceforge.net/

- *Java NIO:* The “New I/O API” was used for everything network related. The usual network handling paradigm of one thread per connection does not scale to hundreds of connections. Thanks to this API BitThief can handle more than 500 concurrent connections without using a noticeable amount of CPU power.
- *Subversion:* A source version control system that offers many advantages over CVS and helps organizing the project.
- *Apache Ant:* A Java-based build tool used for quick deployment of the application.
- *IntelliJ IDEA:* **The** Java IDE of choice!
- *MATLAB and Maple:* Used for various simulations and calculations.

Chapter 4

Mechanisms Inhibiting Free Riding

4.1 Introduction

When it comes to peer-to-peer file sharing, BitTorrent is one of the most used protocols nowadays: It scales well to thousands of peers and performance is almost optimal, especially for large files. However, in [8] we have shown that BitTorrent is not as robust and fair as often assumed. In fact, free riding without uploading any user data often yields the same performance as when contributing regularly to the network, especially in large torrent swarms.

We have been thinking about how to make BitTorrent more robust, i.e. ways to prevent free riding while keeping the simplicity of BitTorrent and its small overhead. Tit for tat [1] seems to be the best strategy for iterated games where nobody can be trusted. BitTorrent is often considered an example for tit for tat, but this is clearly not the case: BitTorrent does *not* implement real tit for tat, it uses a similar but weaker algorithm. In our proposed modifications of BitTorrent we use the real tit for tat algorithm, which renders free riding much more difficult.

First, we introduce the core concepts of our modifications and explain some basic terms used in the later sections. We then discuss the optimal set of parameters for our algorithms and look at simulation results.

4.2 Related Work

Bram Cohen wrote a paper about BitTorrent talking about incentives building robustness in BitTorrent [2] where he claims that BitTorrent “uses tit-for-tat as a method of seeking pareto efficiency”. He explains how efficient BitTorrent is using the available resources and that it scales well. However, he does not look into the problem of free riders. There exist other papers which look at BitTorrent and its robustness against free riding. [7] considers different approaches to exploit BitTorrent, but their focus lies on the stability and efficiency of the whole torrent swarm and not on the achievements of the free riding node. Also, they fail to use their proposed attacks in combination to gain better performance. [6] tries to solve the free riding problem by introducing a real tit for tat mechanism in BitTorrent and they show that it is fairer. Interestingly, they also note that lack of diversity in available pieces in the network reduces overall tit for tat efficiency; an issue we try to solve using network coding where the space of available pieces is much larger than in classic BitTorrent. Tit for tat was dealt with in [1] where the authors examine different strategies for the iterated prisoner’s dilemma and come to the conclusion that tit for tat seems to be the best strategy if an entity can trust nobody else.

4.3 Core Concepts

Our first idea for an improvement was quite appealing: A seeder sends a piece to a peer for free and then asks for some sort of *proof* that the peer has uploaded the piece to at least one other node. Only then the seeder uploads another piece to that peer. This way, we ensure that the node is actively taking part in the network. However, this *proof* is not an easy thing to realize: In a network where nobody else can be trusted, it is impossible to proof anything. Using a Sybil Attack [3] a malicious peer could generate proofs for non-existent nodes. Or, someone could develop a special client that acts in collusion with its own instances to cheat alien peers.

Another idea was for a seeder to send out data to peers A and B in a way that the data A and B receive is useless by itself but can be "decoded" if A sends all its data to B and vice versa. This, however, suffers from the non-existence of a *fair exchange* algorithm without a trusted third party: As soon as A has sent all its data to B, that peer could refuse to send any data back to A and A would thus gain nothing from this scheme. Tit for tat after all seemed to be the easiest and most effective strategy to implement. To ensure that the network would not run dry we used ideas from network coding to enlarge the space of trading candidates massively.

In order to simplify the explanations below we assume that the "file" being shared forms one big chunk of data that can be split into n pieces p_i of size 2^b bytes. 2^b will be the size of our negotiation units. For comparison, in BitTorrent the pieces are created exactly the same way, but the unit of negotation is only a part of such a piece, a so called block of 16KB (fixed) in size. BitTorrent distributes these n pieces directly, e.g. a peer can request a certain part of a piece from another peer which then sends the data. In our proposed scheme however, we don not exchange these pieces itself. Instead, we use a form of network coding as proposed in [4].

4.3.1 Network Coding

Each piece of the data block is mapped to an integer number in $GF(q)$:

$$p_i \in GF(q)$$

with q prime and $q > 2^{8b}$. The shared data can now be represented as a column vector P:

$$P = \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{pmatrix}$$

The pieces p_i are never sent directly on the wire, instead we form linear combinations C of these pieces in $GF(q)$ and then transmit them:

$$C_{c_i} = c_i P \text{ with row vector } c_i = \{\{0,1\}^n | \#1 = k, \#0 = (n-k)\}$$

The transmission unit u is defined as

$$u_i := (c_i, C_{c_i})$$

c_i is the coefficient vector for the linear combination C_{c_i}. [4] uses entirely random coefficient vectors out of $GF(q)^n$ while we use vectors consisting of exactly $n-k$ zeroes and k ones at random locations.

As in BitTorrent, we use the concept of "have" messages to inform neighbors about available linear combinations. Instead of a single integer number for the piece index, as used in BitTorrent, we need to use another approach for enumerating available linear combinations. We chose to use a k-tuple consisting of the positions of the 1s in the coefficient vector c_i. These tuples are of constant size k and will be communicated to the neighbors as part of "have" messages. Also the "bit field" message used by BitTorrent needs to be enhanced into a list of k-tuples.

Peers will not be able to reconstruct the original data P without downloading a set of $m \geq n$ distinct linear combinations $U := \{u_0, u_1, \ldots, u_{m-1}\}$. Peers can then form a matrix $M \in GF(q)^{mn}$ built out of row vectors as follows:

$$M = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix}$$

The clients then need to solve a linear equation system of the following form:

$$MP = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{pmatrix} = \begin{pmatrix} C_{c_0} \\ C_{c_1} \\ \vdots \\ C_{c_{m-1}} \end{pmatrix}$$

The original data P can be reconstructed from this linear system if and only if the rank of M is n.

The transmission units are exchanged between peers according to strict tit for tat rules: Every peer uploads at most one transmission unit more than it has downloaded from a remote peer. The only exception to this rule are the seeders, peers which have obtained the whole file. In BitTorrent, the seeders upload data to any peer in a round-robin way which is perfect for free riders. We tend to use a slightly different approach in our proposal explained in one of the next sections.

BitTorrent uses a rarest first piece selection strategy: A peer determines which pieces of the remote end are interesting and then fetches the one that is found least often in its neighbor set. For network coding this strategy does not necessarily make sense, as there is no finite well defined set of linear combinations needed. Instead we chose a greedy algorithm: We keep a column vector $r \in 0,1^n$ at each node which counts how many times a piece p_i is occuring within our linear combinations. Upon receiving a new linear combination c_i we update the vector r by adding c_i component wise. If a component in r becomes 1, we add a constant threshold value of kn to the component. When deciding which linear combination c_j from a remote peer we should fetch we calculate the following score for each combination available at the other peer:

$$s_j = c_j r$$

We then choose the combination c_j with the lowest score s_j. That way we ensure that the linear combinations we fetch cover as many pieces as possible and that the matrix M contains only non-zero columns.

4.3.2 Homomorphic Hash Function

Ensuring data integrity is becoming more difficult when using network coding: The torrent metafile cannot just contain SHA1 hashes for each piece as there is no way to combine these into a hash for a linear combination of pieces. Storing the SHA1 hashes of all possible $\binom{n}{k}$ linear combinations would be another option but simply not feasible for any $k > 1$ because of the huge amount of storage space needed.

A better solution to this problem has been presented in [5] with the use of a *homomorphic* hash function. The torrent metafile still contains special hashes for each piece which can then be recombined to derive the hash of a linear combination. Once a peer receives a linear combination it can immediately check the validity and malicious peers uploading random garbage can effectively be banned. This is important as otherwise corrupted linear combinations would be spread all over the network and would make reconstruction impossible.

4.3.3 Fast Extension

As part of the BitTorrent fast extension[1] the inventors introduced a mechanism for creating a set of piece indices based on the IP address that peers can request for free. We are using a very similar approach in our extension: Once a peer connects to a seeder, the seeder uses the IP address[2] of the remote peer to seed a pseudo random number generator which will compute a set of $\xi \leq n$ linear combinations, where n denotes the numer of pieces the file is divided in. These linear combinations will then be announced to the connecting peer. This mechanism offers two advantages:

- Peers with different IP addresses obtain entirely different linear combinations from a seeder. This helps in keeping the available number of distinct linear combinations in the network high, which is an important property as we will see in the next chapter.

