Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

Rational File Sharing

submitted by

Anton Krohmer

on September 4, 2009

Supervisor

Prof. Dr. Michael Backes

Advisor

M. Sc. Oana–Mădălina Ciobotaru

Reviewers

Prof. Dr. Michael Backes
Dr. Dominique Unruh

Affidavit

I hereby declare that this bachelor's thesis has been written only by the undersigned and without any assistance from third parties. Furthermore I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Declaration of Consent

Herewith I agree that this thesis will be made available through the library of the Computer Science Department. This consent explicitly includes both the printed, as well as the electronic form attached to this thesis. I confirm that the electronic and the printed version are of identical content.

Saarbrücken, September 3, 2009

(Date) (Signature)

Acknowledgement

I am deeply grateful to Prof. Dr. Michael Backes, for offering me such an interesting research topic and for his support during the writing of the thesis. Not to mention the great lectures and seminars he gave, as those were one main reason why I got interested in cryptography.

I sincerely thank my advisor, Oana Ciobotaru, for her continuous support during the writing. She helped me evaluating my ideas and writing the thesis. Moreover, she proofread the thesis countless times and gave me great feedback such that I always knew where I was standing.

I also thank Ole Rehmsen and Radu Curticapean, my friends and fellow students, for proofreading my thesis, and Hendrik Molter for our fruitful discussions and his design for a logo.

Further on, I would like to thank a few people who had indirect impact on this thesis. First, I express my deepest gratitude to my family who always supported me morally as well as financially. Then, I am thankful to all of my friends who made life worthwhile during my studies.

Finally, I owe special thanks to Lydia Weißbach, who encouraged and supported me throughout my whole bachelor studies. Her help I always greatly appreciated.

Danksagung

Ich bin Prof. Dr. Michael Backes zutiefst dankbar für das interessante Forschungsthema, an welchem ich arbeiten durfte, und für seine Hilfe während ich die Bachelorarbeit schrieb. Ich bin ihm aber auch für die großartigen Vorlesungen und Seminare, welche er anbot, dankbar, da sie mein Interesse in Cryptography geweckt haben.

Ich danke aufrichtig meiner Betreuerin, Oana Ciobotaru, für ihre anhaltende Unterstützung während des Schreibens. Sie half mir, meine Ideen auszuwerten und auf Papier zu bringen. Außerdem las sie unzählige Male Korrektur und gab mir Rückmeldung was noch zu erledigen war.

Ich danke außerdem Ole Rehmsen und Radu Curticapean, meinen Freunden und Kommilitonen, für das Korrekturlesen der Bachelorarbeit, und Hendrik Molter für die interessanten Diskussionen und sein Design für ein Logo.

Ich möchte weiterhin noch einigen Leuten danken, welche nur indirekt zu dieser Bachelorarbeit beitrugen. Zuerst möchte ich besonderen Dank gegenüber meiner Familie ausdrücken, welche mich stets moralisch und finanziell unterstützt hat. Des weiteren bin ich allen meinen Freunden dankbar, die während dem Studium das Leben lebenswert gemacht haben.

Schlussendlich möchte ich Lydia Weißbach besonders danken, da sie mich während des Studiums immer ermutigt und unterstützt hat. Ihre Hilfe habe ich sehr geschätzt.

Abstract

Although BitTorrent is a commonly used peer-to-peer file sharing protocol, it can be easily cheated. We construct “RatFish”, a new file sharing protocol for rational users who try to maximize download speed while uploading as little as possible. We use cryptographic primitves for building incentives for users to follow the protocol and we prove rational users cannot cheat. Threats outside our model are also analyzed, as well as the feasibility of the protocol in reality.

Contents

1 Introduction

BitTorrent is a common file sharing protocol. Based on incentives, it tries to take into account the rational behavior of users as well as their altruism. Experience has shown that BitTorrent yields extraordinaire results compared to previous peer-to-peer applications. However, the protocol behaves only this well because the commonly used clients are implemented to be honest. There exist several clients that bypass the incentives which BitTorrent should provide. As long as most of the people are using fair-minded implementations of the protocol, this might not be a problem. However, if for example the music industry wants to get rid of this particular file sharing protocol, they just need to distribute clients that abuse BitTorrent's weaknesses. Every user would have an incentive to use malicious clients, as the individual download rates get better. However, if everyone starts using such a malicious client, the protocol will not work anymore.

1.1 BitTorrent

In this section, we want to explain how BitTorrent works and look at the reason why it works so much better than previous file sharing applications.

