Master's Thesis

Honor Among Thieves — A Source Coding Based Sharing Mechanism for the BitThief Client

Dorian Kind
dorian@student.ethz.ch

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Supervisor: Thomas Locher

Department of Information Technology and Electrical Engineering
Swiss Federal Institute of Technology (ETH) Zürich
Summer 2008

Abstract

In this thesis, the implementation of a peer-to-peer data distribution system called *T4T* is discussed. *T4T* is based on a truly fair tit-for-tat exchange of data blocks between peers. To circumvent the inherent problems of strict tit-for-tat, source coding of the original data is employed. As only linear combinations of blocks are ever transmitted, the block diversity in the network is greatly increased. In contrast to other network coding systems, we use modular addition in a finite field to encode the original blocks instead of the more typical XOR operation.

Additionally, a mechanism to ensure data integrity of received blocks is proposed and analyzed. We discuss the homomorphic properties of different cryptography systems and outline the design of a hashing scheme that utilizes a homomorphic one-way function based on the discrete logarithm problem. Our scheme enable peers to compute expected hash values of received blocks out of the hashes of the original data. The existing free-riding BitTorrent client *BitThief* is extended to incorporate the discussed new functionality.

Contents

1 Introduction

The problem is to find a form of association which will defend and protect with the whole common force the person and goods of each associate, and in which each, while uniting himself with all, may still obey himself alone, and remain as free as before.

— Jean-Jacques Rousseau, *Du contrat social*

Peer-to-peer file-sharing systems combine sophisticated searching techniques with decentralized file storage to allow users to download files directly from one another. The first mainstream peer-to-peer network, Napster, attracted public attention to the peer-to-peer paradigm as well as tens of millions of users for itself. Napster specialized in helping its users to trade music, as did most of its immediate competitors and successors at the time; today peer-to-peer networks are used to distribute all kinds of digital content and are slowly making progress to supersede the classical server-client model in many areas.

As the power of a peer-to-peer network is based on the resource contribution of its constituent parts, the success of such systems hinges on its ability to cope with selfish peers which aim at benefitting from the network without contributing. One would thus expect that the design of peer-to-peer networks pays special attention to providing incentives for peers to behave in a way that is beneficial for the whole network population. In reality we find that many past and present successful peer-to-peer networks either do not tackle this issue at all or just assume obedient users that strictly adhere to the specified protocol without considering their own utility. This is in stark contrast to the theory of rational choice according to which agents in social and economic environments interact with varying degrees of collaboration and in general try to maximize their benefit while minimizing their costs. A rational user will gladly deviate from a protocol specification in order to increase his utility if there are no consequences to fear. Unfortunately, in doing so, the rational user degrades the utility of the network for all other users.

The fact that many available clients for different peer-to-peer networks usually follow their respective protocol—also, the fact that most users of peer-to-peer software lack the technical expertise to modify the behavior of their client—should not belie the possibilities that are offered to a truly rational peer. This is demonstrated by *BitThief*, a client for BitTorrent[1] networks that is able to download files from a BitTorrent swarm without uploading (contributing) a single bit. What makes this feat more impressive is that BitTorrent is still regarded as a peer-to-peer environment that is quite resistant against malicious peers[2], due to its tit-for-tat inspired data exchange.

The present work aims to show that true fairness in a peer-to-peer file sharing network can be achieved without a central authority or communication overhead. We present an extension to *BitThief* called *T4T* which is based on a quid pro quo exchange of data blocks, resulting in (nearly) equal participation of every single peer. *BitThief* clients, while still only downloading from traditional

[1]See http://www.bittorrent.com/.

[2]The "malicious" peers only act rationally, of course.

BitTorrent peers, can use this protocol extension to fairly trade data among themselves—hence the title of this report.

The cornerstone of *T4T* comes from the field of network coding. In our system, data is only exchanged in the form of linear combinations of original blocks, which provides the increased block diversity within the network that is crucial for maintaining true tit-for-tat trades.

As certain peers may have an interest in disrupting the system by injecting bogus data into the network, we additionally propose a scheme for ensuring the integrity of the blocks a peer receives that is based on hash functions with homomorphic properties.

The rest of this thesis is organized as follows. We will give a short overview of the *BitThief* client, which this work is based upon, in Section 2. In Section 3, we present the *T4T* system after reviewing some related work. Section 4 is concerned with the question of how to provide data integrity in the system. We outline some possible directions for future work in Section 5 and conclude the report in Section 6.

Appendices A.1 and A.2 describe select algorithms, while Appendix B contains several design notes and the specification of the communication protocol.

Acknowledgments

First and foremost I would like to express my thanks to my advisor Thomas Locher for the opportunity to work on *BitThief*, for his support, patience and insightful comments. My thanks extend to Prof. Dr. Roger Wattenhofer and all the members of the Distributed Computing Group, especially Olga Goussevskaia, Michael Kuhn, Yvonne Anne Oswald, and Stefan Schmid, who were excellent advisors while working on previous term papers. Patrick Moor deserves praise for building the original *BitThief* client as well as laying the theoretical foundation upon which this work stands.

2 The BitThief Client

BitThief is a BitTorrent client that was developed at the Distributed Computing Group as part of Patrick Moor's Master's thesis [29] in 2006. Its purpose was to show that it is possible to free-ride in a BitTorrent swarm, i.e., download data without contributing by uploading.

To provide a bit of background, BitTorrent still is the most popular peer-to-peer network. Its enormous user base is illustrated by multiple studies; a recent one analyzing Internet traffic in different regions of the world concluded that—with the notable exception of the middle east—peer-to-peer networks produce more than half of all Internet traffic [34]. Of that traffic volume, between 70% and 97% were stemming from either BitTorrent or eDonkey, with BitTorrent usually taking up around 60% to 70% of all peer-to-peer traffic.

In addition to its immense popularity, BitTorrent is widely considered to be reasonably fair and to offer enough incentives to uploaders to stave off free-riders. In comparison to other peer-to-peer networks, it has the conceptual advantage that a *swarm* (a separate BitTorrent network containing anything between a few up to several thousands of nodes) always only engages in exchange of one single file.[3] This means that, in terms of fairness, peers do not need to take into account what other peers may have been contributing while exchanging other files, but can solely observe another node's momentary behavior in the swarm.

In truly global peer-to-peer networks, peers are (ideally) forming one contiguous mesh, sharing an abundance of different files. Measuring and rewarding participation in such an uncontrolled environment is much more difficult. In many networks, there have been efforts to introduce some sort of reputation that a node can earn by contributing, in order to discourage free-riding. Reputation in this sense can be regarded as a global history of a user's past behavior in the network, and a good reputation is usually rewarded with privileged access to the network's resources, such as faster downloads or shorter queues. The actual implementations of these reputation systems, however, often leave much to be desired. Fasttrack/KaZaA, for example, used to store a user's contribution (which determines his download speeds from other nodes) locally, a scheme that had been quickly cracked and thus proved to be next to useless, because soon every node would claim to have maximum participation. Other mechanisms are more elaborate, but usually require a certain computation and communication overhead. BitTorrent's design is simple yet effective and relies on the fact that all nodes in the network are always competing for the same file.

Without going too much into detail (see [8] for a thorough explanation by the original designer of BitTorrent, Bram Cohen), BitTorrent's attempt at encouraging contribution is its *choke/unchoke* mechanism. A client will only upload data to a maximum number k of other nodes, the k so-called "unchoked" nodes. The choked nodes, on the other hand, retain their connection but do not send requests for data lest they be disconnected. Every once in a while, the client resets the list of choked and unchoked neighbors, sorts the connected peers by their recent upload rate starting with the fastest uploaders, and unchokes the

[3]Technically speaking, BitTorrent has the ability to serve multiple files in a single swarm or "torrent"; however, as an overwhelming majority of users will still get all of the data served by the torrent, one can make this simplification.

first k peers of the list.

While uploading therefore increases a client's chance of becoming unchoked, it is by no means guaranteed that a free-riding client will always stay choked. In addition to the possibility that a non-contributing peer is still in the first k positions of the peer list, there is also a mechanism called "optimistic unchoking" that can be benefitted from. Every node in a swarm has a special upload slot for optimistically unchoking a random peer that used to be choked. This is done in order to detect currently unused connections that might be better (faster) than the used ones. A new, random peer is selected for optimistic unchoking every third unchoking round and offers a malicious peer a chance to download—at least temporarily—without participating.

Another attack vector is the piece selection strategy that BitTorrent clients use. Peers mostly utilize a rarest-first policy, striving to download first those pieces that are least available in their network neighborhood. Additionally, as soon as a sub-piece of a given piece is received, only sub-pieces from that piece are requested until the piece is completed. Those mechanisms try to ensure that pieces remain evenly distributed among all peers, but they can lead to the situation where a peer cannot use his entire potential download capacity because of another peer with available upload capacity not offering the right pieces. A selfish node, however, can just download any piece from the fastest uploader available.

BitThief aims to exploit these weaknesses. It combines multiple attacks, some of the previously described, others inspired by exploits covered in [24]. For a condensed overview and evaluation results, see [25]. It suffices to note here that the BitThief client, in mature swarms with several seeding peers, often reached similar download speeds as other clients, all without uploading a single bit of user data. BitThief was written from scratch in Java, using the implementations of the "mainline" BitTorrent client and the popular open-source client Azureus as references. The latest version is available at http://dcg.ethz.ch/projects/bitthief/.

3 The T4T System

In this section, we will give an overview of the mechanism utilized to introduce true tit-for-tat (T4T) block exchange amongst BitThief clients. First, we will have a quick look at related work in the area, namely techniques whose aim is to ensure fairness in peer-to-peer file exchange networks. Next we briefly describe the motivation for the scheme and cover the actual *T4T* system in detail. Some technical design characteristics and the specification of the communication protocol can be found in Appendix B.

3.1 Related Work

The existence of free-riders in peer-to-peer networks has been observed in multiple studies. An early analysis [1] of the *Gnutella* peer-to-peer network showed that nearly 70% of nodes never upload any data at all. [18] performed a follow-up study of the same network and found that free-riding increased significantly, with now more than 85% of the users not sharing a single file. Another paper by Saroiu et. al. [33] found that in the (now defunct) Napster network, 40% to 60% of the user base provide as little as 5% of all files. It has become clear that the classical "Tragedy of the Commons" problem extends to a digital environment, insomuch as the large percentage of free-riders degrades the network's utility for every user. While the presence of free-riders alone is apparently not driving users away from peer-to-peer networks—a fact that is corroborated by studies with human participants in real life [27]—the full potential of the systems is clearly not tapped as long as free-riding is prevalent.