[1]`http://bittorrent.org/fast_extensions.html`

[2]To make this more robust against peers that switch IPs often one might want to use the class C or even class B network address. That way, a client would probably need to change the ISP in order to get an IP from an entirely different B or C network.

- By making ξ depend on the size of the swarm it is possible to successfully eliminate free riding. For example in a swarm with > 50 peers a seeder does not need to upload more than, say, $\frac{n}{10}$ linear combinations to a peer, as the peer itself can easily retrieve more linear combinations by trading with neighbors. The combinations it receives from the seeder form an important starting set for the peer. As a matter of fact, the chances that any other peer already possesses one of the linear combinations in this starting set are very small and thus trading these combinations will be easy. Because the random linear combination set will be based on a peer's IP, it will not be able to get new combinations by trying another seeder or by reconnecting to the same one.

Estimating the total network size can be hard but we think that this is not necessary: Seeders should choose their ξ based on the number of *open connections* they have, which is, more or less, proportional to the overall network size.

4.4 Optimal Parameter Choice

Our proposed enhancements of BitTorrent are subject to a number of parameters. The most important ones being n, the number of pieces data is split into, and k, the number of pieces combined in a linear combination. In the following sections we provide upper and lower bounds for these two parameters.

4.4.1 Linear Combination Size

The choice of k is crucial: if k is too small, for example 1, the whole advantage of using network coding is gone. On the other hand, if k is very large, for example $k = \frac{n}{2}$, computation of the linear combinations is becoming very expensive: To combine k data pieces, the seeder needs to read $k2^b$ bytes from disk. With $k = \frac{n}{2}$ this is half the total data amount for every single linear combination. With torrents often being larger than 1GB this is not feasible nowadays. For performance reasons k should be as small as possible. On the other hand, in terms of stability, it should be large.

k is important for the rank of matrix M. Clearly, M cannot have a rank of n if there is an all zero column. We now present a lower bound on the probability that every original data piece is part of at least one linear combination in a randomly drawn set of n combinations of k pieces each. Or, simpler: The probability that M has non-zero columns only.

$$P(\text{piece } i \text{ is chosen in a random linear combination}) = \frac{\binom{n-1}{k-1}}{\binom{n}{k}} = \frac{k}{n}$$

$$P(\text{piece } i \text{ is chosen in at least one of the } n \text{ combinations}) = 1 - (1 - \frac{k}{n})^n$$

Unfortunately, the later probability cannot be extended to the probability $P(\text{all pieces are chosen in at least one of the } n \text{ combinations})$ because the individual probabilities are *not* independent. We therefore adjust the process of drawing random combinations to prove a lower bound: Instead of choosing the combination among all possible $\binom{n}{k}$ combinations, we pick k random pieces independently with probability $\frac{1}{n}$. If we happen to pick a piece more than once, we only add it once to the linear combination. The resulting combination will therefore have a lower or equal number of components than one from $\binom{n}{k}$ and thus the probability that a given piece i is part of a combination will be less or equal to $\frac{k}{n}$:

$$P(\text{piece } i \text{ is chosen in a random linear combination}) = 1 - (1 - \frac{1}{n})^k \leq \frac{k}{n}$$

$$P(\text{piece } i \text{ is chosen in at least one of the } n \text{ combinations}) = 1 - \left((1 - \frac{1}{n})^k\right)^n = 1 - (1 - \frac{1}{n})^{kn}$$

$$P(\text{all pieces are chosen in at least one of the } n \text{ combinations}) = \left(1 - (1 - \frac{1}{n})^{kn}\right)^n$$

A plot of this probability function can be seen in Figure 4.1. To determine the best k, we need to reformulate the equation above into a function returning k based on the desired probability p and the number of pieces n.

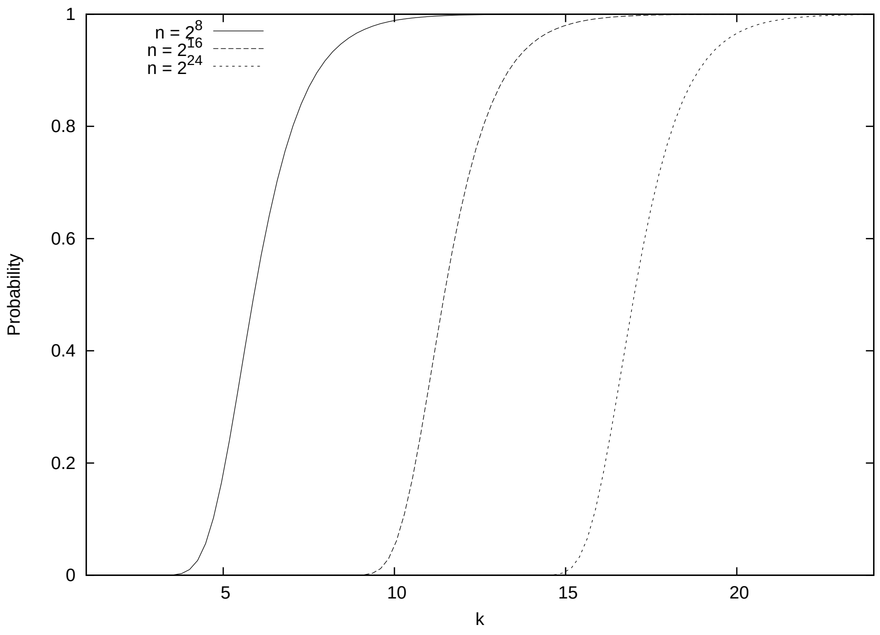


Figure 4.1: This graph shows the probabilities that every piece is chosen at least once in a set of n linear combinations of k components.

$$(1 - \frac{1}{n})^{kn} \approx e^{-k} \text{ for large } n$$

$$p = \left(1 - (1 - \frac{1}{n})^{kn}\right)^n \approx (1 - e^{-k})^n$$

$$k \approx -\log(1 - \sqrt[n]{p})$$

Plotting this function yields the graph visible in Figure 4.2. The number of pieces that need to be combined in linear combinations grows logarithmically with the error probability that there exists a piece which is not part of n randomly selected linear combinations. For all practical purposes, a $k \leq 32$ seems to be sufficient, as $n > 2^{32}$ is not feasible and an error of 10^{-6} is acceptable.

While having each data piece in at least one linear combination is a prerequisite for the matrix M to be of rank n it is in no way *sufficient*: We need a formula to calculate the probability that a matrix M built by n random linear combinations of size k has a rank of n. We tried solving this problem but did not come to an end. Our simulations in MATLAB always calculated a rank of n if every piece was part of at least one linear combination and we thus believe that the resulting system of linear equations will be solvable with high probability.

In [4] the authors use purely random linear combinations: The coefficient vectors c_i are drawn from $GF(q)^n$ in their case. This certainly improves the chances that M is of rank n but on the other hand it also makes computing the combinations very expensive. A seeder would need to read the whole shared file for every single linear combination it generates. This is not practical for files larger than a couple of MB. With our scheme of only k ones in a coefficient vector we can build the combinations much more efficiently: In a sharing scenario with $k = 32$ and $2^b = 64$KB a seeder needs to read $k2^b = 2$MB from disk for each linear combination calculated. Common hard disks support read operations at up to 50MB. The seeder therefore needs at least $\frac{1}{25}$ seconds to generate a combination. This results in at most 25 linear combinations that can be uploaded by a seed, which is a rate of about 13Mbit/s. This should be sufficient for most applications. The rate could further be improved by caching some of the file content in RAM although probably with a limited effect: The pieces that need to be read are distributed completely random over the file and thus a cache will not be able to reduce disk IO that much.

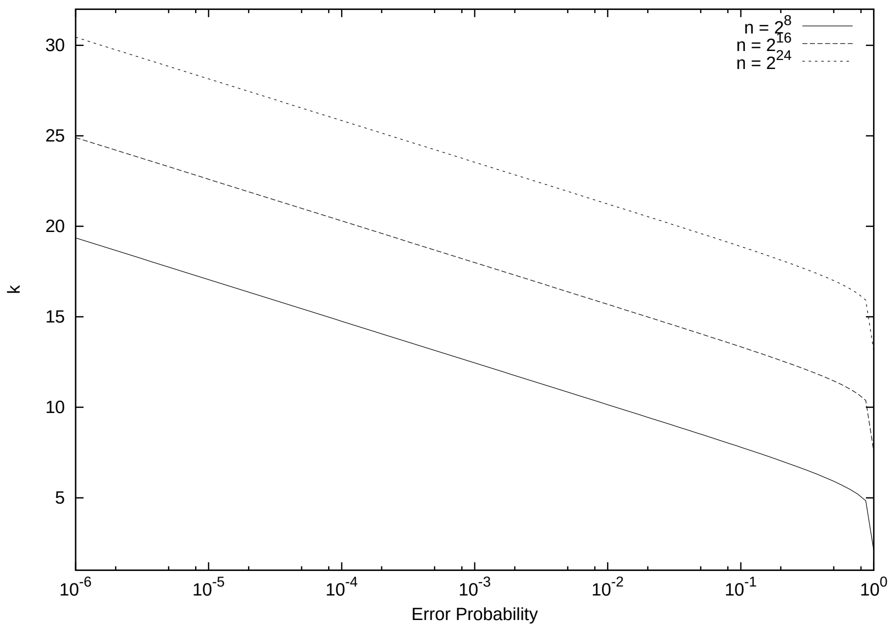


Figure 4.2: Optimal choice of k based on the error probability $1 - p$ that a piece is not part of any of the n linear combinations of size k.

