
Semester Thesis

BitTorrent for Really Selfish Peers – T4T Trading over Cycles

Roger Odermatt

oroger@student.ethz.ch

Professor: Dr. Roger Wattenhofer

Advisor: Raphael Eidenbenz

Abstract

This thesis introduces the design and the implementation of a peer-to-peer data exchange protocol called *Cycle T4T*. The fair sharing mechanism of the widely used *BitTorrent* protocol is commonly believed to discourage freeloading behaviour. However, it does not provide sufficient incentives to rule out free riding, as proven by the BitTorrent client *BitThief*. The main difficulty in designing a cheating proof peer-to-peer protocol is to overcome the problems of finding suitable trading partners and bootstrapping new peers. This thesis proposes a tit for tat trading mechanism to relax these problems and to increase efficiency of a BitTorrent-like system by trading beyond the borders of one file. The BitTorrent protocol is used as a basis to find opportunities to indirectly share among multiple peers, forming a *cycle*.

The new protocol is then implemented into the existing BitTorrent client *BitThief*.

Contents

Abstract	3
1 Introduction	5
2 BitTorrent	6
2.1 Sharing Content	6
2.2 Communication between Peers	7
2.3 Trading.....	7
3 BitThief	8
3.1 Aggressive Connection Opening	8
3.2 Uploading Garbage	8
3.3 BitThief Cooperates	9
4 Cycle T4T	10
4.1 Concept	10
4.1.1 Identifying Peers	10
4.1.2 Finding cycles	11
4.1.3 Trading over Cycles	12
4.2 Implementation Details	13
4.2.1 General Considerations and Data Structures	13
4.2.2 Connection Management	13
4.2.3 Cycle Id.....	14
4.2.4 Message Formats	14
5 Conclusion	16
References	17
Appendix	18

1 Introduction

Peer-to-peer (P2P) systems are decentralized networks, where resources are shared among participating peers. The possibility of load balancing generally offers numerous advantages over centralized systems. P2P systems are inherently scalable and the lack of a single point of failure leads to robustness. The performance of such a system depends heavily on the cooperation of its participants and therefore has to provide strong incentives for cooperative behaviour. The popular P2P protocol *BitTorrent* offers such incentives by employing a simple tit for tat scheme [1]. This mechanism is believed to render freeloading unattractive and discourage selfish peers. However, the observed robustness in BitTorrent results mainly from the widespread use of cooperative client software and the abundance of peers serving every request altruistically. Another main deficiency of the BitTorrent protocol is how it tackles the *bootstrap problem*. As a new peer enters a torrent swarm, it has no data to trade with other peers. Thus it initially needs to get some data without reciprocating at all. BitTorrent addresses the problem by giving free data to random peers periodically. This mechanism however can be exploited, as e.g. Locher et al. show in [2]. As a proof of concept, they implemented a free riding BitTorrent client, *BitThief*.

Since BitTorrent still is one of the most popular file sharing networks, we intend to make it more robust against free riding while keeping the base of the protocol. Tit for tat is a highly effective strategy for repeated interaction with peers we cannot trust [3]. But strict tit for tat entails two problems, the *bootstrap problem* and the problem of *data diversity*. A tit for tat exchange scheme being strict would imply that a newly joining peer will never be able to participate in any sharing activity. The problem of data diversity emerges from the fact that BitTorrent limits the scope of trading to one torrent swarm. This limits the data diversity to the fixed amount of pieces a torrent is split into. As a peer progresses with its download, the chances to encounter a suitable sharing partner rapidly dwindle, since in strict tit for tat both sharing peers must possess a piece needed by the other.

Cycle T4T breaks those boundaries in that it finds cycles among multiple peers over which they can trade indirectly. We want to provide peers joining a new torrent swarm with the chance to trade their valid data from other swarms in exchange for data in the new swarm. By detecting those *cycles* and applying a fair indirect tit for tat sharing mechanism, we intend to overcome the problems of bootstrap and data diversity. Additionally we give peers a strong incentive not to leave a torrent swarm after completely downloading the shared data, as finished downloads now build the basis for future downloading of new torrents.

The thesis is organized as follows. We will give a short overview of the BitTorrent protocol in Section 2 and the BitThief client in Section 3. In Section 4, we present the architecture of the Cycle T4T protocol and provide some implementation details. We conclude the report in Section 5.