1.1.1 History of File Sharing

Before BitTorrent, a lot of other approaches on file sharing existed. The easiest and most common one is internet hosting. That is, a server offers the file which clients can simply download. Of course, this works only if the server offers reasonable download rates, if the file is not too big and if not too many users want the file. Otherwise, the download rates for every single user decrease to a non-acceptable amount. This imposes a problem especially for new and popular files which have a big file size, for example a new Linux kernel or the latest World of Warcraft patch. Sometimes also a normal internet user wants to share a file with other people. Clearly, he can't afford an expensive server for this. This is when people began using peer-to-peer file sharing.

The first peer-to-peer applications, like Kazaa or eDonkey, did only secondarily tackle the missing upload speed. These applications were used to offer files to other users. If a file is offered by many users, then the download speed for this particular file is high. However, people were not given any incentive to share. A lot of users just downloaded their desired file without contributing to the system. This was the main problem with all of those protocols, because there was a tremendous amount of users downloading files, but only a few of them were altruistic and also offering some. Often, those networks were also used to illegally distribute pirated copies. Then, this effect was even more amplified because a lot of users were afraid of being sued. Therefore, all the peer-to-peer approaches basically suffered from the same problems as server-based file sharing.

1.1.2 BitTorrent's Design

The author of BitTorrent, Bram Cohen, tried to address exactly these issues. First of all, he changed the basic model of file sharing. He separated the file search from the file download. While you could use the the preceding peer-to-peer applications to actively search for files, in BitTorrent there is a network of its own for every file shared. Clearly, this means that you can only effectively share files with at least a mediocre popularity and a reasonably big file size, usually between 100 MB and several GB. However, for files which are not popular or significantly smaller you can still safely use internet hosting.

Now that we have a file which is reasonably big and that a lot of people want, we need a mechanism that ensures scalability of the system. This means, the more people are trying to download, the bigger the overall throughput should get. In the end, this is achieved by enforcing the people who download the file to upload it at the same time, even when their download is not yet complete. This is possible by splitting the file into small pieces (usually of size 256 kB). Then, every downloader (*leecher* from now on) simply acts rational: He uploads to those people who upload to him. This way, every leecher in the system has a strong incentive to upload pieces, as he will otherwise receive a poor download speed. We look in detail at the exact mechanisms in the next sections.

1.1.3 The Metainfo File

As already pointed out, in BitTorrent, every shared file has its own network. To be able to join such a network, one needs a metainfo file, the so-called *torrent file* (the name is derived from the fact that such files hold the file extension ".torrent"). The contents of the torrent file are further on used to provide the properties of the shared file. It consists of

- the URL of the tracker,
- the length of the file,
- the name of the file,
- the number of bytes per piece
- and a 20-byte SHA-1 hash for every piece of the file.

Using the URL of the tracker, a new client is able to locate the network which was created for the particular file he wants to download. The SHA-1 hashes are used by the client to verify the integrity of pieces he obtains while participating in the BitTorrent protocol.

1.1.4 The Tracker

To coordinate all actions which happen in BitTorrent, a server, the so-called *tracker* is used. Obviously, the workload on the tracker is much smaller than on a server who shares the whole file. Basically, the tracker is used for one task: Introducing peers to each other. A joining leecher therefore sends a HTTP GET request to the tracker, who responds with a random subset of all users known to him which are currently participating in the protocol.

In addition to this rather simple task, trackers usually keep track of the downloaded and uploaded amount of users. This is often used by registered torrents (i.e., one has to register to a site before being able to download a torrent) to keep track of your global share ratio. People who do not share enough in comparison to their download can then be excluded by the tracker. On a side note, those statistics are sent in by the clients themselves. This makes it trivial to forge yourself a good share ratio. However, this problem is beyond the scope of the thesis.

Also, some clients offer ways to run BitTorrent without a tracker. Azureus, a popular BitTorrent client, first used a DHT (distributed hash table) implementation to create a distributed database. Some time later the original BitTorrent client added an incompatible DHT implementation on its own, which now most clients like μTorrent or BitComet support. Moreover, clients also use the so-called “peer exchange” where two parties exchange their known IPs of other users.

1.1.5 Peers

A *peer* is a user who participates in the BitTorrent protocol. The set of peers is the union of all leechers, who are still downloading, and all *seeders*, who obtained the file and just stay in the system for altruistic purposes. Because the systems tend to be quite large (up to 10,000 peers), a peer does not have connections to everybody else but to a random subset of all other participants, who form his *local neighborhood.* This local neighborhood is initially received from the tracker and then extended by further requests to the tracker or by peer exchanges with other users. The typical size of such a neighborhood is between 20 and 50 connections. A client then uses only his local neighborhood for downloading pieces.