4.4.2 Choice of n

Given a fixed size data chunk to be shared, n directly influences the transfer unit size. If the transfer unit size is chosen too small the protocol will introduce a significant overhead. A size $2^b \geq 4$KB seems reasonable. On the other hand, choosing b too large will have two disadvantages:

- The transfer unit size is atomic, i.e. a unit is either transferred completely or not at all. Partial transmits will not be possible. Ideally, it would take even slow peers only a couple of seconds to upload such a unit to a remote peer. Assuming that rather slow connections can upload about 16KB a second and that they are uploading to multiple peers in parallel, a unit size of $2^b \leq 64$KB makes sense.
- Because of the tit for tat algorithm used a malicious peer can get one transfer unit worth of data *free* from every peer. If the unit size is too large it might be possible to download the whole shared file in a torrent with lots of peers by only profiting from the one-linear-combination-for-free offer at each peer. The transfer unit size should therefore be less than $\frac{1}{\text{total number of peers downloading a torrent over time}}$ of the total data chunk size.

A typical file that might be shared over our network is a 5GB DVD Rip of some movie with a Creative Commons[3] license. To achieve a transfer unit size of 64KB we will need to set n according to:

$$n \geq \frac{5\text{GB}}{64\text{KB}} = 81920$$

4.4.3 Advantages

Trading Unit Diversity

Network coding offers a big advantage over the usual piece-by-piece sharing methods: The space of existing trading candidates grows much bigger and this helps keeping tit for tat running. The following example should clarify this a bit:

Consider two peers, A and B. Both have already downloaded $\frac{n}{2}$ transfer units from distinct sources and we thus assume they have downloaded a random subset of the possible units each. If we would share

[3]http://creativecommons.org/

the BitTorrent way, both peers would have already completed half the shared data chunk. Peer A is, by expectation, interested in about half of the units peer B has got and vice versa. They will thus play tit for tat until they both aquire $\frac{3n}{4}$ units after which they will need new peers to share with.

On the other hand, if we use network coding, the situation looks entirely different: The space of possible units is massively bigger, $\binom{n}{k}$ to be precise. A is therefore, by expectation, interested in about $\frac{n}{2}\left(1 - \frac{n}{2\binom{n}{k}}\right)$ units that B possesses and they will thus be able to almost finish their download by playing fair tit for tat between each other.

This case is idealistic, however. In a real system they will not hold entirely distinct units, as they probably received their units from common peers. Here comes the diversity we mentioned earlier into play: The system as a whole should have as many distinct linear combinations around as possible in order to maximize the probability that two peers are interested in each other. Our algorithm for seeders is a good step into that direction by seeding random linear combinations to different peers.

Endgame

An endgame as in BitTorrent will not be necessary when using network coding because the linear combinations that need to be fetched towards the end are in no way different to the ones earlier: They are completely random. As long as the linear combination diversity in the network is much larger than n we do not need to do any special handling of the endphase which further simplifies the protocol.

4.4.4 Implications

With a choice of $k = 26, n = 2^{17}$ we reach a high probability of getting a matrix M with rank n. The resulting system is a stable, fair peer-to-peer network. There is only one catch left: Solving the linear equation system. This is a trivial step for, say, up to 2^{12} equations on today's computers. However, for $n = 2^{17}$ the solving process would take ages to complete using a Gaussian elimination scheme, as the time complexity is in $O(n^3)$. We therefore need a faster way of solving these systems. After all, the matrix M is *very* sparse, with only k entries in a row of length n. In our proposed example the "load factor" is $k/n = \frac{26}{2^{17}} \approx 0.0001984$ and we assume that there has to be a more efficient way to solve this kind of systems. Unfortunately we were not able to come up with any nor did we find a promising approach.

If the system cannot be solved on a sparse matrix even memory usage becomes an issue: A 16384×16384 matrix barely fits into 1GB of RAM. A simple benchmark using MATLAB's `rank` function to determine the rank of matrices for $64 \leq n \leq 4096$ yields the running times visible in Figure 4.3. If we extrapolate these values to the case of $n = 2^{17}$ we reach a running time of approximately 61 days, provided the whole matrix still fits into memory! This clearly illustrates how impractical this scheme without better suited solving algorithms is.

4.5 Evaluation

We have performed several simulations of our proposed protocol in different scenarios. We always used a file consisting of $n = 4096$ pieces with a piece size of $2^b = 64$KB. Linear combinations were formed out of $k = 8$ pieces and the simulated peers had uniformly distributed upload bandwiths and downstream bandwiths slightly larger. We also experimented with different NAT peer percentages, that is peers that cannot receive incoming connections. The two main scenarios simulated are characterized below:

- One seeder which remains active the whole time and up to two thousand peers arriving in random exponentially distributed intervals. The nodes leave the network as soon as they have downloaded n linear combinations
- The same as above, but the seeder leaves the network immediately after it has pushed $4n$ linear combinations into the network.

The first scenario works very well and all peers are able to finish their downloads. This is due to the fact that the seeder is staying online all the time and can help the peers out if they should ever be stuck with pure

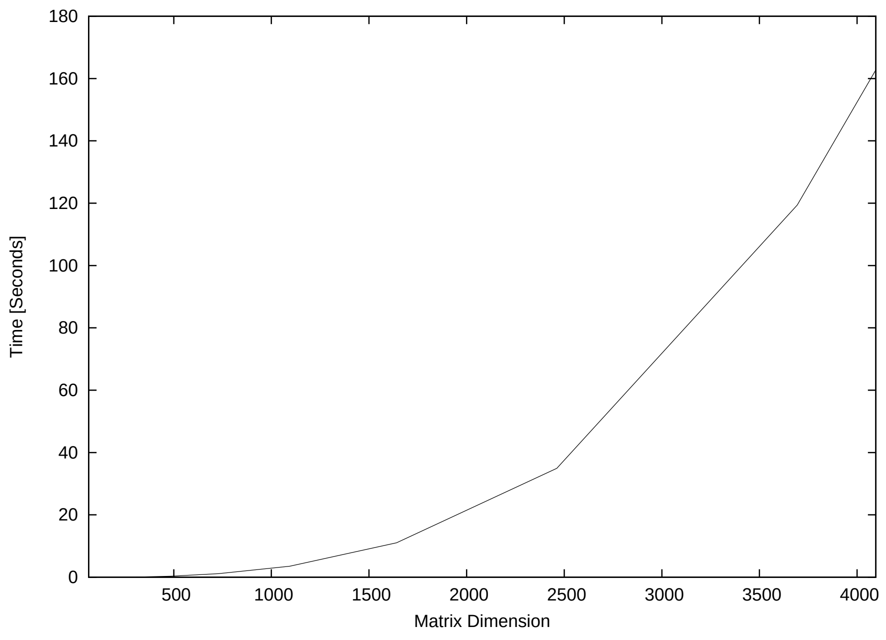


Figure 4.3: Rank calculation times in MATLAB on a 3GHz Pentium 4 computer with 1GB of RAM. The rank of $n \times n$ matrices with $k = 32$ ones in each row was calculated in the real number space, *not* on a finite field.

tit for tat. Surprisingly, the second scenario works very reliably as well. The seeder stays online for only a fraction of the time the network is online. After that, the network is all on its own and all the peers exchange data using strict tit for tat. In Figure 4.4 we compare the diversity of all available linear combinations in the network for these two scenarios. It comes with no suprise that the scenario with the constant seeder features a more stable diversity. However, the diversity of the second scenario is deteriorating very slowly and remains mostly stable. Only towards the end, when no new peers are joining, the quality suffers rapidly.

We can back up the discoveries made in [6] concerning the correlation between download duration and upload speed: In BitTorrent, peers with fast upstreams do not necessarily get the same high downstream they deserve. When using real tit for tat, the upload and download rates correlate strongly as can be seen in Figure 4.5. The relation between upstream and download duration is almost ideal, except for the case with many firewalled peers: Those peers fail to open a sufficient number of connections and therefore need significantly more time.