2 BitTorrent

The BitTorrent protocol and its reference implementation, the mainline client, were developed in 2001 by Bram Cohen [1]. By having the peers download and upload simultaneously, network traffic can be effectively distributed among all peers sharing content.

The BitTorrent protocol has become increasingly popular over the last years. Recent studies state that approximately a third of all internet traffic originates from BitTorrent [5]. The BitTorrent protocol allows people with limited upload resources to make their content available to a large community, without the problems of the traditional client-server model, i.e. overloaded server or bad service for all clients.

2.1 Sharing Content

The first step when using BitTorrent to distribute content is creating a torrent *metafile*, containing necessary information about the files being shared:

- filenames and -paths
- file lengths
- size of pieces
- hashes of pieces
- *tracker* URL

The *tracker* is a server that stores the peer address and the *info hash*, a hash of the metafile used to associate the peer with the correct torrent. Peers have to request a list of these addresses to contact other peers sharing the same torrent. With BitTorrent one can distribute single files or whole folders with multiple files. The data is split into equally sized *pieces*, usually between 64kB and 4MB long, depending on the total amount of data. The metafile also includes checksums for all pieces making integrity verification of received data possible. This helps to recover from communication errors without losing much data and simplifies detecting malicious or free riding peers sending garbage.

In a next step, the distributing peer starts *seeding* the torrent. A peer is called *seed* or *seeder* if it possesses the entire content. The seeder will now announce itself to the tracker URL specified in the metafile. Metafiles are mainly published on websites, but can be distributed in any way. Other peers can now obtain the metafile, contact the tracker and start downloading from the seeder. While a peer is still downloading the files, it is called a *leecher*. A leecher periodically announces itself to the tracker to refresh its list of peers. Peers sharing the torrent form the so called torrent *swarm*. Swarms are independent structures and a peer can be part of multiple swarms at the same time.

2.2 Communication between Peers

After obtaining the list of IP address from the tracker, a TCP connection is initialized to a remote peer and a handshake message is sent. This message contains:

- Reserved bytes
- Info hash
- Peer ID

The reserved bytes contain information about supported protocol extensions, whereas the peer id consists of random bits and information about the used client software. After performing the handshake and checking if the info hashes match, the download progress in form of a *bitfield* is communicated. A bitfield contains one bit for each piece of the torrent indicating whether the respective piece is available for sharing. Whenever a new piece is obtained by a peer, it informs all its sharing partners by sending a *have* message containing the piece's index in the bitfield. Thus communicating peers can determine at any moment which peers possess the missing pieces. A peer notifies others of its current trading intentions by sending *interested* or *non-interested* messages. If a connection is idle for two minutes, a peer must send a *keep-alive* message to prevent the other peer from terminating the connection.

To signal a remote peer if it is allowed to send *request* messages for data, a *unchoke* message is sent. When a peer stops serving requests to a previously unchoked peer, it sends a *choke* message. Furthermore requests can be aborted by sending a *cancel* message, used mainly during the endgame, where the last missing pieces are requested from multiple peers and only the fastest links will be used.

2.3 Trading

BitTorrent provides incentives for the participants to contribute resources via a tit for tat approach. That means in particular that a peer uploads data only to those peers it receives the best download rates from in return. When a peer decides to unchoke another peer and later notices not to benefit from exchanging data, it stops the transmission and chokes the peer again. This decision is reconsidered every ten seconds. Peers offering more missing pieces or the rarest pieces have the highest priority to be unchoked. To solve the bootstrap problem and to find better sharing partners, peers unchoke a random peer every thirty seconds. This mechanism is called *optimistic unchoking*. After finishing a download and becoming a seeder, a peer stays in the swarm and uses a round-robin algorithm to decide which leecher to unchoke next. Seeding is completely altruistic in public torrent swarms. A rational BitTorrent user will therefore leave the swarm after completing her download.

3 BitThief

BitThief is a BitTorrent client developed at ETH's Distributed Computing Group in 2006 [4]. It demonstrated the possibility to download data without contributing anything. BitThief exploits the weaknesses of the BitTorrent protocol using several mechanisms. What follows is a short overview of those mechanisms. For a more detailed discussion, refer to [6].