A lot of work has been done in the past years that promise to provide incentives for participation in peer-to-peer networks. [10] gives a concise general overview of findings and open questions related to free-riding and participation incentives in peer-to-peer systems. Feldman et. al. found in [11]—using a General Prisoner's Dilemma model—that scalable incentive techniques have to rely on shared history, meaning network participants sharing their historical observations of other nodes' actions. There are a multitude of different approaches to do this, of which we briefly list a few:

DHT-based reputation systems store the contribution of a node as observed by other nodes in a network-wide DHT. This—together with a suitable mechanism that privileges peers with good reputation—is a simple solution to discourage free-riding. Unfortunately, DHT lookups needed for updating and retrieving reputation values tend to be expensive in terms of exchanged messages [31]. The scheme is also prone to large-scale *false reports* attacks.

Virtual currency can be utilized in peer-to-peer systems. The "coins" of the virtual currency are used to pay for the download of data, so that nodes that participate more are also going to be more wealthy than other nodes. This usually involves a trusted party (the "central bank", if you will) that manages security and detects fraud attempts like double spending. Disadvantages of the scheme include the fact that virtual coins grow in size each time they are spent [6] and the problem of controlling inflation and deflation. Garcia et al. propose a completely decentralized virtual currency called “Off-line Karma” in [13].

Trust-based networks depend on indirect, second-hand observations to estimate the trustworthiness and reputation of a node. The inherent problem of false reports is overcome by building some sort of trust network. *Eigen-Trust* is an example of such a system that collects reputation values about a given node from a subset of other nodes from the network, weighted by their own respective reputation [19].

3.2 Motivation

As of now, BitTorrent swarms depend on altruistic users that continue to share even when their sharing ratio exceeds one, in other words, peers that upload more data into the swarm than they have received.[4] These benevolent peers allow other, more selfish nodes to get away with greatly reduced sharing ratios; in some cases—as demonstrated by BitThief—not uploading at all. As every peer-to-peer network crucially dependens on collaboration, it is desirable to force these selfish peers to upload as well.

If we regard a BitTorrent swarm from a game-theoretic point of view, we can view the exchange of blocks as rounds of a game of repeated interaction where players can either cooperate (uploading a block to the other player) or defect (not uploading any data). Since Axelrod and Hamilton’s famous work “The Evolution of Cooperation” [3] first examined these kinds of games, *tit-for-tat* is considered as the asymptotically best strategy, given that every participant tries to maximize its own utility. Tit-for-tat’s basic “nice” (it will strictly cooperate as long as not provoked) and “provokable” (once provoked, it will retaliate) properties seem to make it impervious to attacks. Thus, it is natural to try and introduce a fairness model that is based on tit-for-tat.

Note that this is not a contradiction to the previously mentioned findings in [11], as we do not care about another node’s *global* participation. Instead, we only regard the isolated view of the two nodes that engage in data exchange. As all nodes in a BitTorrent swarm are interested in the same single file, fairness in every single exchange implies a fair system overall. This means, however, that nodes cannot accumulate the value gained from participation over time as it is possible in other systems.

Pure tit-for-tat in the BitTorrent environment means that for every block downloaded from a given peer b, a peer a also has to upload a block to b, or b will no longer interact with a. This immediately poses two problems:

The bootstrap problem is immanent in a true tit-for-tat exchange scheme: a newly joining node is not able to participate in any transaction as it has no “seed capital”. This problem is not as severe as it can be circumvented by a simple measure that is similar to BitTorrent’s *Fast Extension* mechanism,

[4]There exist closed BitTorrent communities where members have to always keep their share ratio above 1 or risk being banned. However, even in these swarms it is imaginable that malicious nodes just upload garbage sub-pieces, exploiting the fact that only whole pieces can be verified.

by which a new node is allowed to download a small, well-defined set of blocks for free.

The problem of block diversity is graver: As the number of blocks in a BitTorrent download is fixed (usually in the low thousands) and every single block is required to complete the download, peers quickly run into the situation where node a is interested in some blocks of node b, but not vice versa (because b already possesses all the blocks that a can offer); thus preventing interaction between a and b. As an example, assume that a, b each possess $n/2$ blocks of a file consisting of n blocks. If we assume that the downloaded subset of blocks is random, then a and b are each interested in about half of the blocks the other node has. They will thus play tit-for-tat until both possess $3n/4$ blocks of the file, at which point they have to find other sources to download from, potentially wasting the available bandwidth between a and b. The solution to this problem is the core of the *T4T* system and is described next.

3.3 Source Coding and Block Diversity

Source coding, respectively the more general network coding is a fairly new topic in information theory. Ahlswede et. al. introduced the concept in their article "Network Information Flow" [2] in 2000 by showing that routing and duplicating messages alone is generally not able to achieve maximum throughput[5] in a network graph. Instead, one has to introduce (re-)encoding of symbols of a message at the source and at intermediate nodes (the original paper uses a class of block codes called α-codes). Sinks can then decode the original message as soon as enough encoded symbols have arrived.

While network coding is able to utilize a network's capacity more efficiently, it also has a few drawbacks, namely the increased complexity of coding as opposed to traditional routing and the requirement that the topology of the network be known at the time of constructing the coding algorithm. And while network coding is thought to be of much use in large-scale distributed systems, such as multicast or wireless networks (see [15] and [20] for examples), the assumption of a static network graph questions the usability of network coding in dynamic peer-to-peer networks [35], where nodes may join and exit the swarm at any time. Also, the fact that an optimal network coding algorithm inevitably requires the peers with greater uplink capacity to serve more traffic may discourage selfish peers.

As our primary goal is not necessarily optimizing the information flow inside the network, but to enhance block diversity, we focus our attention to a secondary effect of network coding: The fact that messages are re-encoded such that a certain number of them can be decoded at the source with no communication overhead provides a readily available solution to our block diversity problem. We can even do away with the re-encoding of messages at intermediate nodes, and just take the idea that with transmitting linear combinations of blocks instead of pieces of the original file, we get our desired high block diversity. This approach is a somewhat lesser variant of network coding, and

[5]Maximum throughput meaning the maximum flow in the flow network.

has already been used to design peer-to-peer content distribution systems, for example in [14] and [22].

Similar to the original BitTorrent file, we consider a file f to be composed of n blocks. A block is the basic exchange unit between peers and consists of m symbols. While in other network coding systems these symbols are often elements of a Galois Field $GF(2^p)$ with the XOR operation defined as addition, we chose as symbol domain the finite field $GF(q)$ with q being a large prime, namely the *Mersenne prime* $2^{31} - 1$, together with regular, modular addition. Binary fields have the advantage that they fit nicely into bit strings of length p and their arithmetics can be performed atomically in hardware. But as a Mersenne prime is *almost* a power of two, groups operations can be implemented quite efficiently. When the result of an addition exceeds q, one can just drop the carry bit and increase the result by one. To illustrate the principle, a file f to be downloaded is represented as a $m \times n$ matrix $\mathbf{F}$

$$\mathbf{F} = \begin{pmatrix} b_1[1] & \dots & b_m[1] \\ \vdots & \ddots & \vdots \\ b_1[n] & \dots & b_m[n] \end{pmatrix} ,$$

where $\mathbf{b}[i]$ is the i^{th} block of f represented by a vector of m symbols $\big[b_1[i] \dots b_m[i]\big]$ with $b_j[i] \in GF(q)$ for $0 < i \leq n$, $0 < j \leq m$.

All arithmetic operations performed on block vectors are done component-wise where applicable, i.e. for an operation $\star$ and two blocks $\mathbf{a} = [a_1 \dots a_m], \mathbf{b} = [b_1 \dots b_m]$ we have that $\mathbf{a} \star \mathbf{b} = \mathbf{c}$ where $\mathbf{c} = [a_1 \star b_1 \dots a_m \star b_m]$. Scalar multiplication is performed as usual.

Seeders, that is, peers in possession of the whole file, build linear combinations out of the original blocks by adding k random blocks together. The resulting combination block $\mathbf{c}$ is associated with its vector of coefficients that serves as a kind of block ID. For every combination block $\mathbf{c}$ and corresponding ID vector $\mathbf{v} = [v_1 \dots v_n] \in \{0, 1\}^n$, we therefore have

$$\mathbf{c} = \sum_{i=1}^{n} v_i \mathbf{b}[i] ,$$

where again $\mathbf{b}[i]$ is the i^{th} block of the original file.

Only combination blocks are ever exchanged between nodes, and only seeders create new combinations. This means that a bitfield will always suffice to identify a given linear combinations, as coefficients will always be either 0 or 1.

How exactly is this scheme helping the block diversity problem? If we again take the example of two nodes that are each in possession of (random) $n/2$ blocks, both peers have now a much improved ratio of blocks that they are interested in. They each can use about $\frac{n}{2}(1 - n/(2\binom{n}{k}))$ of the other node's blocks, which means that even for small k, both will be able to almost finish their downloads without interacting with any other peers.

3.4 Decoding

Ideally, a peer is able to decode its received blocks and retrieve the original file as soon as it possesses n linear combination vectors. Decoding in our system is equivalent to solving a system of linear equations. The $n \times n$ coefficient matrix $\mathbf{V}$ is constructed with the ID vectors of the received combinations as row vectors, resulting in a sparse matrix with exactly k ones and $(n-k)$ zeroes in every row. We also have a right-hand side vector $\mathbf{r}$ whose entries are the downloaded linear combination blocks. Thus we solve $\mathbf{Vx} = \mathbf{r}$ for $\mathbf{x}$ and will have the original file's n blocks as entries of $\mathbf{x}$.

It is clear that the system can only be uniquely solved if $\mathbf{V}$ is invertible, i.e. if $\mathbf{V}$ has full rank n. This basic fact is the motivation for our choice of modular arithmetic instead of the XOR operation for the source coding. Simulation reveals that when using XOR in a binary field, the coefficient matrix that results from randomly producing linear combinations of blocks has nearly always rank less than n. In network coding systems utilizing binary fields, random weights for individual blocks are usually employed to ensure the non-singularity of $\mathbf{V}$. We do not use this weighting mechanism, simplifying the coding processes and replacing the transmission of a vector of weighting coefficients with a simple bitmask.