The calculated network overhead (the total amount of bytes sent by all peers divided by the size of the shared file times the number of peers) was usually between 1.1 and 1.15. This does not include TCP/IP overhead but it does include all user data that was transmitted such as handshake messages, having messages, requests, etc.

To demonstrate the effectiveness of network coding we implemented a BitTorrent like file sharing system using strict tit for tat for comparison. As a piece selection algorithm we tried both, rarest first and random piece. The random piece selection performed much worse than rarest first and is thus not considered anymore in the following discussion. We compared a network with 300 nodes joining according to a poisson distribution with the following expectations λ_i:

$$\lambda_i = 10h/(300 - i)$$

This results in a rush period at the beginning and then steadily decreasing peer arrivals. In both scenarios we added one seeder at the beginning which leaves the network after having uploaded $4n$ units and all the other nodes leave the network as soon as they have downloaded the whole file.

The challenge in such a network lies in keeping the unit diversity as large as possible despite the slow peer arrival rate. Figure 4.6 clearly shows the difference between the two schemes: In the network coding case 270 peers finish their download while the BitTorrent simulation ends successfully for a mere 16 nodes. The seeder left the network after 1:40h in the network coding case and after 1:20h in the BitTorrent simulation.

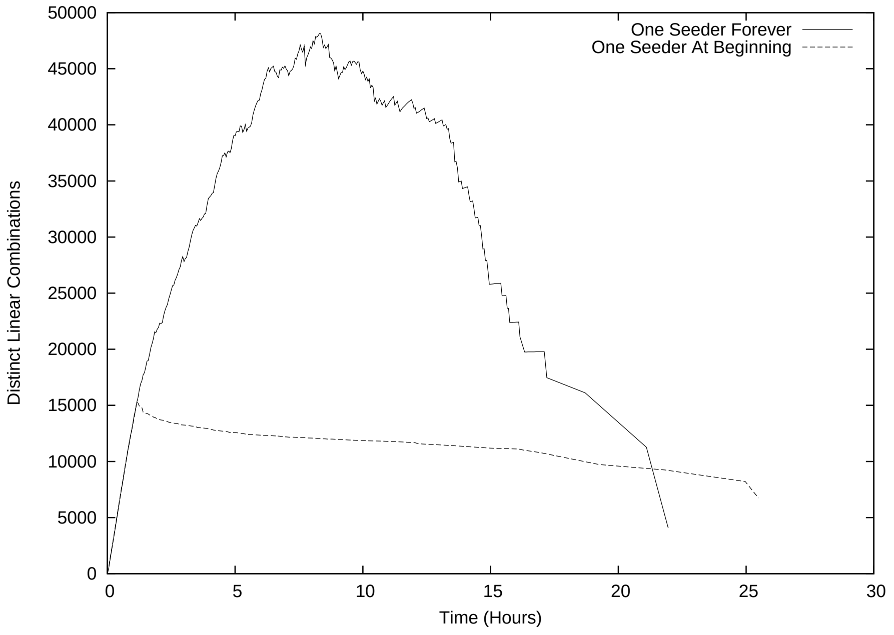


Figure 4.4: The number of distinct linear combinations available in the whole network. The more diverse this set is the better, as tit for tat partners will find more linear combinations to trade. In classic BitTorrent, the size of this set would be constant $n = 4096$ in this case.

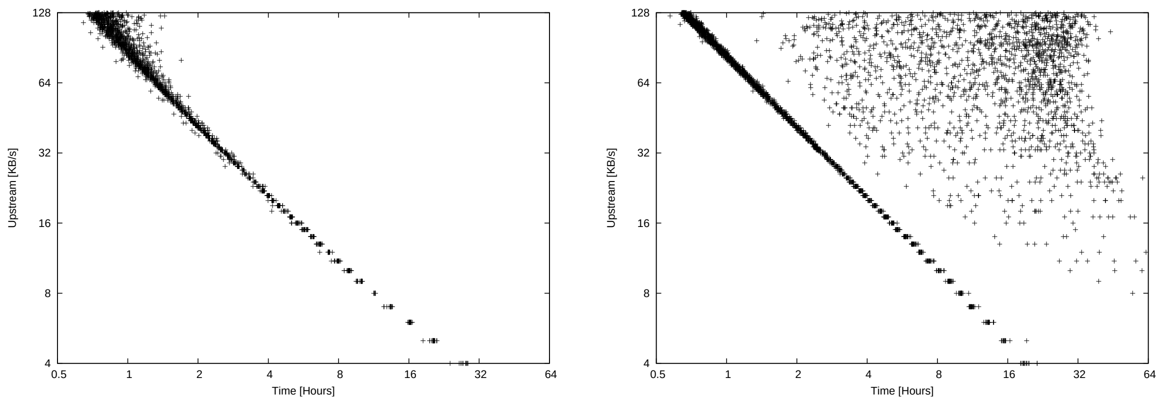
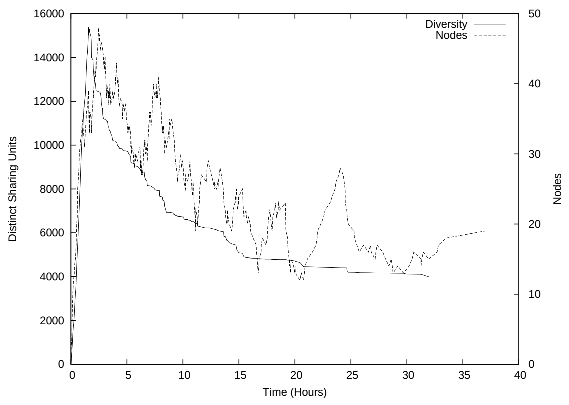


Figure 4.5: Download duration vs. upstream speed for each individual peer. The graph on the left was simulated with 10% firewalled peers, while the second one used 60% firewalled peers. The peers in the "cloud" on the top right in the right graph are the firewalled peers: They do not get the performance they could if they were directly connectable from the outside. The reason for this is clear: It is more difficult for these peers to build up a decent number of connections as they can only connect to non-firewalled peers.



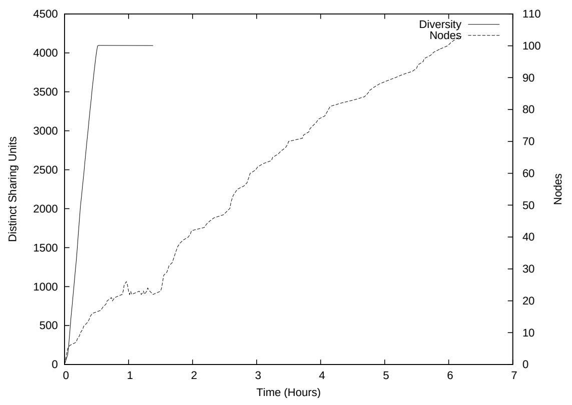


Figure 4.6: Both graphs show the number of active peers and the number of distinct exchange units over time. On the left is the simulation run with network coding ($k = 8$) and on the right the BitTorrent run ($k = 1$) with rarest first strategy.

Already 3 minutes after the seeder left the network degraded and was in a situation it could not recover from. In contrast, the network coding scheme continued to work for 30 more hours before it broke down.

We repeated these runs but the outcome was always the same: Network coding effectively helps in keeping a network diverse. The parameter for the poisson arrival process (expectation of node inter-arrival intervals) is the key factor which determines whether a network will survive or not. If the nodes arrive close enough, both networks do not have a problem working until the end. But if the time between node arrivals grows larger it becomes more difficult as there are nodes leaving the system which take a lot of “knowledge” with them that cannot be replaced.

4.6 Conclusion

Most existing peer-to-peer file sharing systems have been of relatively primitive nature: They focus on speed and ease of implementation. Network coding is in our opinion a step into a new direction: The protocol makes use of sophisticated mathematical constructs which indeed improve the overall quality of the network. This added value comes with a downside, the complexity of the numerical operations involved. Solving a linear equation system with more than 2^{16} unknowns is not an easy task on an ordinary personal computer. Further, some of our assumptions about the rank of the matrix M might be too optimistic, we certainly need to spend more time investigating the exact mathematical problems. Nevertheless, network coding looks like a good addition to existing peer to peer systems and we are confident that there lies great potential that can be utilized to build more robust and fair systems.

Chapter 5

Further Work

5.1 BitThief

Suggestions for improving our free riding BitTorrent client.

5.1.1 Distributed Tracker Protocol

The distributed tracker maintenance code is already in place and working, though it is never enabled at runtime. Still missing is the code to “resolve” a key into a value in the distributed hash table (DHT) and we are thus unable at the moment to get any peer information out of the system. This is a relatively easy task and should be combined with others from below.

5.1.2 Resuming Download Functionality

If BitTorrent should crash or you want to actively pause a currently running download there is no automatic way to resume the download later on. Instead, you need to load the exact same torrent metainfo file again and point BitThief to the same download location you used previously to continue the download. It would be nice if BitThief could actually remember the currently running downloads and resume them automatically.