An obvious strategy for free riding is exploiting the seeder's altruism. Seeders have already completed their download. They do not expect leechers to reciprocate. After a while, many torrent swarms consist of mostly seeders, as some users often remain in the swarm long after becoming seeders and most interested peers have completed the torrent download. The round-robin algorithm guarantees that every leecher will eventually be served.

3.1 Aggressive Connection Opening

The BitThief client opens as many connections as fast as possible, because more open connections result in a higher probability of being optimistically unchoked by leechers and of being selected by seeders. A tracker sets a default announce interval, commonly a few minutes, to notify the peers of how frequent they are allowed to refresh their peer list. A single announce delivers a list of about fifty peers. Nevertheless, BitThief queries the tracker far more frequently without suffering negative consequences. Trackers usually do not punish such behaviour.

3.2 Uploading Garbage

In general, data requests in BitTorrent do not address entire pieces, as it would take slow peers too much time to upload a piece of several megabytes in size. Pieces are further divided into *blocks*, sub-pieces of 16kB to 32kB. This fact allows BitThief to upload garbage, as long as it avoids uploading a complete piece to a peer, preventing a hash check. Even if the downloading peer gets the remaining parts from other peers, the culprit would not be easily identifiable. A strategy to counter this mechanism is to request a full piece entirely from a single peer. Only serving block requests up to a certain percentage of the piece ends up in becoming choked by the remote peer. Many modern clients have such techniques implemented, rendering this strategy almost useless nowadays.

3.3 BitThief Cooperates

As part of the Distributed Computing Group's long-term goal of designing an efficient and cheating-proof P2P system, the newest version of BitThief features a *source coding* based sharing mechanism called *T4T*.

T4T is based on a tit for tat exchange of data blocks directly between two peers. To circumvent the inherent problems of strict tit for tat, source coding of the original data pieces is employed. Data diversity in the network is greatly increased by only transmitting linear combinations of a torrent's pieces.

The bootstrap problem is circumvented by a simple measure, where a new node is only allowed to download a small, well-defined set of blocks for free. This set is provided by the seeders and depends only on the leechers global IP address. Free riding solely by downloading from seeders would be almost impossible, as it requires a peer to have means of changing his global IP address arbitrarily.

T4T additionally ensures data integrity of received blocks by a mechanism that relies on homomorphic hash functions. This enables peers to compute expected hash values of the linear combinations out of the hash values in the metafile.

A drawback of this approach is the computationally very expensive decoding once enough encoded blocks have been traded. Another disadvantage is the need of a large set of leechers in each torrent swarm to minimize the amount of blocks the seeders have to bootstrap. Thus the T4T protocol works out best for popular files with hundreds of peers, but does not reduce the load on seeders of a small swarm. For further information refer to [7].

4 Cycle T4T

4.1 Concept

The idea of Cycle T4T is to trade over cycles of multiple peers using a tit for tat exchange mechanism. Given a peer *A* seeds a torrent *file1* and a peer *B* seeds a torrent *file2*, whereas peer *A* joins the torrent swarm of *file2* and peer *B* joins the torrent swarm of *file1* as leechers. The BitTorrent protocol can not make use of this situation, as there is no interaction intended between the peers in separate swarms. Thus the two peers *A* and *B* take the initiative and engage in a direct strict tit for tat exchange scheme or in other words, start trading over a cycle of length 2. In order to take advantage of this exchange opportunity involving two separate torrents, peer *A* has to be able to identify peer *B* in both swarms as one and the same peer and vice versa.