From all existing connections not every one is used. In general, there are only a few (around 4-10) connections actively used to transmit data. The others, currently idle connections, are said to be *choked.* Consequently, the ongoing transmissions are *unchoked.*

1.1.6 The Uploading Strategy

It is crucial to understand how the uploading strategy of peers works in order to understand the popularity of BitTorrent.

Piece Selection Algorithm

A leecher always has to choose which piece he wants to download next. This is achieved using a simple but effective strategy, called *rarest first*. Here, the leecher looks at the availability of pieces in his local neighborhood and requests the rarest piece he still needs and his partner can offer. This intuitively minimizes the probability of being stuck or, put it more precisely, not having any piece to offer to his neighbors. If there are several pieces which fulfill the rarest first requirement he just chooses a random one.

Reciprocation

Here, we will discuss to whom a leecher will upload. Again, the idea behind this is simple. If a leecher has a neighbor who gives him good download speed, he does not want this peer to leave. Therefore, he starts *reciprocating* to the other leecher, which means that he also uploads data back to convince the other leecher to stay with him.

A leecher splits his overall upload speed into n equal parts. A common value for n is 4. This is called the *equal split*, because a leecher always reciprocates using the same speed. Then, he picks from all his neighbors those $n - 1$ leechers from which he gets the fastest download speed and starts uploading to them. This is reevaluated every 10 seconds.

Every 30 seconds, the leecher additionally starts uploading data to someone at random in his neighborhood. This *optimistic unchoke* has two purposes. First, it is supposed to find someone who gives the leecher even more download speed than his current partners. Second, it helps newcomers to join the system, as they don't have any pieces to offer.

1.2 Exploiting BitTorrent

So far, BitTorrent's mechanisms seemed to give good incentives for following the protocol: Intuitively, as long as you want some good download speed, you also have to offer something back. However, there are two points that immediately seem prone to misuse.

1.2.1 Free Riding

Free riding means that you download from a system without giving something back. In [LMSW06], Locher et al. presented a malicious client, *BitThief*, which works with uploading absolutely nothing while still being almost as fast as the original BitTorrent client. In general, it abuses the optimistic unchokes of leechers, and of course the seeders' altruism.

First, BitThief connects several times anew to the tracker to get a much bigger neighborhood than usual. With a bigger neighborhood, also the chances of getting an optimistic unchoke vastly increase. Then, the client just needs to wait to be optimistically unchoked over and over again until he completed the download.

This rather simple attack poses a serious threat to BitTorrent. If everyone starts to use BitThief, the BitTorrent protocol breaks down immediately, because there is no one left to upload data. Currently, one can only speculate about the reason why nobody uses BitThief. Either people feel altruistic enough to share data, or nobody uses malicious clients due to their small degree of popularity.

1.2.2 Strategic Clients

Considering that we are able to be as fast as the original BitTorrent client without uploading, one could ask how much faster one can get if one additionally uploads. This brings us to our second point of attack, which is the equal split strategy. The original client does not distinguish between clients to whom he uploads, he always uploads with the same speed to the first $n-1$ people who give him the fastest download speed.

Piatek et. al. developed a so–called *strategic client* in [PIA$^+$07], which abuses this fact. To understand how it works, imagine being a user with 400 kB/s upstream and the original BitTorrent implementation. Then, your equal split is 100 kB/s and you reciprocate to 3 people and have one connection for optimistic unchoking. Further on, imagine that 4 people are uploading to you with respectively 300 kB/s, 200 kB/s, 40 kB/s and 10 kB/s.

Now, assume a new client joins the system who has 2000 kB/s overall upload speed. If he split his capacity using equal split, he would upload with 500 kB/s to us, and thus being reciprocated. However, to obtain the reciprocation, he does not need to upload with 500 kB/s, but only with 41 kB/s, because then he still is in our personal top 3 concerning upload speed. This way, he saved 459 kB/s upload stream which he can use for other leechers.

Piatek et. al. showed that using such a strategic client, they are able to obtain a median performance gain of about 70%. Again, one can only speculate why only few people use this client, since it is freely available for download.

1.3 Related Work

BitTorrent was invented and implemented by Bram Cohen, who describes its key ideas in [Coh03]. As we already mentioned, Piatek et. al. examined whether the incentives in BitTorrent really suffice and showed in [PIA$^+$07] that using some simple ideas, one can easily exploit the system. Similarly, Locher et. al. provided a client in [LMSW06] which works without contributing to the system at all, by announcing himself several times to the tracker. A little bit more general is the concept of the *Sybil attack* [Dou02], where an attacker on a peer-to-peer network creates several identities.