Part of this work could also be an improved file validation algorithm: Instead of checking every piece on disc against the hash in the metafile on startup we could somewhere store the pieces already downloaded correctly and thus speed up resume operations significantly, especially on large torrents.

5.1.3 Improving GUI

A lot of parameters can only be set on the command line of BitThief. As most people tend not to like command lines it would be helpful if some of the options could be adjusted in the graphical UI. Another helpful UI display would be an estimated time of completion, which is straightforward to implement.

5.1.4 Connection Encryption

More and more BitTorrent clients are using encrypted connections to bypass ISP throttling mechanisms. Implementing connection encryption for BitThief would enable BitThief to connect to those peers also that allow only encrypted connections.

5.1.5 Partial Piece Storage

BitThief keeps the partially downloaded pieces in memory at the moment. This has several disadvantages:

- If pieces are big ($>$ 1MB) and a lot of them only partially available BitThief might run out of memory.

- If BitThief should crash, all the partially downloaded pieces are lost. This can be a couple of MB wasted.

A simple solution is to store the partially downloaded pieces on disk and to implement a mechanism for restoring the partial pieces after a crash.

5.2 BitTorrent

A couple of possible further research and implementation projects based on BitTorrent.

5.2.1 Hash Tree Extension

As mentioned earlier, BitTorrent's integrity checking mechanisms are not that sophisticated. The existing algorithms can only verify data on a piece level which is up to a few MB in size, while the exchange unit size is a mere 16KB. A possible solution is storing hashes of every 16KB sub-piece in the torrent metafile. Assuming a hash size of 128 bits this would blow a metafile for a 4GB shared file up to 4MB.

Another method is the idea of hash trees or merkle trees: In the torrent metafile we store only a 128 bit hash tree root. We then could introduce a new piece message format which does not only contain the 16KB piece data but also the required hashes from the hash tree in order to verify the integrity of the piece. Assuming a binary hash tree such a piece message would need to contain 18 additional hashes in the case of a 4GB torrent. At 128 bits per hash this is 288 bytes per 16KB sub-piece; a 2% overhead. A 2% overhead per sub-piece exchanged results in a total overhead of about 80MB which might seem like a lot. However, this would make the protocol entirely secure against peers sending corrupted data while even shrinking the metafile size.

Using this scheme a lot of redundant data would be transferred. Instead of adding the hashes needed for verification to the piece message we could introduce a new set of messages for requesting hashes of specific tree nodes which would reduce the overhead for the price of a more complicated design.

The extension could be implemented for Azureus as well as the mainline client in a totally backwards compatible way.

5.2.2 Proof of Concept Implementation of Network Coding

Simulations are a good tool to get a first idea on how things might work. Modelling the Internet is a difficult task however and tends to be oversimplified. According to Bram Cohen strict tit for tat does not work in a real world of peer-to-peer file sharing. We therefore suggest implementing a small test application which uses the ideas from network coding we presented for sharing small files, in order not to run into memory or processing capacity issues.

The application could be implemented on top of BitThief or as a separate stand-alone system and then be deployed among friends and families. We suggest implementing the system without integrity checks first and see how it works. After all there should not be too many malicious peers among your friends and families...

5.2.3 Algorithms for Efficient Solving of Sparse Linear Equation Systems

The problem with network coding schemes clearly is scalability in terms of file size. For efficient sharing of large files we need to be able to work with giant sparse matrices efficiently. A time complexity of $O(n^3)$ is not practical while $O(kn^2)$, the complexity for solving linear equation systems with band matrices, could work out. This task is of pure mathematical nature and independent of any other work.

5.2.4 Proof for Matrix Rank

Another interesting mathematical question is the expectation of the rank of random matrices. Specifically: What is the probability that a $m \times n$ matrix M in $GF(q)^{m \times n}$ with [non-zero columns and] exactly k ones on each row has rank n? The answer to this question is very important for the network coding scheme to work: If this probability is too small the peers would need to fetch more than n linear combinations in order to solve the linear equation system at the end. This reduces overall protocol efficiency by quite a bit.

Appendix A

BitTorrent Protocol Message Formats

Table A.1: Format of the Handshake Message

Offset	0	1	20	28	48
Length	1	19	8	20	20
Content	19	"BitTorrent protocol"	0x00..00	0x00..00	0x00..00
Description	Protocol Length	Protocol Name	Reserved Bytes	Info Hash	Peer ID

Table A.2: Format of the Bitfield Message

Offset	0	4	5
Length	4	1	N
Content	$N+1$	5	0x00..00
Description	Length	Type	BitField

Table A.3: Format of the Have Message

Offset	0	4	5
Length	4	1	4
Content	5	4	0x00..00
Description	Length	Type	Piece Index

Table A.4: Format of the Keep-Alive Message

Offset	0
Length	4
Content	0
Description	Length

Table A.5: Format of the Choke Message

Offset	0	4
Length	4	1
Content	1	0
Description	Length	Type

Table A.6: Format of the Unchoke Message

Offset	0	4
Length	4	1
Content	1	1
Description	Length	Type

Table A.7: Format of the Interested Message

Offset	0	4
Length	4	1
Content	1	2
Description	Length	Type

Table A.8: Format of the Not-Interested Message

Offset	0	4
Length	4	1
Content	1	3
Description	Length	Type

Table A.9: Format of the Request Message

Offset	0	4	5	9	13
Length	4	1	4	4	4
Content	13	6	0x00..00	0x00..00	2^{14}
Description	Length	Type	Piece Index	Offset	Length

Table A.10: Format of the Cancel Message

Offset	0	4	5	9	13
Length	4	1	4	4	4
Content	13	8	0x00..00	0x00..00	2^{14}
Description	Length	Type	Piece Index	Offset	Length

Table A.11: Format of the Piece Message

Offset	0	4	5	9	13
Length	4	1	4	4	N
Content	$N + 9$	7	0x00..00	0x00..00	0x00..00
Description	Length	Type	Piece Index	Offset	Data

Appendix B

Paper: Free Riding in BitTorrent is Cheap

On the following 6 pages we present our paper about free riding in BitTorrent. The paper was submitted on August 11 for the “Fifth Workshop on Hot Topics in Networks” (HotNets-V)[1] in Irvine, California.

[1]http://www.acm.org/sigs/sigcomm/HotNets-V/

Free Riding in BitTorrent is Cheap

Thomas Locher, Patrick Moor, Stefan Schmid, Roger Wattenhofer
{lochert@tik.ee., pmoor@, schmiste@tik.ee., wattenhofer@tik.ee.}ethz.ch
Computer Engineering and Networks Laboratory (TIK), ETH Zurich, 8092 Zurich, Switzerland

ABSTRACT

We address the question whether it is possible to download entire files without reciprocating in BitTorrent. To this end, we developed BitThief, a free riding client that never contributes any real data. Our findings suggest that downloading without sharing is indeed feasible and that simple tricks suffice in order to achieve high download rates, even in the absence of seeders. Moreover, we show that sharing communities provide many incentives to cheat. Finally, we illustrate how peers in a swarm react to various sophisticated attacks.

1 INTRODUCTION

As pure peer-to-peer (p2p) systems are completely decentralized and resources are shared directly between participating peers, all p2p systems potentially suffer from *free riders*, i.e. peers that eagerly consume resources without reciprocating in any way. Not only do free riders diminish the quality of service for other peers, but they also threaten the existence of the entire system.

For that reason, it is crucial for any system without a centralized control to incorporate a rigorous incentive mechanism that renders freeloading evidently unattractive to selfish peers. Unfortunately, however, many solutions so far either could easily be fooled or were unrealistically complex. Bram Cohen's BitTorrent protocol heralded a paradigm shift as it demonstrated that cooperation can easily be fostered among peers interested in the same file and that concentrating on one file is often enough in practice. The fair sharing mechanism of BitTorrent is widely believed to successfully undermine freeloading behavior.

Contrary to such belief, we show that BitTorrent in fact does not provide sufficient incentives to rule out free riding. The large degree of cooperation observed in BitTorrent swarms is mainly due to the widespread use of obedient clients which willingly serve all requests from other peers. We have developed our own BitTorrent client *BitThief*[1] that never serves any content to other peers. With the aid of this client, we demonstrate that a peer can download content fast *without uploading any data*. Surprisingly, BitThief always achieves a high download rate, and in some experiments has even outperformed the *official client*. Moreover, while *seeders* ("altruistic peers") clearly offer the opportunity to freeload, we are even able to download content quickly if we ignore seeders and download solely from other peers that do not possess all pieces of the desired content (*leechers*). This implies that the basic piece exchange mechanism does not effectively restrain peers from freeloading.