4.1.1 Identifying Peers

To identify a single peer in different swarms, BitTorrent offers several approaches. As we use the internet as our communication medium, we could try to use the IP address as identifier. As we consider widely deployed network technologies like network address port translation (NAPT) and BitTorrent clients using different or random ports for each individual torrent, the IP address may not be the best choice in our case. One way to solve the problem could be the exchange of info hashes of all our active torrents with other peers after the handshake, but this raises concerns regarding privacy. As we are not willing to give up more private information on our download behaviour than necessary, we have to concentrate on what the BitTorrent protocol already offers. There is the peer id in the initial handshake message, a string of 20 bytes length. According to Bram Cohen's specification [1], a new peer id is randomly generated by a downloader for every torrent before announcing to the tracker. Originally the tracker reported the peer ids along with the IP addresses of other peers. A peer was expected to terminate a connection if the id from the tracker list and the handshake message didn't match. Eventually a proposal for a compact peer list was accepted by the official BitTorrent developers and is now the prevalent format. The compact peer list requires only 6 bytes per peer for IPv4 address and port, getting rid of the 20 additional bytes for the peer id, effectively reducing response size and computational requirements in trackers without losing any useful functionality. In addition, BitTorrent client software developers use the peer id as a way not only to identify peers but also the client implementation and version in use. A list of known encoding styles and clients utilizing those can be found here [1]. The main portion of the 20 bytes is still random.

Thus we slightly deviate from those specifications by not generating a new peer id for every single download and reuse the same peer id while handshaking in different swarms. The question is: How do we detect cycles consisting of more than two peers and extending over several swarms?

4.1.2 Finding cycles

To achieve this, peers have to cooperate with each other in finding those cycles. First we specify some terms.

After handshaking and exchanging bitfield messages, a peer can tell whether the remote is *interesting*, *interested*, *mutually interested* and *mutually not interested*. The latter is only the case if two peers have identical bitfields. Mutually interested means that the peers do not have to immediately proceed with the cycle search and can start trading. Eventually their relation changes to another type and the following applies. If the remote peer is interesting or interested, we set up two lists, an *inlist* for the remote peers that are one-sidedly interested and an *outlist* for the peers one-sidedly interesting. In addition, a *table* to store the information we receive from cooperating peers. The “interested-relation” between the peers form a directed graph which we call the *demand link graph*, where the edges point in the direction of the actual data flow, once a cycle would have been found.

A peer shares its inlist with the peers of its outlist. The thus received information is stored in the table and again forwarded to the peers of the outlist. As peer *A* interested in peer *B* has to be indirectly interesting to *B* in order to trade, peer *A* only needs a peer in his inlist which is directly or indirectly interesting to *B*. When trading over a cycle, a peer is sharing directly only with his two neighbours on the demand link graph. Thus peer *A* does not require the exact topology of the demand link, however the fact that any of his inlist neighbours is interesting to peer *B* suffices to find the cycle. The required information is limited to a peer id and a hop counter, describing the distance to the respective peer on the demand link graph. The hop counter serves as a way to distinguish between multiple paths connecting peers on the demand link graph. If for example peer *A* finds a second path to peer *B*, it does not have to inform its outlist neighbours in case of a longer second path. The distance of the peers in the initial inlist is thus one. The counter has to be increased whenever the peer id is forwarded accordingly. Because a peer only obtains this information from its direct neighbours in the demand link graph, it never learns the exact topology of the graph, but it only knows the distance to another peer id and the next hop to reach that peer.

When a peer is equipped with such a table, it has now the means to discover cycles by doing a table lookup, once he handshakes with a peer considered interesting and finds the new peer’s id in the table. In this case that peer would be the first to realize the presence of a cycle, and of course has still to forward its table, or the other involved peers won’t find the cycle by themselves. Obviously, one way to detect cycles is to check the received table updates against the outlist. Thus, all peers – except for the peer “closing” the cycle – will learn about the cycle.

Before it is time to start trading, a peer would like to confirm the integrity of the found cycle. Because peer relations may fluctuate over time, table entries might be outdated or perhaps some peer ids were faked to lead us to falsely expect a cycle. An easy way to make sure of a cycle's integrity is to send a message to one sharing partner on the cycle and receiving that message from the other direct exchange partner. This message should be short, as we try to avoid spending much upload capacity, and not predictable, as to defend ourselves from possibly selfish peers with the intention to free ride. We make use of this message to also assign a unique *cycle id* to a cycle.