After inverting $\mathbf{V}$, all n downloaded linear combinations are multiplied with $\mathbf{V}^{-1}$ to reconstruct the original blocks. Inverting a square matrix of size n can be done in $O(n^3)$ time, and the multiplication with the received blocks takes another $O(n^2m)$. With m normally being much larger than n, the decoding time is dominated by the time required to multiply the inverse with the blocks. However, as the product nm is constant for a given file size, reducing m will actually slow down the decoding. We found that even for moderately large n, the time to invert $\mathbf{V}$ and decode the blocks must be measured in minutes, not seconds. Relief comes from the fact that the second part of the decoding process is easily parallelized, so that we can take advantage of the contemporary shift towards multi-core home computers.

The choice of k is crucial for the algorithm. While even with $k = 2$ the block diversity is already great enough to ensure smooth tit-for-tat exchange, k has to be larger to guarantee the invertibility of the coefficient matrix $\mathbf{V}$. On the other hand, a large k makes the encoding slower as more blocks have to be added together for each combination, and results in a denser matrix $\mathbf{V}$ that is inverted less efficiently. A preliminary analysis shows that in order for $\mathbf{V}$ to have full rank, it is necessary—yet not sufficient—that every column vector has at least one non-zero entry. A column consists of only zeroes with probability $(1 - \frac{k}{n})^n$, which is approximately e^{-k} for large n. It follows that k must grow at least logarithmically in n for the expected number of columns consisting solely of zeroes to remain constant. We followed [26] and set $k = \log n + 2$, which, according to simulation, results in an invertible matrix with very high probability. If a peer still ends up with a singular matrix, all it has to do is to download one or more additional linear combinations.

3.5 Slices

We have seen previously that the decoding of the received linear combination blocks is quite time-consuming, which is especially bothersome as the process can only start as soon as the last of the n blocks has arrived, which is the moment at which the typical user of a peer-to-peer client would expect to be able to use the downloaded file.

A simple yet effective solution to this problem is proposed, among others, in [7] and [14], which just divides the file to be downloaded in multiple segments of fixed size. Linear combinations are then only ever built using blocks from the same segment. We call these segments *slices* and let them consist of a specific number n_s of blocks. Every aspect of the source coding mechanism operates on a single slice in lieu of the entire file, which means that as soon as all n_s blocks of a specific slice have been received, the decoding of that slice can begin even if no other blocks from the file have been downloaded at all. This has two profound advantages, the first being that the decoding of the whole file takes time linear in the number of slices instead of polynomial in file size, the other that the decoding process is more evenly spread during the duration of the download as single slices can be decoded individually.

3.6 Helper Blocks

A helper block is created for every slice of the download. It serves two purposes: First, it contains the orphaned bits at the end of each block. Suppose we have a block size of 128 KByte, so that a block consists of 33'825 symbols of 31 bits each. This leaves an incomplete symbol with the size of a single bit at the end of every block that is not accounted for in the construction of the linear combinations and thus never transmitted. We could, of course, only allow block sizes that align with symbol boundaries (e.g. 124 KByte), but storing the incomplete symbols in a separate block is a more flexible approach. Even when choosing a worst case block size such as 120 kByte, resulting in 30 surplus bits per block, the helper block for a 128 MB slice is approximately 4 kByte in size, which should be acceptable.

Second, there is a slight probability that the 31 bits which make up a symbol are all ones. As the order of our finite field is $q = 2^{31} - 1$, we cannot represent this bit pattern. For this reason, the helper block also indicates the location of the occurrences of this specific pattern. With compressed file formats being overwhelmingly used in the distribution of digital content, the frequencies of such a particular bit pattern—note that the 31 ones also have to be located exactly inside the boundaries of a symbol—should be small and not inflate the size of the helper block unreasonably.

The helper blocks are exchanged directly and never used for source coding. To distinguish them from normal blocks, they have an ID consisting solely of zeroes. As helper blocks are comparably light-weight, they are not subject to the tit-for-tat policy of normal block exchange, and a peer newly joining a swarm will always try to download the helper blocks first, so they stay available in the network.

3.7 Seeding and the Bootstrap Problem

We previously mentioned the bootstrap problem of newly joining nodes. In order to make it possible for new peers to acquire a set of blocks with which they can start engaging in tit-for-tat exchange, seeders will offer a small set of blocks to any leecher for free. It makes sense to relax the tit-for-tat demand for the specific combination of seeders and starting nodes, as the former have nothing additional to gain from further exchanges and the latter no blocks to offer. In contrast to the *Fast Extension* mechanism of BitTorrent, only seeders will offer such free sets, as leechers never build any new combinations. When a newly joined node contacts a seeder, the seeder will calculate a pseudo-random set of block IDs and announces them to the other node. The seed for the PRNG that generates the set is the new node's IP address and the size of the set is proportional to the number of nodes in the swarm, which can be estimated using the number of neighbors the seeder sees.

Note that a peer's seeding status is applied to individual slices and not to the whole file. A peer can very well be already seeding a certain slice while it has only received few blocks from another slice. Accordingly, the free starting set is provided per slice, so that a new node has a starting capital for every slice to begin trading.

The proposed scheme serves two purposes: First, every peer will be assigned and provided with a set of unique linear combinations, which increases block diversity and ensures that the newly joining node has blocks that other peers will be interested in. Second, as all seeders will calculate the same free set for a given peer and the size of the free set depends on the (estimated) size of the swarm, free-riding by malicious peers is not possible and the burden on the seeders remains low.

4 Data Integrity

4.1 Introduction

While the previous sections were concerned with aspects of fairness and network coding, an area that has so far been left open is the topic of security. A peer-to-peer network should be able to provide protection from adversaries that try to upload bogus data into the swarm. While it seems very hard to ensure that the file a user downloads from the network is really the one he was looking for, we can find ways to at least make sure that every peer receives the exact same copy of the originally seeded file.

In BitTorrent, data integrity is provided by means of a list of hashes in the Torrent's *metainfo file*. For every piece in the data file, a 160-bit SHA-1 hash is computed and stored. Upon receiving a piece from the network, a node can instantaneously check whether the piece has been altered—either already at the source or mid-transit—by computing the hash of the received data and comparing it to the correct hash. This approach is straightforward enough, although a user still has to trust the publisher of a BitTorrent file to include valid hashes; then again, the publisher could just provide an inaccurate description of the downloaded file's content.

From the previous sections it becomes immediately clear that storing and communicating hashes in this fashion cannot be a viable way to ensure data validity in the T4T environment. As the potential number of tradeable blocks—the number of linear combinations that can be constructed—is $\binom{n}{k} \cdot n_s$ where n is the number of blocks per slice and n_s stands for the number of slices in the file, one would have to store and transmit much more data for the hashes than for the data itself, even using large blocks and small hash sizes.

Pre-computing all possible hashes is therefore out of the question. But there is another way: observe that while encoding the data to be transmitted, we just perform an addition of symbols in a symbol space $GF(q)^m$. If there were a function that had the one-way-ness required for hashing, yet still offer a possibility to compute the hash of the addition of two symbols using only their hash values, we could use it as a basis for our data integrity scheme. Basically, we are looking for some sort of homomorphism between the symbol space and the hash value space whose inverse is intractable. Fortunately, such functions have already been discovered and described, and we will use them for our purposes.

Another way to ensure data integrity that has recently been proposed is the use of so-called *secure random checksums*. We quickly discuss this approach in Section 4.7, but do not use them in the context of *BitThief*.

4.2 Homomorphic Encryption

A homomorphic encryption scheme allows certain arithmetic operations to be performed on ciphertexts. Different variants of this property are found, for instance, an additively homomorphic cryptographic algorithm may allow the decryption of the addition of two ciphertexts to be the same as the addition of the original plaintexts. Known homomorphic cryptography schemes so far only support group operations on plaintexts; one could imagine that there also exist systems that preserve a ring structure of the message space.

These homomorphic encryption schemes are especially useful in scenarios where someone who does not have decryption keys needs to perform arithmetic operations on a set of ciphertexts, for example the tallying of votes in a secure electronic voting system or for tamper-resistant data aggregation in sensor networks. In other scenarios, the property of homomorphism is undesired, as it automatically implies *malleability* of the cryptographic system. A malleable cryptography scheme offers the possibility for an adversary to transform a given ciphertext c into another ciphertext $\hat{c}$, whose corresponding plaintext $\hat{m}$ is related to the original plaintext m via the relation $\hat{m} = f(m)$ for some known function f.

In fact, most asymmetric cryptography systems that are used today have homomorphic properties and are thus malleable. As public-key cryptography is in practice mostly used to derive a session key for the actual, symmetric encryption and decryption of a message, this is not of much concern. In any case, signatures may be used to thwart an adversary's tampering attempts.

We give a more formal description next: let $\mathcal{A}$ be a cryptographic algorithm with encryption function $\boldsymbol{\varepsilon}$ and corresponding decryption function $\boldsymbol{\delta}$ operating on a message space M and a ciphertext space C. If we have operations $+, \times$ such that M is a group under $+$ and C is a group under $\times$, then we say that $\mathcal{A}$ is $(+, \times)$-homomorphic if

$$c_1 \times c_2 = \boldsymbol{\varepsilon}(m_1 + m_2)$$

or, alternatively,

$$m_1 + m_2 = \boldsymbol{\delta}(c_1 \times c_2).$$

4.3 Different Homomorphic Encryption Schemes

We will now have a look at different homomorphic encryption schemes and analyze whether they could be adapted as a basis for our data integrity mechanism. As we are essentially looking for a one-way function, and asymmetric cryptography is based on trapdoor functions (encrypting is easy, but decrypting is hard without knowledge of a key), we can start by looking at existing public-key cryptography systems. Many of those can be broadly divided into the following three classes defined by their underlying intractable problem:

4.3.1 Factorization-Based

One of the first public-key cryptography systems that gained public recognition was the *RSA* algorithm designed at MIT by Rivest, Shamir and Adleman and described in [32]. While Diffie and Hellman had previously come up with a scheme for securely exchanging keys over a public channel (which was based on the discrete logarithm problem discussed next), RSA also covered the actual encryption and decryption process.