Further on, Shneidman examined how to exploit BitTorrent using backtracing in [SPM04]. Aside from the attack points we discussed, he found out that one should reconnect upon getting choked, because the chances of being optimistically unchoked then increase. Additionally, he showed that uploading garbage can help, because even if the garbage is detected, it still counts as valid incoming traffic. This especially comes in handy when you just joined a system and have nothing to offer yet.

Thommes and Coates give some sophisticated techniques on how to make BitTorrent more robust against these attacks in [TC05]. For example, they propose to choke those connections where the fairness (the fraction of downloaded over uploaded data) is too small. However, those modifications are just mitigation techniques.

Hales and Patarin give their explanation on why BitTorrent is so popular and effective although it is easily exploited in [HP06]. According to them, this is due to the fact that every shared file creates its own small network. People are more likely to act social for local networks they benefit from compared to global systems where every single user's influence is just a negligible fraction.

Finally, Qiu and Srikant created a fluid model of BitTorrent in [QS04], which they use for giving expressions for the average downloading time. They show a Nash equilibrium point exists, but they prove this by looking at different aspects than in this thesis. Namely, they show what happens in BitTorrent if everyone wants to maximize his download speed while being able to alter his upload speed and sticking to the protocol. According to their paper, users build groups such that in every group all users approximately have the same upload speed.

The cryptographic primitives we are going to use are all taken from Goldreich's reference books [Gol01] and [Gol04]. The basic definition and intuition on game theory as well as rational cryptography is given in [Kat08].

1.4 Our Contribution

We build a new file sharing protocol "RatFish". This is partially based on the ideas of BitTorrent but enhanced such that it provides stronger incentives. Furthermore, we *prove* that our protocol cannot be cheated by rational users.

For this, we first define a model in theory: We use rational cryptography, which itself is a combination of game theory and cryptography. This makes sense because one can model rational file sharing users well using utility functions as in game theory. On the other hand, we have strong tools from cryptography to make RatFish non-cheatable.

The utility functions basically say that every user wants to minimize the upload capacity as well as the time needed to complete a download. In the protocol, we will then secure certain steps, for example the piece exchange between two leechers, using cryptography. We stick to basic cryptographic tools such that our protocol is feasible in reality.

Afterwards, we prove that a user who is covered by our utility functions cannot increase his utility by deviating from the given protocol, as long as everyone else sticks to it. This is a basic concept from game theory, namely the Nash equilibrium.

We also reason that RatFish is applicable in reality. Finally, our protocol defends against all attacks that we mentioned in this chapter and we believe it can successfully replace BitTorrent.

1.5 Overview

This thesis presents a file sharing protocol that cannot be cheated by rational players. We construct such a protocol in two different models where the first model is idealistic but intuitive and the second model is close to reality.

- All necessary prerequisites, i.e., definitions from game theory and cryptography, are given in Chapter 2.
- Chapter 3 introduces the idealistic model, which is unrealistic but gives good intuition on what will await us in the next chapter.
- Chapter 4 is the main part of the thesis. It describes a model which is close to reality and gives a rational file sharing protocol, inspired by BitTorrent, along with a proof that a rational player cannot get better by deviating from the protocol.
- Chapter 5 discusses what happens if the assumptions which we required in the realistic setting are violated.
- Chapter 6 will reason why the protocol which we worked out in theory is also feasible in reality by using constraints of modern server architecture and personal computers.
- Chapter 7 concludes our thesis by summarizing our achievements and giving an outlook on what still needs to be done.

2 Related Definitions

We want to make this thesis as self-contained as possible. Clearly, by only reading these definitions one probably will not understand the key concepts of game theory and cryptography. Nevertheless, we need these definitions in our proofs, and therefore introduce them here along with short explanations such that the reader does not have to consult any other source if he does not want to.

2.1 Game Theory

We use the basic ideas of game theory, namely the Nash equilibrium concept in a slightly altered form which is suitable for computational purposes.

2.1.1 Nash Equilibrium

A Nash Equilibrium indicates the situation when all parties of a game will stick to their actions because they do not gain more payoff by deviating. We introduce the pure-strategy Nash equilibrium for reasons of understanding. Later, we will only refer to the computational Nash equilibrium. The definitions are taken from [Kat08].