Sharing communities are also investigated in this paper. By banning users with constantly low sharing ratios or by denying them access to the newest torrents available, such communities encourage users to upload more than they download, that is, to keep their sharing ratio above 1. We will show that sharing communities are particularly appealing for free riders, and that cheating is easy.

We believe that the possibility to freeload which does not come at the cost of reduced quality of service (e.g., download rate) is attractive for users: Not only because wasting more expensive upload bandwidth is avoided, but also because media contents such as music or video shared in p2p networks may be subject to copyrights. However, as more and more users decide to free ride, the usefulness of a p2p system will decline quickly. Thus, spreading such freeloading clients might prove to be an efficient attack for corporations fighting the uncontrolled distribution of their copyrighted material.

2 BITTORRENT

The main mechanisms applied by *BitTorrent* are described in [4]; for additional resources including a detailed technical protocol, the reader is referred to *www.bittorrent.org*. Basically, BitTorrent is a p2p application for sharing files or collections of those. In order to participate in a *torrent download*, a peer has to obtain a *torrent metafile* which contains information about the content of the torrent, e.g. file names, size, tracker addresses, etc. A tracker is a centralized entity that keeps track of all the peers (TCP endpoints) that are downloading in a specific torrent swarm.[2] Peers obtain contact information of other participating peers by announcing themselves to the tracker on a regular basis. The data to be shared is divided into pieces whose size is specified in the metafile (usually a couple of thousand pieces per torrent). A hash of each piece is also stored in the metafile, so that the downloaded data can be verified piece by piece. Peers participating in a torrent download are subdivided into *seeders* which have already downloaded the whole file and which (altruistically) provide other peers with any piece they request, and *leechers* which are still in progress of downloading the torrent.

[1]Available at http://dcg.ethz.ch/projects/bitthief/.

[2]Recently, a distributed tracker protocol has been proposed and implemented by most modern clients.

While seeders upload to all peers (in a round robin fashion), leechers upload only to those peers from which they also get some pieces in return. The peer selection for uploading is done by unchoking a fixed number of peers every ten seconds and thus enabling them to send requests. If a peer does not contribute for a while it is choked again and another peer is unchoked instead.

The purpose of this mechanism is to enforce contributions of all peers. However, each leecher periodically *unchokes* some neighboring leecher, transferring some data to this neighboring peer for free. This is done in order to allow newly joined peers without any pieces of the torrent to *bootstrap*. Clearly, this unchoking mechanism is one weakness that can be exploited by BitThief.

3 BitThief: A Free Riding Client

In this section we provide evidence that, with some simple tricks, uploading can be avoided in BitTorrent while maintaining a high download rate. In particular, our own client *BitThief* is described and evaluated. BitThief is written in Java and is based on the official implementation[3] (written in Python, also referred to as *official client* or *mainline client*), and the *Azureus*[4] implementation. We kept the implementation as simple as possible and added a lot of instrumentational code to analyze our client's performance. BitThief does not perform any chokes or unchokes of remote peers, and it never announces any pieces. In other words, a remote peer always assumes that it interacts with a newly arrived peer that has just started downloading. Compared to the official client, BitThief is more aggressive during the startup period, as it re-announces itself to the tracker in order to get many remote peer addresses as quickly as possible. The tracker typically responds with 50 peer addresses per announcement. This parameter can be increased to at most 200 in the announce request, but most trackers will trim the list to a limit of 50. Tracker announcements are repeated at an interval received in the first announce response, usually in the order of once every 1800 seconds. Our client ignores this number and queries the tracker more frequently, starting with a configurable interval and then exponentially backing off to once every half an hour. Interestingly, during all our tests, our client was not banned by any of the trackers and could thus gather a lot of peers. The effect of our aggressive behavior is depicted in Figure 1. Finally, note that it would also be possible to make use of the *distributed tracker protocol*.[5] This protocol is useful if the main tracker is not operational. Thus far, we have not incorporated this functionality into our client however.

Having a large number of open connections improves the download rate twofold: First, connecting to more seeders allows our client to benefit more often from their round robin unchoking periods. Second, there will be more leechers in our neighborhood that include BitThief in their periodical optimistic unchoke slot. Opening more connections increases download speed linearly, as remote peers act independently of the number of our open connections. However, note that opening two connections to the same peer does not help, as the official client, Azureus, and presumably all other clients as well immediately close a second connection originating from the same IP address.

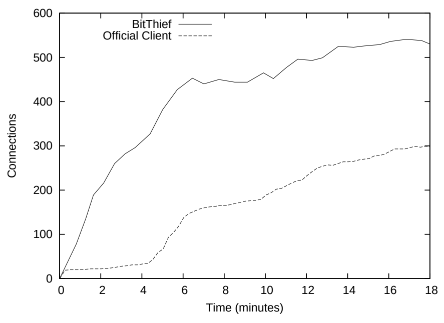


Figure 1: Number of open connections over time: In comparison to the official client, BitThief opens connections much faster.

The piece selection algorithm is simple: We fetch whatever we can get. If our client is unchoked by a remote peer, it picks a random missing piece. Further, we strive to complete the pieces we downloaded partially as soon as possible in order to check them against the hash from the metafile and write them to the harddisk immediately. Our algorithm ensures that we *never* leave an unchoke period unused.

3.1 Seeders

We first tested the client on several torrents obtained from *Mininova*[6] and compared it to the official client.[7] By default, the official client does not allow more than 80 connections. In order to ensure a fair comparison, we removed this limitation and permitted the client to open up to 500 connections. In a first experiment, we did not impose any restrictions on our client, in particular, BitThief was also permitted to download from seeders. The tests were run on a PC with a public IP address and an open TCP port, so that remote peers could connect to our client. We further blocked all network traffic to or from our university network, as this could bias the measurements. The properties of the different torrents used in this experiment are depicted in Table 1. Note that the tracker information is not very accurate in general and its peer count should only be considered a hint on the actual number of peers in the torrent.

[3]See http://bittorrent.com/.

[4]See http://azureus.sourceforge.net/.

[5]See http://www.bittorrent.org/Draft_DHT_protocol.html.

[6]See http://www.mininova.org/.

[7]Official client vers. 4.20.2 (linux source). Obtained from bittorrent.com, used with parameters: `--min_peers 500 --max_initiate 500 --max_allow_in 500`.

	Size	Seeders	Leechers	μ	σ
A	170MB	10518 (303)	7301 (98)	13	4
B	175MB	923 (96)	257 (65)	14	8
C	175MB	709 (234)	283 (42)	19	8
D	349MB	465 (156)	189 (137)	25	6
E	551MB	880 (121)	884 (353)	47	17
F	31MB	N/A (29)	N/A (152)	52	13
G	798MB	195 (145)	432 (311)	88	5

Table 1: Characteristics of our test torrents. The numbers in parentheses represent the maximum number of connections BitThief maintained concurrently to the respective peer class and is usually significantly lower than the peer count the tracker provided. μ and σ are the average and standard deviation of the official client's download times in minutes. The tracker of Torrent F did not provide any peer count information. Based on the number of different IP addresses our client exchanged data with, we estimate the total number of peers in this torrent to be more than 340.

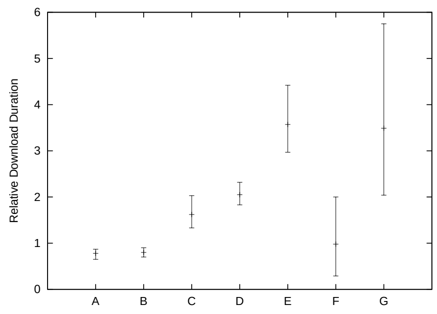


Figure 2: Relative download times for six torrents. The download time of the official client is normalized to 1.0. Every torrent was downloaded three times with both clients. The plot shows relative download times with the fastest run at the lower end of the bar, the average running time at the level of the horizontal tick mark, and the slowest run at the upper end of the bar.

The results are summarized in Figure 2. As a first observation, note that in every experiment, BitThief succeeded eventually to download the entire file. More interestingly, the time required to do so is often not much longer than with uploading! Exceptions are Torrents E and G, where there are relatively few seeders but plenty of leechers. In that case, it takes roughly four times longer with our client. However, the download came at a large cost for the official client as it had to upload over 3.5GB of data. Torrents A, B and F also offer valuable insights: In those torrents, BitThief was, on average, slightly faster than the official client, which uploaded 232MB in a run of torrent A and 129MB in a run of Torrent B. As far as relatively small torrents are concerned, BitThief seems to have an advantage over the official client, probably due to the aggressive connection opening.

3.2 Leechers

In this section, we further constrain BitThief to only download from other leechers. Interestingly, as we will see, even in such a scenario, free riding is possible.