RSA assumes that the problem of factoring large integers is hard to solve, specifically factoring semiprimes (the product of two primes). A quick overview of the procedure follows: Alice chooses large primes P, Q and takes the product

N. She then computes the totient of N, which is $\phi(N) = (P-1)(Q-1)$ (as P, Q are prime). She also has to choose a public exponent e, smaller than and co-prime to $\phi(N)$.[6] Her private key exponent d is then computed to satisfy $de \equiv 1 \pmod{\phi(N)}$, which can be efficiently done using the extended Euclidean algorithm.

Alice transmits (N, e) to Bob, who encrypts his plaintext m to a ciphertext $c = m^e \mod N$. Alice can decrypt m using her private key exponent d, as $m = c^d \mod N$.

Let us take a look at the homomorphic properties of this scheme. For messages m_1, m_2 and corresponding ciphertexts c_1, c_2, we have:

$$\varepsilon(m_1 \cdot m_2) = (m_1 \cdot m_2)^e = m_1^e \cdot m_2^e = c_1 \cdot c_2$$

While this is an example of homomorphic encryption, this particular case is ill-suited for our purposes as the construction of transmitted blocks is done by adding symbols together, not multiplying them. Thus we have to look beyond RSA's one-way function.

4.3.2 Discrete-Logarithm-Based

These are cryptographic systems which were designed under the assumption that the problem of computing a discrete logarithm is intractable, i.e., finding x such that $g^x = h$ for given g, h where all operations take place in a finite cyclic group.

Diffie and Hellman first described the use of the discrete logarithm problem in public-key cryptography in their seminal article [9]. Their algorithm was strictly for distributing keys (establishing a common secret over insecure channels) and did not cover encryption. The *El Gamal* cryptography system described in [12] is probably the best known DL-based scheme[7] based on the work of Diffie and Hellman. We thus focus on El Gamal next.

Again we quickly describe the encryption process: Alice first chooses a large prime p and selects a generator g for a cyclic multiplicative group G of order p. Her private key is a random integer $x, 0 \leq x < p$, her public key is $y = g^x \mod p$. She then transmits a description of G, the generator g and her public key y to Bob.

Bob encrypts a message $m \in G$ by choosing a random $k \in \mathcal{Z}, 0 \leq k < p$ and computing a tuple (c_1, c_2):

$$c_1 = g^k, c_2 = my^k$$

The ciphertext is $c = (c_1, c_2)$. Alice can decrypt c with her private key:

$$m = \frac{c_2}{c_1^x}$$

El Gamal is homomorphic, but again multiplicatively. For random k_1, k_2 and with a slight abuse of notation:

$$\varepsilon(m_1 \cdot m_2) = (g^{k_1+k_2}, (m_1 \cdot m_2)y^{k_1+k_2}) = (g^{k_1}, m_1 y^{k_1}) \cdot (g^{k_2}, m_2 y^{k_2}) = c_1 \cdot c_2$$

[6]A popular choice for e is $2^{16} + 1$.

[7]It is used in the popular cryptography software PGP, for example.

El Gamal does not appear to be directly usable for our purposes. What we can do, however, is to take El Gamal's basic one-way function g^x as the encrypting function ε and notice that it is homomorphic over addition:

$$\varepsilon(m_1 + m_2) = g^{m_1+m_2} = g^{m_1} \cdot g^{m_2} = c_1 \cdot c_2$$

It looks like we found a homomorphism that fulfills our requirements. We will take this approach further in Section 4.4.

4.3.3 Elliptic Curve-Based

Cryptography systems based on the mathematical objects known as elliptic curves have first been proposed by Neal Koblitz [21] and Victor S. Miller [28] in 1985. An elliptic curve is a smooth algebraic curve which can generally be characterized by the Weierstrass equation:

$$y^2 = x^3 + ax + b$$

The points on such a curve can be shown—together with an identity element O, the *point at infinity*—to form an abelian group with respect to multiplication. If one chooses point coordinates from a finite field, the solutions (points) of the equation for given coefficients a, b form a finite abelian group, in which the discrete logarithm problem is believed to be more difficult than the corresponding problem in the underlying finite field. It is thus assumed that cryptography algorithms using elliptic curves (*ECC*) requires much smaller key sizes than other systems while maintaining the same security.

In practice, one usually chooses as underlying finite field either a *prime field* $GF(p)$, containing a large prime number p of elements, or a *binary field* $GF(2^m)$, where m is called the *degree* of the field. Binary fields have the advantage that their elements can be represented as bit strings of length m and that the field arithmetic can be efficiently implemented in terms of operations on those bit strings.

The underlying one-way function of ECC systems is very similar to the case previously discussed: Define an elliptic curve E_K over a finite field K and a point G on said curve of order r, r being a large prime. The number of points on the curve is $n = fr$ for some integer f[8]. We can define addition over E_K such that $(E_K, +)$ represents an Abelian group with O acting as its identity. For points P, Q, we especially have $P + Q = O$ if $P = -Q$ and $P + Q = Q$ if $P = O$. The negative $-Q$ of a point $Q = (x, y)$ is the point $(x, p - y)$ for a finite field of order p.

Multiplying a point P by an integer n is intuitively defined as adding P to itself n times and is analogous to the exponentiation operation in multiplicative groups. Then the problem

Given E_K and a point Q on E_K, find an integer x such that $Q = xG$, if such x exists.

is believed to be intractable. The best algorithms that solve it take time

[8] f, called the *cofactor*, is usually a small integer ≥ 1.

exponential in the size of the curve.[9] Calculating xG, however, can be done in $O(\log_2 x)$ point doublings and additions. Thus the function

$$h(x, n) : E_K \times \mathbb{N} \to E_K, \quad h(x, n) = nx$$

together with a suitable mapping of curve points to hash values could be used as the basis for a hashing scheme. The homomorphism of the scheme is easily confirmed:

$$h(x_1 + x_2, n) = x_1 n + x_2 n = h(x_1, n) + h(x_2, n)$$

As finding sensible choices for the parameters (prime modulus p, coefficients a, b, number of points n, base point G and its order r) of a ECC system is non-trivial, and suboptimal values can seriously compromise the security of the system, the National Institute of Standards and Technology (NIST) has published a document [30] concerning federal standards for digital signature schemes that lists recommended curves. In particular, there are 5 prime fields and 5 binary fields of different orders (which correspond to the resulting key lengths) recommended.

Elliptic curve cryptography is a comparably young field and much work remains to be done; also, there are already some issues about patents that seem to cover certain areas. Still, its use is growing and it seems that ECC is becoming a viable alternative to existing public key cryptography systems. We thus tried to design a data integrity protocol based on elliptic curve cryptography; see the next section for the results.

4.4 Choice and Implementation of Two Algorithms

Here we describe the two hashing algorithms we designed for the T4T system. First some words about notation: As before, we split the file to be shared into n blocks,[10] each block consisting of m symbols in $\mathbb{Z}_q$, where q is the Mersenne prime $2^{31} - 1$. We use bold symbols $\mathbf{b}, \mathbf{c}, \mathbf{v}$ for vectors. The file is then regarded as a $m \times n$ matrix $\mathbf{F}$, where the j^{th} column vector corresponds to the j^{th} message block $\mathbf{b}$ of the file. For notational convenience, we refer to the i^{th} symbol of block $\mathbf{b}$ as b_i. When we build linear combinations of the blocks during the source coding process, we combine random k of the n blocks into a combination block $\mathbf{c}$. In order to identify which original blocks were used in the combination, every block $\mathbf{c}$ has a corresponding block id vector $\mathbf{v} \in \{0, 1\}^m$ representing the coefficients of the linear combination.

As for the implementation itself, there was and still is considerable concern about the performance of the hashing algorithms as they require multiple expensive operations in large finite fields. The standard Java `BigInteger` class has proven to be inadequate for this purpose, so all the integer arithmetic was re-implemented using the *GNU MP Bignum Library*[11] (libgmp), which provided

[9]Compare this to case of finding a discrete logarithm in groups generated by a large prime where there are algorithms that run in subexponential time.

[10]Actually, the file is first divided into individual slices. As blocks from different slices are not mixed, we can simplify things by using the term file and keeping in mind that the following holds true for every slice.

[11]See http://gmplib.org/.

a significant speed boost. The library is called via a small wrapper interface written in C++ which in turn is available to the Java T4T code via JNI.

An additional performance gain was achieved by using a simple yet clever algorithm for computing a product of powers, taken from [4]. The algorithm is listed in appendix A.2.

4.4.1 DL-Based

For our first attempt at designing a suitable hashing mechanism, we choose to use a scheme based upon the discrete logarithm problem in finite fields. The following algorithm was first proposed by Bellare et al. in [5] as an efficient way for recalculating hashes of messages that are incrementally updated. In [22], Krohn et al. further refined the concept and adapted it for a network coded file distribution system, utilizing its homomorphic properties.

We briefly explain the scheme here: First we choose a set of hash parameters G consisting of large random primes (q, p) where $|p| = \lambda_p$, $|q| = \lambda_q$ and $q|(p-1)$, and a vector of generators $\mathbf{g}$ that have a multiplicative order modulo p of q, similar to the scheme that is used for DSA (see again [30]). Because the symbols that make up the block to be hashed are elements from $\mathcal{Z}_q$, we set $q = 2^{31} - 1$ and choose p appropriately.

As $G = (q, p, \mathbf{g})$ is public, there has to be a way to provide clients with a possibility to check the soundness of the parameters, or else a dishonest node might publish hash parameters that enable it to find collisions and thus poison the network with bogus data. See appendix A.1 for an algorithm that enables such proof.