Definition 1 (Nash Equilibrium)**.** *Let* $\Gamma = (\{A_i\}_{i=1}^n, \{u_i\}_{i=1}^n)$ *be a Game, i.e.* A_i *is the set of actions player* i *can play and* $u_i : A_1 \times \ldots \times A_n \to \mathbb{R}$ *is the utility function for player* i*. Furthermore, let* $\boldsymbol{a}_{-i} := (a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$ *for arbitrary* $a_j \in A_j$*. A* Nash equilibrium *is a vector of actions* $\boldsymbol{a} = (a_1, \ldots, a_n) \in (A_1 \times \ldots \times A_n)$*, such that* $\forall i \in \{1, \ldots, n\}, \forall a_i' \in A_i$ *:*

$$u_i(a_i', \boldsymbol{a}_{-i}) \leq u_i(\boldsymbol{a}).$$

Basically, this means, that no player can get a better payoff by deviating from the action given by the Nash equilibrium, if every other player sticks to his designated action. To make this more intuitive, we show a simple example for such a game in figure 2.1. Here, we have two players where both of them have either the possibility to play “chicken”, which means to be on the safe side, or to play “dare” which can be interpreted as “wanting all or nothing”. Player one’s actions are depicted on the left and player two’s actions on the top, and the left side of a tuple represents the payoff for player one while the right side shows the payoff for player two.

	C	D
C	$(3,3)$	$(1,5)$
D	$(5,1)$	$(0,0)$

Figure 2.1: An example game, usually called chicken or dare.

One can check that we have two Nash equilibria here, namely the actions (C, D) and (D, C). Basically, this means that one player has to play "dare" while the other one chickens. Those outcomes are Nash equilibria, because if e.g. the player who chickens decides to play "dare", he gets 0 instead of 1 payoff, and if the player who has "dare" switches to "chicken", he obtains a reward of 3 instead of 5.

2.1.2 Computational Nash Equilibrium

Using the Nash equilibrium, we face several problems when players are represented by probabilistic polynomial-time interactive Turing machines (*ITM* from now on). First, the actions in a Nash equilibrium are chosen deterministically whereas such a Turing machine is probabilistic. We therefore do not talk about actions anymore but call the different Turing machines *strategies* for a player. Such a strategy will not deterministically pick an action to play but rather create a probability distribution over all possible actions of the player.

Second, one might need superpolynomial time for computing the best strategy, but we just have polynomial time to decide. To fix these issues, we introduce the *computational game.*

Definition 2 (Computational Game)**.** *A* computational game Γ *is a Game, where the played action of each participant i is computed by a probabilistic polynomially bounded ITM M_i, and where the utility u_i of each player i is polynomial-time computable.*

Instead of choosing a deterministic action, each player i now decides on a Turing machine M_i (from the set of all probabilistic Turing machines) and runs it. The Turing machine then outputs the action which eventually will be played by player i.

Definition 3 (Computational Nash Equilibrium)**.** *Let $\Gamma = (\{A_i\}_{i=1}^n, \{u_i\}_{i=1}^n)$ be a computational Game and k the security parameter. A strategy vector consisting of ITMs $\boldsymbol{M} = (M_1, \dots, M_n)$ is a* computational Nash equilibrium *if for all i and any probabilistic polynomial-time ITM M_i' there exists a negligible function ϵ such that*

$$u_i(k, M_i', \boldsymbol{M}_{-i}) - u_i(k, \boldsymbol{M}) \leq \epsilon(k).$$

Note that the utility functions u_i are now giving the expected payoff, because we have probabilistic strategies.

If you further on compare the inequality with Definition 1, you see that it is almost the same. The only difference is that you now also can play strategies which give you a worse payoff up to a negligible difference. If this was not allowed, the best strategy for players would be to always break cryptographic primitives, like encryptions. The chance for this to happen is still negligible, but if it succeeds, the payoff is in general huge. Therefore, we need this small addition in order not to force players into brute forcing cryptographic primitives.

We also introduce the *outcome* of a game, which basically is a transcript of everything that happened during an entire game. The notation of the utility function is overloaded. Whereas it describes the expected payoff for a strategy vector, it gives the exact payoff when applied on an outcome. This makes sense, as an outcome basically shows a finished game where every player can view his payoff directly.

Definition 4 (Outcome)**.** *An* outcome *of a computational Game consists of the players' inputs and the actions taken by them. In other words, an outcome is a complete transcript of a computational Game.*

2.2 Cryptography

As already pointed out, we use several standard cryptographic tools. Here, our definitions are close to those in [Gol01] and [Gol04], but taken from [Bac09].

2.2.1 Symmetric Encryptions

The most basic tool is an encryption of a secret message. For this, two parties need to share the same key (thus *symmetric*).