Seeders are identified by the *bitmask* the client gets when the connection to the remote peer is established, and the *having-message* received every time the remote peer has successfully acquired a new piece. As soon as the remote peer has accumulated all pieces, we immediately close the connection. We conducted the tests at the same time as in

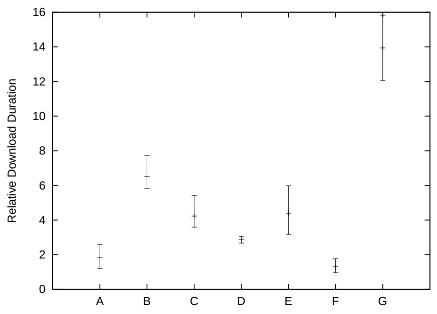


Figure 3: Relative download times of BitThief for six torrents *without downloading from any seeders*. The download time of the official client is normalized to 1.0. As in the first experiment, the torrents were downloaded three times with the official client and three times using BitThief restricted to download from leechers only. The bars again represent the same minimum, average and maximum running times.

Section 3.1 and also used the same torrents. The running times are depicted in Figure 3. It does not come as a surprise that the average download time has increased. Nevertheless, we can again see that all downloads finished eventually. Moreover, note that the test is slightly unfair for BitThief, as the official client was allowed to download not only from the leechers, but also from all seeders! In fact, in some swarms only a relatively small fraction of all peers are leechers. For example in Torrent C, merely 15% are leechers, and BitThief can thus download from less than a sixth of all available peers; nevertheless, BitThief only requires roughly 5 times longer than the official client.

We conclude that even without downloading from seeders, BitThief can download the whole torrent from leechers exclusively. Therefore, it is not only the seeders which provide opportunities to free ride, as well the leechers can be exploited.

3.3 Further Experiments

The measurements presented so far have all been obtained through experiments on the Internet and hence were subject to various external effects. For example, in case BitThief was allowed to download from seeders, it sometimes downloaded at a high rate, but then—a few minutes later—the download rate declined abruptly due to a powerful seeder having left the network. In order to get reproducible results, we set up a pet network environment on a host, consisting of a private tracker, a configurable number of official clients as seeders and leechers and one instance of our own client. We evaluated different scenarios. In the following, our main findings will be summarized briefly.

In scenarios with many seeders and only very few leechers, our client will download most data from seeders. As the leechers often do not fill up all their upload slots with other leechers, we are being unchoked all the time, yielding a constant download rate.

More interesting are scenarios with a small number of seeders. A fast seeder is able to push data into the swarm at a high rate and all the leechers can reciprocate by sharing the data quickly with their upstreams fully saturated. In this situation, it is difficult for our client to achieve a good downstream: We only get a small share of the seeders' upstream and all the other leechers are busy exchanging pieces between them. Hence, we only profit from the optimistic unchoke slots, which results in a poor performance. However, note that many leechers will turn into seeders relatively soon and therefore our download rate will increase steadily.

A slow seeder is not able to push data fast enough into the swarm, and the leechers reciprocate the newly arrived pieces much faster without filling all their upload slots. Although BitThief can not profit from the seeders, it can make use of the leechers' free upload slots. The attainable download rate is similar to the one where there are many seeders. The download rate will go down only when BitThief has collected all pieces available in the swarm. When a new piece arrives, the leechers will quickly exchange it, enabling BitThief to download it as well with almost no delay. An experiment illustrating this behavior is given in Figure 4. Note that the execution shown in the figure is quite idealistic, as there are no other leechers joining the torrent over time.

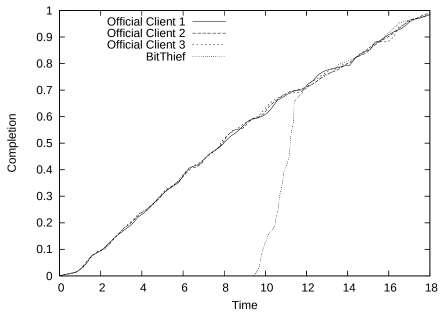


Figure 4: Download times for three official clients and one BitThief client in the presence of a slow seeder. BitThief starts downloading 9 minutes later than the other clients, but catches up quickly. Ultimately, all clients finish the download roughly at the same time.

In summary, the results obtained from experiments on the Internet have been confirmed in the experiments conducted in our pet network.

3.4 Exploiting Sharing Communities

Finding the right torrent metafile is not always an easy task. There exist many sites listing thousands of torrents (e.g., Mininova), but often the torrents' files are not the ones mentioned in the title or are of poor quality. Therefore, a lot of *sharing communities* have emerged around BitTorrent. These communities usually require registration on an invitation basis or with a limit on the number of active users. Finding good quality torrents in these communities is much more convenient than on public torrent repositories. Sharing communities usually encourage their users to upload at least as much data as they download, i.e., to keep their sharing ratio above 1.0. This is achieved by banning users with constantly low sharing ratios or by denying them access to the newest torrents available.

Andrade et al. [2] studied these communities and analyzed how sharing ratio enforcement influences seeding behavior. The authors find that seeders are staying in a torrent for longer periods of time, i.e., typically the majority of peers are seeders. These communities thus exhibit ideal conditions for BitThief, provided that we can find ways to access and stay in this communities without uploading.

We have found that this can often be done by simply pretending to be uploading a lot. The community sites make use of the tracker announcements which every client performs regularly. In these announcements the client reports the current amount of data downloaded and uploaded. These numbers are stored in a database and used later on to calculate the sharing ratios. The tracker typically does not verify these numbers, although it would be possible to determine bad peers in our opinion.[8]

The tracker can also be cheated easily: Clients can announce bogus information and fake peers so that the tracker's peer list fills up with dozens of clients which do not exist. The seeder and leecher counts reported by the tracker can therefore be misleading as there are usually not that many real peers downloading a given torrent. Even worse, peers asking a tracker for other peers can get a lot of invalid or stale information, which makes torrent starts slow.

An alternative is used by recent BitTorrent clients: A distributed tracker protocol which manages the torrent swarm. The technique of faking tracker announcements has been used in a couple of torrents in our tests and we now have a sharing ratio of 1.4 on *TorrentLeech*[9] without ever uploading a single bit.

An example which emphasizes how dramatic the difference between a community internal and an external download can be, is given in Figure 5. We used a torrent that was published on TorrentLeech approximately 12 hours before conducting this experiment and looked for the same one on Mininova, where it had appeared 4 hours earlier. The torrent was 359MB in size on TorrentLeech and slightly smaller (350MB) on Mininova. We first downloaded the torrent three times from Mininova,

[8]For instance, in a torrent with 100 seeders and just one leecher, it looks suspicious if the leecher is constantly announcing large amounts of uploaded data. Alternatively, the sum of all reported download and upload amounts could be analyzed over different torrents and time periods, in order to detect and ban dishonest peers.

[9]See http://torrentleech.org/.

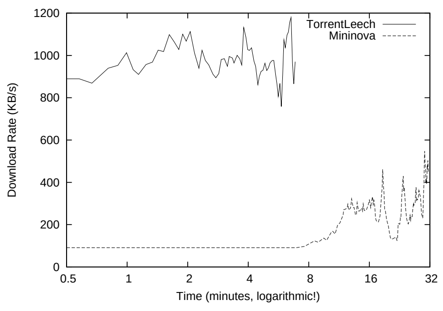


Figure 5: Download speed comparison between a community version of a torrent and one found on Mininova.

then three times from TorrentLeech. The Mininova runs took 32/32/37 minutes, while on TorrentLeech the runs completed in 7:25/7:08/7:08 minutes, respectively. This is more than four times faster. Considering that there were only 25 (24 seeders, one leecher) peers in the TorrentLeech swarm and more than 834 (531 seeders, 303 leechers) peers in the other swarm, this is surprising.

4 Sophisticated Attacks

The previous section has shown that simple tricks can be applied to increase download rates without uploading, and one can think of many more sophisticated exploits to improve the situation further. In this section, some of these potential enhancements are discussed. We will see that BitTorrent is sometimes quite robust to attacks.

First, we have investigated an exploit proposed in [10], which truly violates the BitTorrent protocol: The selfish client announces pieces as being available even if it does not possess them. If such an unavailable piece is requested by a remote peer, the client simply sends random data (garbage). The remote peer has no possibility to verify the subpiece's correctness, as only the integrity of whole pieces can be checked. Although this behavior cannot be considered free riding in the pure sense, it is a strategy that does not require to upload any *valid* user data, thus making it attractive in certain countries where only the *distribution of copyrighted material* is unlawful.