We then define the hash function H_G for a block $\mathbf{b}$ as follows:

$$H_G(\mathbf{b}) = \prod_{i=1}^{m} g_i^{b_i} \mod p$$

To verify an incoming block $\mathbf{c}$ with block id vector $\mathbf{v}$, a node needs to check that

$$H_G(\mathbf{c}) = \prod_{i=1}^{n} H_G(\mathbf{b_i})^{v_i} \mod p$$

holds, which works because of the homomorphic property of H_G such that for two blocks $\mathbf{b_1}$ and $\mathbf{b_2}$:

$$H_G(\mathbf{b_1} + \mathbf{b_2}) = H_G(\mathbf{b_1}) \cdot H_G(\mathbf{b_2})$$

The security analysis of this scheme was done in [22], which in turn refers to [5]. It can be shown that if there is an algorithm $\mathcal{A}$ that can find collisions on H_G with probability p_A and time t, then we can use an oracle machine to construct algorithm $B = U^A$ that succeeds in breaking the DL problem $\log_g(x)$ with probability $p_A/2$ and in polynomial time. However, it must be noted that the original analysis was for a scheme that operated just on a cyclic multiplicative group G_p of large prime order p, not on the subgroups characterized by (q, p) as in this system. The discussion of the implications this has for the security of the scheme was absent from [22] and is beyond our capabilities. It surely holds that finding collisions is at least as hard as computing the discrete logarithm in a multiplicative group of order q, but a better analysis would be desirable.

4.4.2 ECC-Based

We also designed a scheme that is very similar to the previous one, but operates on elliptic curves. Again we have our data block $\mathbf{b}$ consisting of symbols $b_1, \ldots, b_m$, $0 \leq x_i \leq q$ for $1 \leq i \leq m$. We choose an elliptic curve E_K over a group K (the NIST curve *P-192* in particular), and define a mapping ϕ that injectively (with respect to the x-coordinate) maps points on E_K to K, so that $\phi(P) = \phi(P')$ implies $P = P'$ or $P = -P'$. We can just use the affine x-coordinate of a point P for that purpose. We then randomly choose generating points S_i, $1 \leq i \leq m$ of prime order greater than q, which are the public parameters of the hashing function H_{E_K}. H_{E_K} is defined as:

$$H_{E_K}(\mathbf{b}) = \phi\left(\sum_{i=1}^{m} b_i S_i\right)$$

The basic operations of elliptic curve arithmetic – mainly point addition, doubling and multiplication – were taken from the excellent "Guide to Elliptic Curve Cryptography" [17].

The security of this scheme can be shown by reducing it to the discrete logarithm problem in E_K, similar to the case for the DL-based algorithm in 4.4.1: If there exists an algorithm $\mathcal{A}$ that is able to compute collisions for H_{E_K} in time t and with probability p, then there is an algorithm $\mathcal{B} = U^{\mathcal{A}}$ which calculates the discrete logarithm $\log_P(Q)$ for given points $P, Q \in E_K$ with probability $2p/m$ and in polynomial time.

$\mathcal{B}$ works as follows: we choose random values w_i, $1 < w_i < q$ for $1 \leq i \leq m$ and a random index j, $1 \leq i \leq m$. We then define

$$\hat{H}_{E_K}(x) = \phi(x_1\hat{S}_1 + \ldots + x_m\hat{S}_m)$$

with

$$\hat{S}_i = \begin{cases} w_i P & \text{for } i \neq j \\ w_i Q & \text{for } i = j \end{cases}$$

If the $\hat{S}_i$ are not pairwise distinct, we either already found the discrete logarithm by chance (if a $\hat{S}_k, k \neq j$ is identical to $\hat{S}_j$, then we know that $\log_P(Q) = \frac{w_k}{w_j}$), or we choose different w_i.

Now we use the oracle $\mathcal{A}$ to find a collision of two blocks/messages $\mathbf{x}, \dot{\mathbf{x}}$ so that $\hat{H}_{E_K}(\mathbf{x}) = \hat{H}_{E_K}(\dot{x})$. We know from the definition of ϕ that the points on the curve E_K that correspond to $\mathbf{x}, \dot{\mathbf{x}}$ have the same x-coordinate. Thus, either

$$(x_1 - \dot{x}_1)\hat{S}_1 - \ldots - (x_m - \dot{x}_m)\hat{S}_m = O$$

or

$$(x_1 + \dot{x}_1)\hat{S}_1 + \ldots + (x_m + \dot{x}_m)\hat{S}_m = O$$

holds. Furthermore, as $\mathrm{ord}(\hat{S}_i) > q$, for at least 2 indices i, the coefficients term $(x_i \pm \dot{x}_i)$ does not vanish. We namely have $1 \leq c, d \leq m$ so that $x_c \neq \dot{x}_c$ and

$x_d \neq \dot{x}_d$. Now if $j \in \{c, d\}$ we can solve the equation that results from the collision and get the discrete logarithm $\log_P(Q)$:

$$Q = \left(\frac{1}{w_j(x_j \pm \dot{x}_j)} \sum_{1 \le i \le m} w_i(x_i \pm \dot{x}_i) \right) V$$

As all elements from $E_K \setminus \{O\}$ are generators of the cyclic group, the distributions of the tuples $(\hat{S}_1, \ldots, \hat{S}_M)$ are independent of the choice of j, as is the behavior of $\mathcal{A}$ on input $\hat{H}$. Thus we have the claimed success probability of $2p/m$. The algorithm requires m point multiplications and several computations in $\mathbb{Z}_q$ for the final calculation of $\log_P(Q)$ and is thus polynomial in its input length.

4.5 Batching

In order to further speed up the process of verification, we can use a batching technique. The homomorphic property that allows us to verify every possible linear combination of original blocks in the first place makes it also possible to build a linear combination of some received blocks and just verify that combination. Therefore, instead of independently checking received blocks $\mathbf{c}, \mathbf{c}'$ with ID vectors $\mathbf{v}, \mathbf{v}'$, we can verify the combination $\mathbf{c} + \mathbf{c}'$ for ID $\mathbf{v} + \mathbf{v}'$.

A node can thus introduce a batching window of a certain size l, wait until enough blocks have arrived to fill the window, and then verify the sum of the blocks in the window. This way, the most expensive operation, the calculation of the hash of a new block, needs to be performed only for every l blocks.

We have to be aware, however, that this procedure enables an attack where a malicious peer can transmit construct two bogus blocks $\mathbf{f}, \mathbf{f}'$ based on real blocks $\mathbf{b}, \mathbf{b}'$ with $\mathbf{f} = \mathbf{b} + \epsilon, \mathbf{f}' = \mathbf{b}' - \epsilon$ for some random ϵ, which will remain undetected as long as they are checked in the same batching window [16]. There is a simple mechanism that will thwart such attacks: using random weight coefficients for the blocks in the batching window. When doing so, an attacker would have to produce two blocks with errors that, when multiplied by random coefficients w_j, w_i, will cancel each other, which is highly unlikely.

All put together, we verify the integrity of l blocks $(\mathbf{c_1} \ldots \mathbf{c_l})$ in a batching window by producing a vector of random integer coefficients $\mathbf{w} = [w_1 \ldots w_l]$ and check that

$$H\left(\sum_{j=1}^{l} w_j \mathbf{c_j} \right) = \prod_{j=1}^{l} H(\mathbf{c_j})^{w_j} \ .$$

This way, instead of $l \cdot m$ exponentiations to verify l blocks, we only need to do $l + m$, which leads to a near linear speedup in l.

More advanced batching techniques are described in [4], unfortunately they are not applicable to our system as they do not assume different generators g for different symbols.

Window size	Speed (KByte/s)	
l	G4	C2D
1	51	129
8	411	1040
16	787	2063
32	1626	4096
64	3061	8325

Block size 128 KByte (33825 Symbols), $\lambda_p = 512$

Window size	Speed (KByte/s)	
l	G4	C2D
1	71	189
8	584	1515
16	1155	2961
32	2134	5957
64	4469	11924

Block size 128 KByte (33825 Symbols), $\lambda_p = 384$

4.6 Evaluation

In order to evaluate the algorithm described in 4.4.1, we wrote a simulation framework that goes through all the steps encountered in the verification process: It produces linear combinations of a file of random data, computes the hashes of the original blocks and verifies the integrity of the encoded blocks using those hashes. We ran the simulation on a PowerPC G4 (*G4*) laptop clocked at 1.67 GHz and on a desktop computer equipped with a Intel Core 2 Duo (*C2D*) CPU at 2.67 GHz.

The results confirm the expectation that batching increases the verification speed approximately linear in the batching window size l. The choice of the security parameter λ_p also influences the performance, as the time taken for the operations in $\mathbb{Z}_p$ increases with the size of p.

We also built a proof-of-concept simulator for the ECC-based algorithm from section 4.4.2 that performed the same steps. However, its performance was found to be lackluster. This is mainly due to the fact that the elliptic curve we used, *P-192*, is defined over a finite field with an order that is some multiple magnitudes larger than the one we used in 4.4.1. The usage of an elliptic curve over a smaller finite field would certainly be an interesting direction for future

Window size	Speed (KByte/s)	
l	G4	C2D
1	127	293
8	993	2340
16	1973	4686
32	3771	9351
64	6989	18788

Block size 128 KByte (33825 Symbols), $\lambda_p = 256$

work; unfortunately choosing appropriate domain parameters for a custom curve is far from trivial (especially counting the points on the curve), and brings the risk of inadvertently constructing a weak curve (that is, one which is susceptible to several known attacks).

Additionally, if we have a closer look at the necessary operations, we notice that the algorithm we used for multiplying a point G with an integer b requires approximately $2.5 \cdot |b|$ squarings, $3 \cdot |b|$ multiplications and $1.5 \cdot |b|$ inversions in the underlying field [17]. Compare this to the about $1.5 \cdot |b|$ multiplications that are needed to compute g^b for the discrete log case. It is clear that the often claimed performance advantage of ECC stems form the fact that—maintaining the same security level—key sizes can be about half as large as those in systems based on prime-generated groups and not due to inherently less expensive arithmetics.

4.7 Secure Random Checksums

Secure random checksums have been proposed by Gkantsidis et al. in [14] as a simple alternative to more complex and computationally expensive homomorphic hashing functions. SRCs work well in Galois Fields of the form $GF(2^q)$, while homomorphic hashing takes place in modular fields of prime order, where arithmetic operations are more expensive.