Definition 5 (Symmetric Encryption Scheme)**.** *An* encryption scheme *with* indistinguishable encryptions *is a triple* $(\mathsf{G}, \mathsf{E}, \mathsf{D})$ *of probabilistic polynomial-time algorithms satisfying the following three conditions:*

1. *On input* 1^n*, algorithm* G *(called* key-generator*) outputs a bit string.*
2. *For every* k *in the range of* $\mathsf{G}(1^n)$*, and for every* $\alpha \in \{0,1\}^*$*, algorithm* E (encryption) *and* D (decryption) *satisfy*
$$\Pr[\alpha = \alpha' : \alpha' \leftarrow \mathsf{D}(k, \mathsf{E}(k, \alpha))] = 1.$$
3. *For every polynomial-size circuit family* $\{\mathsf{C}_n\}$*, there is a negligible function* μ*, such that for every* $x, y \in \{0,1\}^{poly(n)}$ *(i.e.,* $|x| = |y|$*):*
$$\begin{aligned}&|\Pr\left[b = 1 : k \leftarrow \mathsf{G}(1^n), b \leftarrow \mathsf{C}_n(\mathsf{E}(k, x))\right]\\&- \Pr\left[b = 1 : k \leftarrow \mathsf{G}(1^n), b \leftarrow \mathsf{C}_n(\mathsf{E}(k, y))\right]| \leq \mu(n).\end{aligned}$$

This means that an adversary getting some cipher text c cannot tell whether this resulted from an encryption of message m or of message m', provided that both messages have the same length.

2.2.2 Digital Signatures

Also, we will use *digital signatures* to validate statements. A digital signature is used to express that a certain signer approves of a message. Of course, no one should be able to create a digital signature except for the signer himself, but everyone has to be able to check the validity. Therefore, the signer needs a key pair, a public key and the secret key.

Definition 6 (Digital Signatures)**.** *An* unforgeable signature scheme *is a triple* $(\mathsf{G}, \mathsf{S}, \mathsf{V})$ *of probabilistic polynomial-time algorithms fulfilling the following conditions:*

1. *On input* 1^n*,* G *outputs a pair of bit-strings* (s, v)*, called* signature key *and* verification key*, respectively.*
2. *For every* $x \in \{0,1\}^*$*, the algorithms* S (signing) *and* V (verification) *satisfy*

$$\Pr[\mathsf{V}(v, x, \mathsf{S}(s, x)) = 1 : (s, v) \leftarrow \mathsf{G}(1^n)] = 1.$$

3. *For every probabilistic polynomial-time machine* M *with access to an oracle* O *(denoted by* M^{O}*) it holds that*

$$\Pr\left[\mathsf{V}_v(x, y) = 1 \wedge x \notin Q_{\mathsf{M}}^{\mathsf{S}_s}(1^n, v) : (s, v) \leftarrow \mathsf{G}(1^n), (x, y) \leftarrow \mathsf{M}^{S_s}(1^n, v)\right]$$

is negligible in n*, where* $Q_{\mathsf{M}}^{\mathsf{O}}(x)$ *is the set of queries* M *issues to the oracle* O*, when given the input* x*.*

The second condition says that having the key s is needed for signing, but for verification it suffices to have the key v. The last condition basically only says that no matter how (polynomially) many signatures an adversary obtains on chosen messages, he is still not able to forge a signature on a new message.

2.2.3 Hash Functions

Finally, we will use *hash functions* to have an efficient way of checking whether one has received a correct piece or not. Hash functions are characterized by having a short output corresponding to a long input. In addition, if you are asked to find in polynomial time two different strings which hash to the same value (i.e. you are asked to find a *collision*), you will only succeed with negligible probability. Note that there *do* exist collisions, because there exist fewer short strings than long strings, but nevertheless finding one should be infeasible.

Definition 7 (Collision-Free Hashing Function)**.** *Let* $l : \mathbb{N} \to \mathbb{N}$*. A collection of functions* $\{h_r : \{0,1\}^* \to \{0,1\}^{l(n)}\}_{r \in \{0,1\}^*}$ *is called* collision-free hashing *if there exists a probabilistic polynomial-time algorithm* I *such that the following holds:*

- Admissible indexing (technical): *For some polynomial* $p(\cdot)$*, all* n*, and every* r *in the range of* $\mathsf{I}(1^n)$*, it holds that* $n \leq p(|r|)$*. Furthermore,* n *can be computed in polynomial time from* r*.*

- Efficient evaluation: *There exists a polynomial-time algorithm that computes $h_r(x)$ from r and x.*
- Collision-freeness: *We require that for every probabilistic polynomial-time algorithm* A,

$$\Pr[h_r(x) = h_r(x') \wedge x \neq x' : r \leftarrow \mathsf{I}(1^n), (x, x') \leftarrow \mathsf{A}(r)] \textit{ is negligible.}$$

3 Idealistic Setting

Before we give a protocol which works in a real life scenario, we first look at a simple and straightforward case. In this idealistic setting, we use some very strong assumptions which do not hold in reality. However, we are able to illustrate the basic idea of the protocol and give some important intuition which also comes in handy when considering the realistic setting.