In a first implementation, all requests are answered by uploading entire garbage pieces. As has already been pointed out in [10], this approach is harmful: Both the official client and Azureus store information from whom they have received subpieces and will thus immediately ban our IP address once the hash verification fails.[10] Consequently, we have tried to answer all requests for a piece except for one subpiece, which would force the remote peer to get that subpiece from a different peer. The idea is that the remote peer cannot tell which peer uploaded the fake data, as it might as well be the other peer which only supplied one subpiece. Indeed, the official client can be fooled this way and will just discard the piece and fetch it again, without any consequences for our client. Azureus is smarter and uses an interesting approach: Once it has determined that the piece is not valid, it looks up from which peer it received most subpieces. The piece is then reserved for that peer, and Azureus aims at fetching all remaining subpieces from the same peer. There are now two choices for BitThief: Either we upload the subpieces requested, but this results in the peer having received the whole piece from us and hence our IP address would be banned, or, we refuse to answer the requests. In the latter case, however, the connection will stall because Azureus will not request any other subpieces before it receives the denied ones. Even worse, if we fail to provide Azureus with the missing pieces within a reasonable amount of time, it will ban our IP address as well. We have also tried to not only deny one subpiece per piece, but more than half of them, so that the chances that we become the peer which provided most subpieces gets smaller. Finally, we closed connections to remote peers that had only forbidden subpiece requests in the queue, in order to circumvent the banning algorithm of Azureus. The connection to that peer might be reopened later on, but then again, after uploading a couple of subpieces the connection stalls again, because there are only refused requests pending from the remote peer. In conclusion, our various tests showed that uploading random garbage, in any way, does not improve performance. In every test our client configured for not uploading any data finished the torrents faster than the one that sent garbage.

BitTorrent peers inform each other at the beginning of a connection about their download status by sending a list of pieces that they have already successfully downloaded. While the connection is active, peers send messages to each other for each new piece they downloaded. Peers therefore always know the progress of their neighbors. We sought to measure the influence that this information has on a remote peer. Currently, BitThief sends an empty list of available pieces during connection setup and it does not inform the remote peer about any new pieces it acquires. We tried different settings, such as announcing 0%, 50%, 99% or 100% of all the pieces at the beginning of the connection. The experiments showed that it does not matter what percentage of pieces are announced, as long as BitThief did not announce all pieces to be available. By doing so, a remote peer considers BitThief a seeder and therefore does not respond to any piece requests and will even close the connection once it becomes a seeder itself. Thus, the performance was noticeably worse when announcing 100% of the pieces.

BitThief profits from the optimistic unchoke slots of leechers and from the round robin unchoke scheme of seeders. Thus, a client could possibly increase the chance of being unchoked by being present in the remote peer's neighborhood more than once. This is known as *Sybil attack* [5]. However, this attack involves opening two or more connec-

[10]Note that an appealing solution would be to fake entire pieces by using contents yielding the same hash values. Unfortunately, however, the computation of such SHA-1 hash collisions is expensive and would yield huge tables which cannot be stored in today's databases.

tions to a remote peer. Both the official client and Azureus prevent such a behavior. If multiple IP addresses are available, it would be an easy task to extend the client in a way to fake two entities and trick remote peers. The peers would gladly open a connection to both external addresses and thus our download rate might increase up to twofold.

5 Related Work

In 2000, Adar and Huberman [1] noticed the existence of a large fraction of free riders in the file sharing network *Gnutella*. The problem of selfish behavior in peer-to-peer systems has been a hot topic in p2p research ever since, e.g. [8, 12], and many mechanisms to encourage cooperation have been proposed, for example in [6, 7, 11, 13, 14].

BitTorrent [4] has incorporated a fairness mechanism from the beginning. Although this mechanism has similarities to the well known *tit-for-tat mechanism* [3], the mechanism employed in BitTorrent distinguishes itself from the classic tit-for-tat mechanism in many respects [9]. This fairness mechanism has also been the subject of active research recently. Based on PlanetLab tests, [9] has argued that BitTorrent lacks appropriate rewards and punishments and therefore peers might be tempted to freeload. The authors further propose a tit-for-tat-oriented mechanism based on the iterated prisoner's dilemma [3] in order to deter peers from freeloading. However, in their work, a peer is already considered a free rider if it contributes considerably less than other peers. We, on the other hand, aim at attaining fast downloads strictly without uploading any data. This is often desirable, since in many countries downloading certain media content is legal whereas uploading is not.

The paper closest to our work is by Liogkas et al. [10]. The authors implement three selfish BitTorrent exploits and evaluate their effectiveness. They come to the conclusion that while peers can sometimes benefit slightly from being selfish, BitTorrent is fairly robust. Our work extends [10] in that, rather than concentrating on individual attacks, we have implemented a client that combines several attacks (an open question in [10]). In contrast to our work, the authors examine the effect of free riders on the *overall system* and argue that the quality of service is not severely affected by the presence of some peers that contribute only marginally. We focus strictly on maximizing the download rate of a *single, selfish peer*, regardless of what effect this peer has on the system.

Finally, [2] has studied the cooperation in *BitTorrent communities*. It has been shown that community-specific policies can boost cooperation. In our work, we have demonstrated that cheating is often easy in communities and selfish behavior even more rewarding.

6 Conclusion and Outlook

The advent of free riders which do not upload anything at all, and the lack of punishment, raises concerns about the future of peer-to-peer file sharing systems. In a first thread of future research, we aim at incorporating further selfish attacks such as *collusion* into BitThief. Moreover, current trends such as *ISP caching*[11] could also introduce new potential exploits.

In a second thread of research, we extend our BitThief client such that it truly enforces cooperation among peers. For this purpose, the *Fast Extension*[12] might serve as a promising starting point. A challenging problem which has to be addressed is to find a mechanism that applies some kind of tit-for-tat algorithm for older peers in the system, while at the same time solving the *bootstrap problem* of newly joining peers efficiently: As these new peers inherently do not have any data to share, they must be provided with some "venture capital".

References

[1] E. Adar and B. A. Huberman. Free Riding on Gnutella. *First Monday*, 5(10), 2000.

[2] N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Ripeanu. Influences on Cooperation in BitTorrent Communities. In *Proc. 3rd ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, 2005.

[3] R. Axelrod. The Evolution of Cooperation. *Science*, 211(4489):1390-6, 1981.

[4] B. Cohen. Incentives Build Robustness in BitTorrent. In *Proc. 1st Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, 2003.

[5] J. R. Douceur. The Sybil Attack. In *1st International Workshop on Peer-to-Peer Systems (IPTPS), Cambridge, MA, USA*, pages 251–260, 2002.

[6] M. Feldman and J. Chuang. Overcoming Free-Riding Behavior in Peer-to-Peer Systems. *ACM Sigecom Exchanges*, 6, 2005.

[7] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer. Havelaar: A Robust and Efficient Reputation System for Active Peer-to-Peer Systems. In *1st Workshop on the Economics of Networked Systems (NetEcon), Ann Arbor, Michigan, USA*, June 2006.

[8] D. Hughes, G. Coulson, and J. Walkerdine. Free Riding on Gnutella Revisited: The Bell Tolls? *IEEE Distributed Systems Online*, 6(6), 2005.

[9] S. Jun and M. Ahamad. Incentives in BitTorrent Induce Free Riding. In *Proc. 3rd ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, 2005.

[10] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting BitTorrent For Fun (But Not Profit). In *Proc. 5th Itl. Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[11] S. Sanghavi and B. Hajek. A New Mechanism for the Free-rider Problem. In *Proc. 3rd ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, 2005.

[12] J. Shneidman and D. C. Parkes. Rationality and Self-Interest in Peer to Peer Networks. In *Proc. 2nd Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[13] K. Tamilmani, V. Pai, and A. Mohr. SWIFT: A System with Incentives for Trading. In *Proc. 2nd Workshop on Economics of Peer-to-Peer Systems*, 2004.

[14] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A Secure Economic Framework for P2P Resource Sharing. In *Proc. Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, 2003.

[11] See CacheLogic: Press Release http://www.cachelogic.com /home/pages/news/pr070806.php.

[12] See http://bittorrent.org/fast_extensions.html.

Bibliography

[1] R. Axelrod. The Evolution of Cooperation. *Science*, 211(4489):1390-6, 1981.

[2] B. Cohen. Incentives Build Robustness in BitTorrent. In *Proc. 1st Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, 2003.

[3] J. R. Douceur. The Sybil Attack. In *1st International Workshop on Peer-to-Peer Systems (IPTPS), Cambridge, MA, USA*, pages 251–260, 2002.

[4] C. Gkantsidis and P. R. Rodriguez. Network Coding for Large Scale Content Distribution. *IEEE Infocom 2005*, 2005.

[5] C. Gkantsidis and P. R. Rodriguez. Cooperative Security for Network Coding File Distribution. Technical report, Microsoft Research, 2006.

[6] S. Jun and M. Ahamad. Incentives in BitTorrent Induce Free Riding. In *Proc. 3rd ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, 2005.

[7] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting BitTorrent For Fun (But Not Profit). In *Proc. 5th Itl. Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[8] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. *Under Submission*, 2006.