SRCs are created by a server in possession of the complete file. The server chooses a vector $\mathbf{r} = [r_1 \dots r_m]$ of random coefficients from the same field that is used for the source coding operations. The secure random checksum of an original block $\mathbf{b}$ is then defined as the sum of pairwise products of $\mathbf{r}$ and $\mathbf{b}$:

$$\mathrm{SRC}(\mathbf{b}) = \sum_{i=1}^{m} r_i b_i$$

This process is repeated for all n blocks of the file, and the SRCs together with $\mathbf{r}$ are transmitted to the client.[12] Because of the linear nature of the computation, it is obvious that the SRCs of the original blocks can be used to calculate SRCs for any received encoded blocks. A peer that received a combined block $\mathbf{c}$ with associated ID $\mathbf{v}$ needs just to check that (here, $\mathbf{b}[i]$ refers to the i^{th} original block of the file)

$$\sum_{j=1}^{m} r_j c_j = \sum_{j=1}^{m} r_j \left(\sum_{i=1}^{n} v_i \mathbf{b}[i] \right)$$

SRCs have some very compelling advantages over the homomorphic hashing schemes discussed earlier:

Hash sizes are very small. A hash for a block is just a single symbol, meaning that for a field where symbols can be encoded in q bits, we only need $n \cdot q$ bits for all SRCs of a file, plus a small constant number of bits for the seed to the PRNG.

Speed of computation is several orders of magnitudes faster. The authors of [14] achieved a performance of 2 GBytes/s on a 3.0 GHz Pentium 4

[12]It suffices of course to just send a seed for a PRNG instead of the whole vector $\mathbf{r}$.

CPU for the calculation of SRCs, which is close to the cost of reading the file and much faster than the rate at which encoded blocks are produced.

However, these advantages come at a price. As the knowledge of $\mathbf{r}$ enables a malicious node to effortlessly produce bogus blocks, a separate random vector $\mathbf{r}$ has to be uniquely assigned to each peer in the swarm by a trusted authority. This also has the consequence that the trusted party has to recalculate and securely transmit SRCs to every newly joining node (as a malicious node could get $\mathbf{r}$ from the SRCs).

There is thus a tradeoff between performance and confidentiality requirements in the usage of SRCs. While they certainly represent an interesting approach to ensuring data integrity in a source-coded peer-to-peer environment, we chose not to pursue this method further.

5 Future Work

In this section we introduce some concepts that are of interest for future work on *BitThief* and *T4T*.

5.1 Large Scale Testing

While we believe the concepts of the presented system to be sound and performed unit and small-scale tests with a few nodes, the size and dynamic nature of a real BitTorrent swarm will pose additional challenges. Deployment at a multitude of different nodes with varying connectivity and computing resources will be required to truly evaluate the fitness of *T4T*.

5.2 Revisiting ECC

As we noted in Section 4.6, our attempt at implementing a data integrity mechanism based on elliptic curve arithmetic was under-performing. This can be attributed to the fact that we used a curve whose security parameters are designed to provide unbreakable encryption for quite some time into the future. As a BitTorrent/*BitThief* swarm is usually not long-lived, we could content ourselves with much weaker security. A curve over a finite field of size, say, 64 bits would probably be secure enough for our purposes, yet offer a massive performance gain over the 192-bit field we used. As [30] lists only curves at least as strong as the one we used, one would to have to find suitable domain parameters. [23] presents a procedure to construct elliptic curves with given group order over large finite fields, which could be used for this purpose.

5.3 Making BitThief Independent of BitTorrent

At the moment, a *BitThief* client, even if using the *T4T* protocol to communicate with other *BitThief* clients, is still reliant on a BitTorrent tracker to find peers who offer a specific file. A future version could get rid of that dependency by implementing its own tracker mechanism or even come up with a completely decentralized solution such as a DHT overlay that handles file and peer lookups.

5.4 Identity Concealment

BitThief uses a specific peer ID format in the initial BitTorrent connection handshake and a bit in the reserved field to identify itself to other *BitThief* clients so that a *T4T* connection can be initiated. It is imaginable that other BitTorrent clients will start refusing connection to a recognized *BitThief* client because of its perceived selfishness. In order to circumvent such a scenario, *BitThief* would need to be able to fake the identity of a regular BitTorrent client while still recognizing fellow *BitThief* clients. This could, for instance, be done by sending characteristic yet unsuspicious looking bitfield messages after the handshake.

5.5 Enhancing Performance

Decoding a received slice and verifying the hashes of incoming blocks are both expensive processes. This is a cause for concern as it may prolong the time until the file is completely downloaded and ready for use, which is annoying to users and thus detrimental to the adoption of the *T4T* protocol among *BitThief* clients. Any work that goes toward increasing the decoding and verifying process would therefore be welcome. As an example, the multiplication step of the decoding process is well-suited to be implemented using the SIMD instructions of modern processors.

5.6 Precomputing Exponentiation Tables

Similar to 5.5, the computation of all the hashes of the blocks of a large file poses a considerable burden on a seeding peer. This is mostly because of the multiple finite field exponentiations that need to be performed. If we use k-ary exponentiation instead of the usual iterative-squaring technique, we can trade an additional memory requirement of $(2^x - 1)/x$ times the original method for a factor of $x/2$ in speed increase (minus the time for the precomputation) for a chosen x [22].

6 Conclusions

We presented the *T4T* system, a peer-to-peer communication protocol that utilizes source coding to create a fair sharing network where data is exchanged in a strict tit-for-tat fashion. The computational complexity of the scheme is lower than in other network coding systems, but still considerable when compared to existing peer-to-peer file sharing networks that do not take fairness issues into account. Our system solves the block diversity and bootstrap problems inherent to true tit-for-tat sharing while preventing the exploitation of seeding peers that is possible in protocols such as BitTorrent.

To ensure the integrity of the transmitted data, we introduced a hashing mechanism that is based on a homomorphic hashing function. Peers can verify an incoming block by computing its expected hash value out of the original blocks' hashes. The scheme's security properties are equivalent to other systems relying on the hardness of the discrete logarithm problem. Our solution's weak spot is its performance, which is much worse than those of traditional hashing algorithms due to the expensive arithmetic operations in large finite fields.

We have to point out that an important prerequisite of the presented work is the fact that all peers in a BitTorrent swarm are competing for the same file, enabling us to only take momentary exchanges between two peers into account with regard to fairness. Thus the tit-for-tat technique we used is not readily applied to other types of peer-to-peer file sharing environments where all peers form one contiguous network, trading a multitude of different files. How well source coding mechanisms are able to provide fairness and robustness in these systems is the subject of ongoing research.

Another interesting point is how true fairness affects the performance of the network as a whole. Many users of peer-to-peer networks connect to the Internet with asymmetrical connections that have larger downstream than upstream capacities. In the *T4T* system, these peers will only be able to download data from the swarm at the same rate at which they upload, while in other peer-to-peer networks, altruistic peers and seeders help overcome the asymmetry. The total download rate in a swarm utilizing *T4T* is therefore expected to be lower than in the same swarm using classic BitTorrent sharing, if many peers are bound by asymmetrical connections.

It will be intresting to see whether the future development of BitTorrent is affected by the concepts we showed. If free-riding clients such as *BitThief* become prevalent, users of traditional clients might resort to forming closed wsharing communities with strict membership control. Anonymous swarms would then have to use *T4T* or another form of enforced collaboration in order to survive, thereby perhaps setting an example for other and future peer-to-peer networks. If the possible success of *BitThief* will lead to increased consideration of fairness and collaboration issues in peer-to-peer systems' design, it would have surpassed its original purpose and introduced the perspective of the rational peer—and what can be learned from it—into file sharing. Because sometimes, thieves are more honorable than honest men.

References

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

[2] R. Ahlswede, N. Cai, S.-Y. Li, and R. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.

[3] R. Axelrod and W. D. Hamilton. The evolution of cooperation. *Science*, 211(4489):1390–1396, 1981.

[4] M. Bellare, J. A. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In *Advances in Cryptology (EUROCRYPT '98)*, volume 1403 of *Lecture Notes in Computer Science*, pages 236–250, 1998.

[5] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology (CRYPTO '94)*, volume 839 of *Lecture Notes in Computer Science*, pages 216–233, 1994.

[6] D. Chaum and T. P. Pedersen. Transferred cash grows in size. In *Advances in Cryptology (EUROCRYPT '92)*, volume 658 of *Lecture Notes in Computer Science*, pages 390–407, 1992.

[7] P. A. Chou, Y. Wu, and K. Jain. Practical network coding. In *Proc. 51st Allerton Conference on Communication, Control, and Computing*, 2003.

[8] B. Cohen. Incentives build robustness in BitTorrent. In *Proc. First Workshop on Economics of Peer-to-Peer Systems (P2PEcon*, 2003.

[9] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[10] M. Feldman and J. Chuang. Overcoming free-riding behavior in peer-to-peer systems. *SIGecom Exch.*, 5(4):41–50, 2005.

[11] M. Feldman, K. Lai, I. Stoica, and J. Chuang. Robust incentive techniques for peer-to-peer networks. In *Proc. ACM Conference on Electronic Commerce (EC)*, pages 102–111, 2004.

[12] T. E. Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology (CRYPTO '84)*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18, 1985.

[13] F. D. Garcia and J.-H. Hoepman. Off-line Karma: A decentralized currency for peer-to-peer and grid applications. In *Proc. Third International Conference on Applied Cryptography and Network Security (ACNS)*, volume 3531 of *Lecture Notes in Computer Science*, pages 364–377, 2005.

[14] C. Gkantsidis, J. Miller, and P. Rodriguez. Comprehensive view of a live network coding P2P system. In *Proc. 6th ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 177–188, 2006.

[15] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *Proc. 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 2235–2245, 2005.

[16] C. Gkantsidis and P. Rodriguez. Cooperative security for network coding file distribution. In *Proc. 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1–13, 2006.

[17] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, New York, NY, USA, 2004.

[18] D. Hughes, G. Coulson, and J. Walkerdine. Free riding on Gnutella revisited: The bell tolls? *IEEE Distributed Systems Online*, 6(6), 2005.

[19] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust algorithm for reputation management in P2P networks. In *Proc. 12th International Conference on World Wide Web (WWW)*, pages 640–651, 2003.

[20] S. Katti, H. Rahul, W. Hu, D. Katabi, and J. Crowcroft. Network coding made practical. Technical report, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2006.

[21] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[22] M. N. Krohn, M. J. Freedman, and D. Mazieres. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 226–240, 2004.