3.1 Model

We capture the goals of the users in utility functions. Clearly, seeders and leechers should have different utility functions. It might even be possible that two leechers pursue different goals. For example, while some of them want to download the file as fast as possible, disregarding drawbacks, other leechers may care for the amount uploaded. In the idealistic setting, we only look at the initial seeding phase, where only one seeder exists but lots of leechers want to download the file. Additionally, we exclude the case where other leechers join during the initial seeding phase. Presumably, the bottleneck in this particular model lies in the upload capacity of the seeder. Nowadays, for the majority of users, the downstream is much bigger than the upstream. Our protocol will work for all cases where the downstream is at least as big as the upstream.

For simplicity, in our model the time is represented by rounds. Also, we assume that all players have the same upload bandwidth and are able to upload $1/N$ of the file per round. Finally, let each participant have a key pair (pk, sk) with a publicly known pk, which is later on used for digital signatures. For now, we do not care how these key pairs are established. For example, they could be provided by a public key infrastructure.

3.1.1 Variables

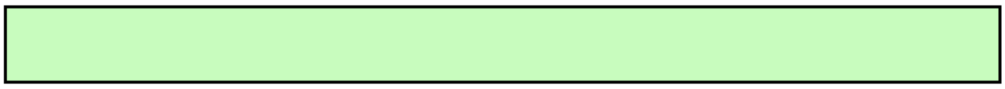
Our *file* f which consists of $B \cdot N$ bytes is split into N *pieces* $f_1, \ldots, f_N \in \{0,1\}^{8B}$. Also, N different *leechers* $L_1, \ldots, L_N$ are from the beginning in the system. All of them try to download the file from the *seeder* S. As already pointed out, we measure time in *rounds* which are referred to by the integer t. We want that every peer is able to upload exactly B bytes per round, and every peer is able to download at least B bytes in one round.

To be able to reason about the preferences of peers, we introduce the *uploaded capacity*, which is denoted by $C_i^l(t)$ for leechers and $C^s(t)$ for the seeder. Obviously, the overall uploaded capacity depends on the time. Furthermore, we introduce a variable $a(t)$ which is used to express how much of the file is available in the whole system. The larger $a(t)$ is, the healthier the system is. The *availability* is computed as follows:

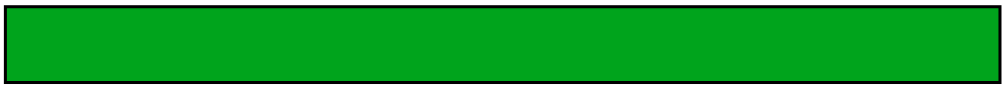
$$a(t) = \min\{a_1(t), \dots, a_N(t)\} + \sum_{k \in A} \frac{1}{N},$$

where $a_i(t)$ is the number of peers holding piece f_i at time t and A is the set of all pieces that are held by strictly more peers than $\min\{a_1(t), \dots, a_N(t)\}$. Note that we are including the seeder if he is still in the system. A few examples:

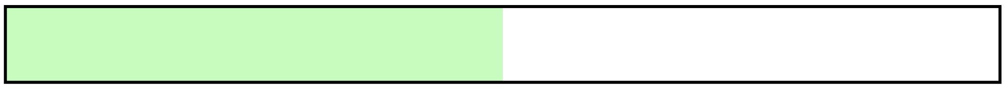
- Only the seeder has pieces: $a(t) = 1$.



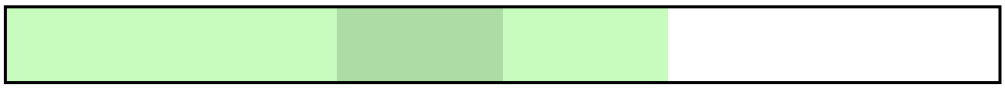
- All N leechers finished downloading: $a(t) = N + 1$.



- One leecher has the first half of the file, everyone else left: $a(t) = 0.5$.