4.1.3 Trading over Cycles

After a successful integrity check we are finally capable of sharing on one cycle. This is a great achievement, but what if there happens to be more than one cycle concurrently involving several peers? We need to provide each peer participating in exchanges over multiple cycles with the means to distinguish between each cycle, i.e. attaching a unique identifier to every single cycle. Consider the example illustrated in Figure 3.1. The peers *A*, *B* and *E* are sharing on two cycles of length 4. Peer *E* for example sends data to both peer *C* and *D*, but is receiving data only from peer *A*. Peers *C* and *D* may upload at a different rate that could also change dynamically. Peer *B* is the only peer that knows the exact values, but the information is then lost, as *B* uploads at a rate equal to the combined rates to peer *A*, in turn uploading at this rate to peer *E* again. The proposed unique cycle id could facilitate the decision process of peer *E* how to adjust its upload to peer *C* and *D*. As we previously sent some messages along the cycle, we may just as well use that occasion to compute the *cycle id*. In section 3.2 we provide the details of our implementation.

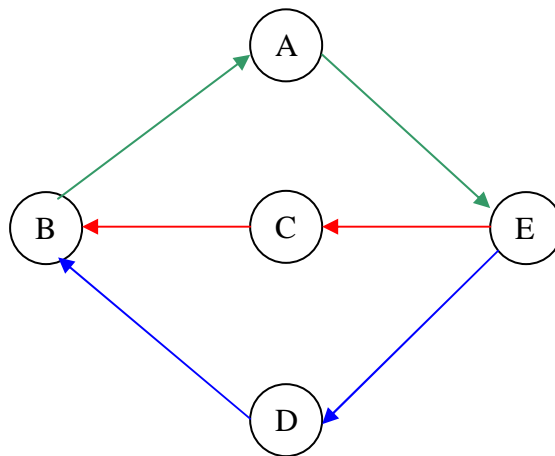


Figure 3.1: Five peers trading over two cycles of length 4

4.2 Implementation Details

4.2.1 General Considerations and Data Structures

To determine the requirements our implementation has to meet and where we have to set limits, we assume a very simple topology: Each peer seeds one file and leeches one file. From the seeders point of view there are ten leechers in the swarm. Those leechers in turn seed their own file where they see ten leechers interested in the file and so on. For every additional hop, the amount of inlists a peer has to store in its table increases at most by a factor of ten. Therefore we have to limit the search range for our protocol, as after 6 hops the table could already hit the million entries mark. While each entry consists of two peer ids and a hop count – that's 44 bytes if we use an integer for the hop – our table is already blown up to nearly 50MB. As we also need to forward this data to all the peers on our outlist, we fixate 6 hops as our maximum search depth. We therefore don't have to forward entries with hop count greater than 5 and can conserve already scarcely available memory. As the real world topology may differ from our model, we also limit the maximal amount of table entries to one million. Of course those are worst case assumptions where we hardly find any cycles.

The inlist and outlist are expected to contain only the few peer ids of the direct neighbours in the demand link graph, so the utilised data structure matters little. We use hash maps. For the table we decided to use nested hash maps for enhanced lookup speed of entries, but for an increased memory cost caused by the mappings.

4.2.2 Connection Management

After a handshake and potential exchange of table entries, the TCP connections are possibly idle for a very long time, until a cycle has been found. As a consequence we close TCP connections until an exchange over a cycle can take place. As we still need to keep the peers on the outlist up-to-date, but don't want to permanently re-establish TCP connections for table updates, we use UDP for updating purposes. In order to receive those updates, a peer is expected to listen on a UDP port, namely the same port as used for TCP announced to the tracker. An update is expected at least every three minutes, otherwise a peer is considered to no longer be interested and the table entries involving that peer will be deleted. If no new update entries are available but a peer is still interested, an empty table update packet can be sent. A peer has to attach his peer id to a UDP update packet to facilitate the correct insertion into the table.

4.2.3 Cycle Id

To ensure that our integrity check message or *cycle id message* cannot be forged, we use cryptography. A peer can apply an arbitrary encryption technique to compute a 20 byte long personal token he will be looking for whenever a cycle id messages is received. Additionally when confirming a new cycle, a peer encrypts the participating peer of the inlist and again builds a 20 byte hash, associating the hash with the peer. A peer now sends his personal token and the hash to the participating peer on the outlist. When receiving a cycle id message without his personal token, but from a peer he is expecting to receive his own cycle id message from, the peer just adds his token and the hash to the message and forwards it to the corresponding peer on the outlist. In case a peer identifies his own personal token, he checks if the hash after the token is indeed the one he encoded. If that applies, it means the cycle id message has really traversed the whole cycle and the personal tokens and hashes of all involved peers are present in the message. As the other peers messages contain the same tokens and hashes but in a different order, we build a 20 byte wise XOR of the message, resulting in the same 20 byte *cycle id* for every peer. We also use XOR as to receive a diverse cycle id that is with high probability unique for the fix set of participating peers. For robustness we send the cycle id message on a second round, as some cycle id messages might arrive before a table update and therefore cannot be correctly forwarded. After two rounds every peer is guaranteed to have found the cycle id.