[23] G.-J. Lay and H. G. Zimmer. Constructing elliptic curves with given group order over large finite fields. In *Proc. First International Symposium on Algorithmic Number Theory (ANTS)*, pages 250–263, 1994.

[24] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting BitTorrent for fun (but not profit). In *Proc. 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[25] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap. In *Proc. 5th Workshop on Hot Topics in Networks (HotNets)*, 2006.

[26] T. Locher, S. Schmid, and R. Wattenhofer. Rescuing tit-for-tat with source coding. In *Proc. 7th IEEE International Conference on Peer-to-Peer Computing (P2P)*, 2007.

[27] G. Marwell and R. E. Ames. Experiments on the provision of public goods. II. Provision Points, stakes, experience, and the free-rider problem. *The American Journal of Sociology*, 85(4):926–937, 1980.

[28] V. S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology (CRYPTO '85)*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426, 1985.

[29] P. Moor. Free riding in BitTorrent and countermeasures. Master's thesis, ETH Zürich, Zürich, Switzerland, 2006.

[30] National Institute of Standards and Technology. Digital signature standard (DSS). *Federal Information Processing Standards Publication 186-2*, 2000.

[31] T. G. Papaioannou and G. D. Stamoulis. Effective use of reputation in peer-to-peer environments. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 259–268, 2004.

[32] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[33] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Computing and Networking (MMCN)*, 2002.

[34] H. Schulze and K. Mochalski. Internet study 2007. Technical report, ipoque GmbH, 2007.

[35] M. Wang and B. Li. How practical is network coding? In *Proc. 14th IEEE International Workshop on Quality of Service (IWQoS)*, pages 274–278, 2006.

A Algorithms of Interest

A.1 Generating Provably Secure Hashing Parameters

The algorithm described in 4.4.1 requires parameters $G = (q, p, \mathbf{g})$ that are public yet sensitive to the security of the system. A malicious node could, for example, choose $\mathbf{g}$ so that it knows i, j, x_i, x_j with $g_i^{x_i} = g_j^{x_j}$, making it possible to compute hash collisions with ease.

Nodes thus have to be able to ensure that G was honestly chosen. The following algorithm, taken from [22], is able to deterministically and securely generate G from a seed s. We use the MD5-hash of the file as listed in the metainfo file as s and feed it to the pseudo-random number generator $\mathcal{G}$. We write $\mathcal{G}(x)$ to indicate that we retrieve the next pseudo-random value from $\mathcal{G}$, scaled to the interval of integers $\{0, \dots, x-1\}$. The differences to the original version from the cited article stem from the fact that our q is fixed.

Function `pickGroup(`λ_p, m, s`)`

```
Input: $\lambda_p, m, s$
Output: $q, p, \mathbf{g} = [g_1 \dots g_m]$
Seed PRNG $\mathcal{G}$ with $s$.
$q \leftarrow 2^{31} - 1$
repeat
    $p \leftarrow pGen(\lambda_p)$
until ($p \neq 0$) ;
for $i = 1$ to $m$ do
    repeat
        $x \leftarrow \mathcal{G}(p-1) + 1$
        $g_i \leftarrow x^{(p-1)/q} \mod p$
    until ($g_i = 1$) ;
end
return $(p, q, \mathbf{g})$
```

Function `pGen(`q, λ_p`)`

```
Input: $q, \lambda_p$
Output: $p$
for $i = 1$ to $4\lambda_p$ do
    $X \leftarrow \mathcal{G}(2^{\lambda_p})$
    $c \leftarrow X \mod 2q$
    $p \leftarrow X - c + 1$   // $p \equiv 1 \pmod{2q}$
    if $p$ is prime then
        return $p$
    end
end
return 0
```

A.2 Computing a Product Of Powers

The *fastMult* algorithm[13] from [4] can be used to efficiently calculate a product a of a series of powers $a_i^{b_i}, 0 < i \leq n$. It takes $(1 + n/2) \cdot log_2 b$ multiplications. In contrast, a naive algorithm uses $n \cdot (1 + E_a(b)) - 1$ multiplications, where $E_a(b)$ is the number of multiplications required to calculate a^b (approximately equal to $1.5 \cdot \log_2 b$).

```
Function fastMult($a_1, b_1, \ldots, a_n, b_n$)
  Input: Tuples $(a_1, b_1), \ldots, (a_n, b_n)$
  Output: $a = \prod_{i=1}^{n} a_i^{b_i}$
  $a \leftarrow 1$
  for $j = t$ downto 1 do
    $a \leftarrow a^2$
    for $i = i$ to $n$ do
      if $b_i[j] = 1$ then
        $a \leftarrow a \cdot a_i$
      end
    end
  end
  return $a$
```

[13]Note that the original version of the algorithm contained an off-by-one error that we have corrected here.

B *T4T* Protocol Specification

B.1 Nomenclature

File A stream of data that is served by a given torrent. May—contrary to its name—also consist of multiple concatenated files (in that case, the directory structure is preserved in the torrent metafile).

Piece A contiguous part of a *file*, used as an exchange unit in the original BitTorrent specification. It has a given nominal piece size (usually a power of 2). The piece size is typically chosen based on the total amount of file data in the torrent, constrained by the fact that piece sizes too large cause inefficiency, and too small a piece size will result in a large .torrent metadata file. The last piece of a file may have a smaller length. Data integrity of transmitted pieces is ensured by hash codes stored in the metainfo file.

Block The analogous unit of exchange in the *T4T* protocol. Unlike a *piece*, a block is not direct file data, but a linear combination of k parts of the original file from the same *slice* (see below). Another difference is that the transfer of a block is an atomic operation in *T4T*, while in the BitTorrent protocol, pieces are further divided into sub-pieces (usually of size 16 kB) that are exchanged. For simple interaction with clients that do not support *T4T*, the block size s_b should be chosen in relation to the original piece size s_p so that $\gcd(s_b, s_p) = \min(s_b, s_p)$. Sometimes the expression "original block" will be used to refer to the special linear combination of just one single part of a slice, i.e., a part of the original file that has length s_b. If the length of the last block of the last slice of a file is smaller than s_b (which is likely), then that block is padded with zeroes so that it can still be used to compute linear combinations.

Slice A slice is a contiguous part of a *file* with a specific length of c original blocks. c must by divisible by 8 to facilitate message handling. Only original blocks from the same slice are used when computing linear combinations, thus restricting the size of the linear equation system required to get the original data out of the received blocks to $c \times c$. The length of the last slice of the file may be smaller than the other slices.

Peer ID / Session ID The peer ID consists of 20 bytes that uniquely identify a client in the original BitTorrent protocol. There are different conventions on how to construct a peer ID. *BitThief* so far uses an ASCII string that consists of the prefix "M4-4-0–" followed by random data. The peer ID is only used in the initial BitTorrent connection handshake. In the native *T4T* protocol, a 2-byte session identifier defined in the initial handshake is used to distinguish between connections. Note that session IDs are only required to uniquely identify connections between the same two clients that engage in the exchange of different files simultaneously, so 2 bytes should be enough for the foreseeable future.

File ID 20 bytes that uniquely identify a served file. In the BitTorrent specification, this is constructed by computing a SHA-1 hash over the "info" key's data from the metainfo file's dictionary (in the "info" entry, the checksums of all pieces of a given file are stored). For a start, it will be sufficient to use this bit string as file ID in the *T4T* protocol as well.

Block ID A unique identifier for a block inside a certain slice. It is constructed by taking a bit string of length c and filling in ones for every original block that was used in the linear combination and 0 else, resulting in a bit mask indicating which original blocks a block consists of. There are two special Block IDs: The ID where all bits are set to zero refers to the helper block of a given slice, while the ID consisting of c ones identifies the hash block of the slice (as normal block IDs have exactly $0 < k < c$ bits set, there is no ambiguity).

B.2 Handover from Standard BitTorrent Protocol to *T4T* and Communication Concepts

The *T4T* Protocol is employed only after a standard BitTorrent connection has been established; a *T4T*-enabled client sends the initial HandShake message with reserved bit 49 set. If the responding peer indicates that it also supports *T4T*, the initiating peer immediately sends a *T4T* hello message. When the receiving peer acknowledges by returning a corresponding hello message, the handover is complete and only *T4T* protocol messages are exchanged until the connection is closed. It would also be imaginable to drop the existing connection and initiate a new connection on a different port so that a *T4T* session could be initiated separately of a BitTorrent connection; this would however require the user to keep track of an additional port that he may have to forward. In any case, the initiator of the *T4T* will send his handshake message with the File ID of the torrent he is interested in, and a random 16-bit session identifier that will be used in all later messages. The receiving node acknowledges with a hello message containing the same File ID and Session ID, upon which the connection is established.

Peers will send a slice list request shortly after establishing a connection to get an overview of the availability of blocks at the remote node. If the remote node offers blocks within slices that the local peer has yet to complete; or if the remote peer is a seeder, the local peer will send a block list request message for a slice it is interested in. The remote peer will answer with a block list message containing all linear combinations it possesses for the specified slice, or with a block list that contains the well-defined[14] set of blocks the local node is allowed to download for free if the remote peer is a seeder. If the local node can find an "innovative" block in the block list, that is, a linear combination it is not yet in possession of, it reacts by sending a block request message for that block. Alternatively, if the local peer has only none to few blocks of a slice, it may skip the block list request and send a block suggestion request to which it gets a response with a block ID the remote peer is able to provide, and for which the local peer may then send a request.

Either way, the remote peer will eventually respond to the block request with

[14]The local peer's class C subnet address is used as seed for a PNRG which computes a set of linear combinations (blocks). The size of the set is dependent on the number of peers in the swarm, which can be estimated using the size of a node's neighborhood.

a block message that contains the requested block, or a block denial message if it is not able or willing to fulfill the request. A block denial message can be caused by the local peer having to send a block first because it has already downloaded a block from the remote peer.

If the local peer is no longer in use of a block it previously requested but did not receive yet (because it could already complete the linear equation system for the given slice with other blocks), it may send a block request cancel message, which causes the remote peer to respond with a block denial message.

Optionally, when a peer has downloaded a certain number of blocks from other nodes, it may send a new block message to all the non-seeding nodes in its neighborhood containing the block IDs of the newly acquired blocks. That way, peers have a more or less up-to-date overview of which blocks are available at their neighbors without having to repeatedly request block lists, which are quite expensive. The new block message is optional as it is not orthogonal to the block list and block suggestion mechanism.

Peers can request the addresses of additional *T4T*-enabled clients in the swarm by sending a peer list request, which is answered by a peer list message. To prevent an idle connection from being dropped, peers may send keep-alive message every once in a while.

B.3 Message Types

Hello

The *Hello* message starts communication between *T4T*-enabled peers (seeders and non-seeders alike). It is acknowledged by returning a *Hello* message with the same file ID if the local peer serves that file. Otherwise, the connection is dropped.

Peer List Request

The *Peer List Request* message asks the remote peer to return a list of *BitThief* (or other *T4T*-aware) peers. This message type is optional as *BitThief* clients anyway register with a standard BitTorrent tracker as of now and can get a list of peers from there (albeit with no discrimination between peers that support *T4T* and others).

Peer List

The *Peer List* message is sent upon receiving a *Peer List Request*. It contains a list of peers (seeders and non-seeders) that are known to the local peer.

Slice List Request

The *Slice List Request* prompts the remote peer for a list indicating the availability of blocks within the different slices.

Slice List

The *Slice List* message answers a *Slice List Request* and consists of a vector that states for every slice of the served file how many unique blocks the sending peer has to offer. Using a flag, the sender can in turn request the remote peer's slice list. A special form of the slice list message indicates that the sender is a seeder and will provide a well-defined set of blocks from every slice to any requesting peer.

Block List Request

The *Block List Request* is sent to acquire information about the actual blocks that the remote peer is offering for a given slice.

Block List

The *Block List* message is the response to a *Block List Request* and contains a vector listing every available block within the specified slice. As uniquely identifying a block in a slice requires c bits and there may be up to c available blocks per slice, efforts have to be taken to minimize the amount of *Block List* messages.

Block Suggestion Request

The *Block Suggestion Request* message is used by a client that possesses only none to few blocks for a specific slice. It prompts the remote peer for a suggestion of a block it may send.

Block Suggestion

The *Block Suggestion* message contains the ID of a block that the sending peer possesses. The receiving peer may then respond with a block request for that block if it is innovative for him (which is highly likely given the great block diversity stemming from the source coding).

Block Request

The *Block Request* message intends to initiate a block transfer by asking the remote peer for a specific block from a given slice. It is responded to by ultimately sending either a *Block Delivery* or a *Block Denial* message. The helper block and the hash block of a slice are also requested with a *Block Request*.

Cancel Block Request

The *Cancel Block Request* message informs the remote peer that a pending block request should no longer be considered valid. The response is a *Block Denial* message for the given block.

Block Delivery

The *Block Delivery* message handles the actual transfer of a block. As a block is atomically transmitted in its entirety, this message can have a considerable length. Helper blocks and hash blocks are also transmitted inside a *Block Delivery* message.

Block Denial

The *Block Denial* message is either the negation of a pending block request from a remote peer or the acknowledgment of a received *Cancel Block Request.* Reasons for the first case may be a violation of the tit-for-tat principle (the remote peer needs to provide a block before being able to receive another) or that the sending peer is not in possession of the requested block, for example.

Keep Alive

As peers may drop a connection to a remote peer after a certain idle time, the *Keep Alive* message may be regularly sent over an otherwise idle connection to reset the idle timer.

New Block

The *New Block* message is sent to all non-seeding peers in a peer's neighborhood to inform them of newly acquired blocks. This is done to avoid sending more than a single block list message to a remote peer. To further reduce overhead, multiple blocks may be announced in the same New Block message.

B.4 Message Format

Note: All data types are encoded in *big endian* mode.

Hello

(Type −1)

Type	Length	Reserved	File ID	Session ID
1 byte	4 bytes	4 bytes	20 bytes	2 bytes

Type Message type ID — *-0x01*

Length The number of bytes in the whole message — *31*

Reserved 32 reserved bits for future extension of the protocol — *0x00000000*

File ID Unique identifier for downloaded file, might use BitTorrent's InfoHash at first for compatibility — *20-byte SHA-1 hash of the info key in the BT metainfo file*

Session ID Random 16-bit string established in the handshake and used in all further messages. The initiator of the connection chooses a session ID, the receiver acknowledges it by responding with the same ID — *e.g. 0x2C06*

Peer List Request

(Type −2)

Type	Length	Session ID	# of Peers
1 byte	4 bytes	2 bytes	1 byte

Type Message type ID — *-0x02*

Length The number of bytes in the whole message — *variable*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

# of Peers The maximum number of peers the responding peer should send (up to 50) — *0x01 – 0x32*

Peer List

(Type −3)

Type	Length	Session ID	# of Peers	Peers
1 byte	4 bytes	2 bytes	1 byte	n * (4 bytes + 2 bytes)

Type Message type ID — *-0x03*

Length The number of bytes in the whole message — *variable*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

# of Peers The number of peers in the list — *0x01 – 0x32*

Peers A list of *T4T* peers with their IP and port

Slice List Request

(Type −4)

Type	Length	Session ID
1 byte	4 bytes	2 bytes

Type Message type ID — *-0x04*

Length The number of bytes in the whole message — *7*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

Slice List

(Type –5)

Type	Length	Session ID	Request	Bits per Slice (BPS)	Block Availability
1 byte	4 bytes	2 bytes	1 Bit	7 Bits	$\frac{\text{BPS}\times\#\text{ of slices}}{8}$ bytes

Type Message type ID — *-0x05*

Length The number of bytes in the whole message — *variable*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

Request Flag indicating that the receiving peer should send its slice list as well — *0x0 or 0x1*

Bits per Slice How many bits per slice are used to encode the availability of blocks. The maximum number of available blocks per slice is c as c independent linear combinations suffice to reconstruct the original slice; thus $\lceil \log_2 c \rceil$ bits are required for each block at most. If this field is zero, then the sending peer is a seeder/source and can provide blocks from any slice — *0x01 – 0x7F or 0x00*

Block Availability A list of n integer values, each BPS bits long, that states how many blocks are available for each corresponding slice $s_i, i \in [0, n-1]$; or empty if sender is a seeder

Block List Request

(Type –6)

Type	Length	Session ID	Slice Number
1 byte	4 bytes	2 bytes	4 bytes

Type Message type ID — *-0x06*

Length The number of bytes in the whole message — *11*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

Slice Number The number of the slice for which a block list is requested — *0x00000000 – 0xFFFFFFFF*

Block List

(Type –7)

Type	Length	Session ID	Slice Number	# of Blocks	Blocks
1 byte	4 bytes	2 bytes	4 bytes	2 bytes	$\frac{\text{\# of blocks} \times c}{8}$ bytes

Type Message type ID — *-0x07*

Length The number of bytes in the whole message –*variable*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

Slice Number The number of the slice for which the block list follows — *0x00000000 – 0xFFFFFFFF*

# of Blocks The number x of blocks in the block list. $x \in [1, c]$

Blocks A bit string that contains the concatenated block IDs that the sending peer possesses

Block Suggestion Request

(Type –8)

Type	Length	Session ID	Slice Number
1 byte	4 bytes	2 bytes	4 bytes

Type Message type ID — *-0x08*

Length The number of bytes in the whole message — *11*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

Slice Number The number of the slice for which a block suggestion is requested — *0x00000000 – 0xFFFFFFFF*

Block Suggestion

(Type –9)

Type	Length	Session ID	Slice Number	Suggested Block
1 byte	4 bytes	2 bytes	4 bytes	$\frac{c}{8}$ bytes

Type Message type ID — *-0x09*

Length The number of bytes in the whole message — *variable*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

Slice Number The number of the slice which contains the suggested block — *0x00000000 – 0xFFFFFFFF*

Suggested Block Block ID of a block from the specified slice that the sending peer is able to provide

Block Request

(Type −10)

Type	Length	Session ID	Slice Number	Block
1 byte	4 bytes	2 bytes	4 bytes	$\frac{c}{8}$ bytes

Type Message type ID — *-0x0A*

Length The number of bytes in the whole message — *variable*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

Slice Number The number of the slice from which a block is requested — *0x00000000 – 0xFFFFFFFF*

Block Block ID of the requested block. Two special cases exist: Block ID 0, i.e. all zeroes, indicates the helper block, while the block ID consisting of only ones refers to the hash block.

Cancel Block Request

(Type −11)

Type	Length	Session ID	Slice Number	Block
1 byte	4 bytes	2 bytes	4 bytes	$\frac{c}{8}$ bytes

Type Message type ID — *-0x0B*

Length The number of bytes in the whole message — *variable*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

Slice Number The number of the slice of the specified block — *0x00000000 – 0xFFFFFFFF*

Block Block ID of a block which has been previously requested

Block Delivery

(Type −12)

Type	Length	Session ID	Slice Number	Block ID	Block Data
1 byte	4 bytes	2 bytes	4 bytes	$\frac{c}{8}$ bytes	block size (s_b) bytes

Type Message type ID — *-0x0C*

Length The number of bytes in the whole message — *variable*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

Slice Number The number of the slice of the following block — *0x00000000 – 0xFFFFFFFF*

Block ID Block ID of the following block

Block Data Raw data of the specified block

Block Denial

(Type −13)

Type	Length	Session ID	Slice Number	Block ID
1 byte	4 bytes	2 bytes	4 bytes	$\frac{c}{8}$ bytes

Type Message type ID — *-0x0D*

Length The number of bytes in the whole message — *variable*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

Slice Number The number of the slice for which the block request is denied — *0x00000000 – 0xFFFFFFFF*

Block ID The ID of the block that is denied

Keep Alive

(Type −14)

Type	Length	Session ID
1 byte	4 bytes	2 bytes

Type Message type ID — *-0x0E*

Length The number of bytes in the whole message — *7*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

New Blocks

(Type -15)

Type	Length	Session ID	# of Blocks	Slice/Block List
1 byte	4 bytes	2 bytes	2 bytes	# of blocks $\times$ (4 bytes + $\frac{c}{8}$ bytes)

Type Message type ID — *-0x0F*

Length The number of bytes in the whole message — *variable*

Session ID Random 16-bit string established in the handshake to discriminate multiple connections between the same two peers — *e.g. 0x2C06*

# of Blocks The number of blocks in the block list

Slice/Block List A list consisting of (slice number, block ID) tuples