Apart from forged messages a peer is vulnerable to collusion. Two colluding peers would only have to trace their victim in two fitting torrent swarms in order to deceive it and getting some data for free. To mitigate this threat, we limit the amount of data to be traded before receiving the expected contribution to a few blocks.

4.2.4 Message Formats

Note that to identify other clients supporting Cycle T4T, the reserved bit 33 is used in the initial handshake message.

Table updates have to initially be requested by sending the keep alive message on the TCP connection after handshake and bitfield exchange.

Format of the Piece Message:

Offset	0	4	5	25	29	33
Length	4	1	20	4	4	N
Content	N + 33	101	0x00..00	0x00..00	0x00..00	0x00..00
Description	Length	Type	Cycle Id	Piece Index	Offset	Data

Format of the Cycle Id Message:

Offset	0	4	5	9
Length	4	1	4	$40 \cdot n$
Content	$40 \cdot n + 9$	102	n	0x00..00
Description	Length	Type	Number of Hashes	Hashes

Format of the Table Update Message:

Offset	0	4	5	9
Length	4	1	4	$25 \cdot n$
Content	$25 \cdot n + 9$	103	n	0x00..00
Description	Length	Type	Number of Entries	Update Entries

An Update Entry consists of 1 byte type, 20 byte peer id followed by the 4 byte integer hop count. As obsolete table entries may have to be removed from the table and to inform other peers about those now invalid entries, the type byte has been introduced, where “0” means an add update and “1” means a remove update.

5 Conclusion

We presented Cycle T4T, a P2P protocol based on the popular BitTorrent file sharing network. It employs an indirect tit for tat exchange mechanism to trade beyond the borders of a single swarm. By giving users the possibility to capitalize on finished downloads and thus creating an incentive for peers to stay in a torrent swarm. We tackled the two inherent problems of strict tit for tat, increase data diversity and relax the bootstrap problem.

The introduction of the cycle id, the minimum of shared information and the restricted local view obtained by a cooperative peers should lead to a good acceptability and should render most free riding attempts unattractive, as the effort to locate a victim and the required resources to exploit it is not worthwhile.

The important open yet to be answered is how common trading cycles of the limited length up to six peers are in reality. Deploying and publishing a new version of the BitThief client with Cycle T4T could help us a great deal in gaining a valuable insight into the nature of demand graph topologies of BitTorrent networks.

From a personal view, I was able to get to know what Java has to offer in various fields ranging from network communication over GUI to debugging methods. Thanks to this semester thesis I'm no longer terrified of large software projects and even started to like digging into tons of source code.

References

- [1] Bram Cohen - The BitTorrent Protocol Specification
http://www.bittorrent.org/beps/bep_0003.html
- [2] Free Riding in BitTorrent is Cheap
Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer.
5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA,
November 2006.
<http://disco.ethz.ch/publications/hotnets06.pdf>
- [3] R. Axelrod. The Evolution of Cooperation. Science, 211(4489):1390-6, 1981.
- [4] BitThief website
<http://disco.ethz.ch/projects/bitthief/download.php>
- [5] BitTorrent Still King of P2P Traffic, Feb 09
<http://torrentfreak.com/bittorrent-still-king-of-p2p-traffic-090218/>
- [6] Free Riding in BitTorrent and Countermeasures,
Master's Thesis by Patrick Moor, Summer 06
http://disco.ethz.ch/theses/ss06/bitthief_report.pdf
- [7] Honor Among Thieves - A Source Coding Based Sharing Mechanism for the
BitThief Client, Master's Thesis by Dorian Kind, Summer 08
http://disco.ethz.ch/theses/fs08/bitthief_coding_report.pdf

Appendix

This CD contains a complete list of changes made to the BitThief source code during the course of this semester thesis, as well as an executable JAR file of BitThief.