- One leecher has the first half of the file, another leecher has the second third of the file, everyone else left: $a(t) = \frac{2}{3}$.



Finally, we want to introduce the *completion* $\eta_i(t) \in \{0, 1\}$ of leechers which indicates whether a leecher completed the download.

3.1.2 Initial Seeder's Utility

The initial seeder is the person who wants to share a file. His utility therefore differs strongly from subsequent seeders or from leechers. In any case he will upload the whole file at least once, because otherwise, he would not share it in the first place. Furthermore, he wants as many people as possible to get the file. Depending on the preferences of the seeder, he might also want to minimize one of the following things:

- The time needed to distribute the file to the people.
- The overall upload capacity.

We express these preferences by a general utility function, which still leaves a lot of possibilities.

Definition 8 (Seeder's Utility). *Let T_{fin} be the number of rounds a system runs. We say that u^s is a* utility function for a seeder *if for two outcomes o, o' of a file sharing game it holds that $u^s(o) > u^s(o')$ if and only if*

- *$a(T_{fin})$ for outcome o is bigger than for o' or*
- *$C^s(T_{fin})$ for outcome o is smaller than for o' but bigger or equal than N or*
- *T_{fin} for outcome o is smaller than for o'.*

Note that this definition does not prioritize one of the variables. If for example T_{fin} gets smaller but $C^s(T_{fin})$ increases, our general utility function does not specify how the payoff changes.

3.1.3 Leecher's Utility

The main goal of a leecher is to get the file. We assume no altruism here, such that our model gets robust against strongly rational clients.

There are different types of users to consider. For example, there might exist people who want the file as soon as possible, disregarding drawbacks. In this case, their uploaded capacity only secondarily influences the utility function. To be able to express all possible preferences, we keep the model as general as possible.

Definition 9 (Leecher's Utility). *Let $T_{i,fin}$ be the number of rounds leecher L_i stays in the system. We say that u_i^l is a* utility function for a leecher L_i *if for two outcomes o, o' of a file sharing game it holds that $u_i^l(o) > u_i^l(o')$ if and only if*

- *$C_i^l(T_{i,fin})$ for outcome o is smaller than for o' or*
- *$T_{i,fin}$ for outcome o is smaller than for o'.*

Additionally, we require that $u_i^l(o) > u_l^i(o')$, if in outcome o' we have $\eta_i = 0$, meaning that the file was not completely downloaded, whereas in outcome o we have $\eta_i = 1$.

We see that a leecher either tries to decrease his uploaded capacity or the time needed for download.

3.2 Our Protocol

Now, with the utilities being defined, we will construct a protocol. This protocol will provide real incentives for all users, such that there will exist no reason to deviate from the protocol.

3.2.1 The Seeding Algorithm

As the seeder's utility depends on the wasted upload capacity, the seeder wants the leechers to share parts of the file among each other. This increases the upload speed of the whole system, and thus, its throughput. Therefore, what the seeder should *not* do, is to upload the whole file to an arbitrary leecher. Chances are, this leecher will just quit afterwards and leave all other users unsatisfied. To provide an incentive for the leechers to interact, the seeder can send to everyone a piece of the file and then wait for them to exchange all pieces. Note that using this strategy, the seeder will increase the availability $a(t)$ of the file at least by one.

The seeder also uses a small trick. As he is able to upload $1/N$ of the file per round, he will provide $1/N^2$ of the file to every leecher in each round. This way, already after the first round every leecher has something to offer to the others. The exact protocol is described in Figure 3.1.

```
InitSeed$(f, N)$

  1. Split file $f$ into $N^2$ pieces $f_1, \dots, f_{N^2}$.
  2. In every round $j \in \{1, \dots, N\}$ for all $i \in \{1, \dots, N\}$, send $f_{N\cdot(j-1)+i}$ to $L_i$.
  3. Along with each $f_{N\cdot(j-1)+i}$, send a signature on $(i, N \cdot (j-1) + i)$.
```

Figure 3.1: Algorithm for the seeder describing whom to upload pieces to.

3.2.2 Piece Exchange between two Leechers

As all leechers want to minimize the upload capacity, they have to protect themselves from being cheated by another leecher. During the exchange phase, the leechers query each other and get the pieces they do not have. To ensure that one user gets back the capacity he offered to someone else, an exchange between two users is done according to Figure 3.2.

Using the signatures, upon broadcasting a list of all held pieces, other leechers can verify whether someone indeed sent you the shares you pretend to have. Thus a leecher cannot pretend to have other shares than those received from exchanges as he is not able to forge signatures.

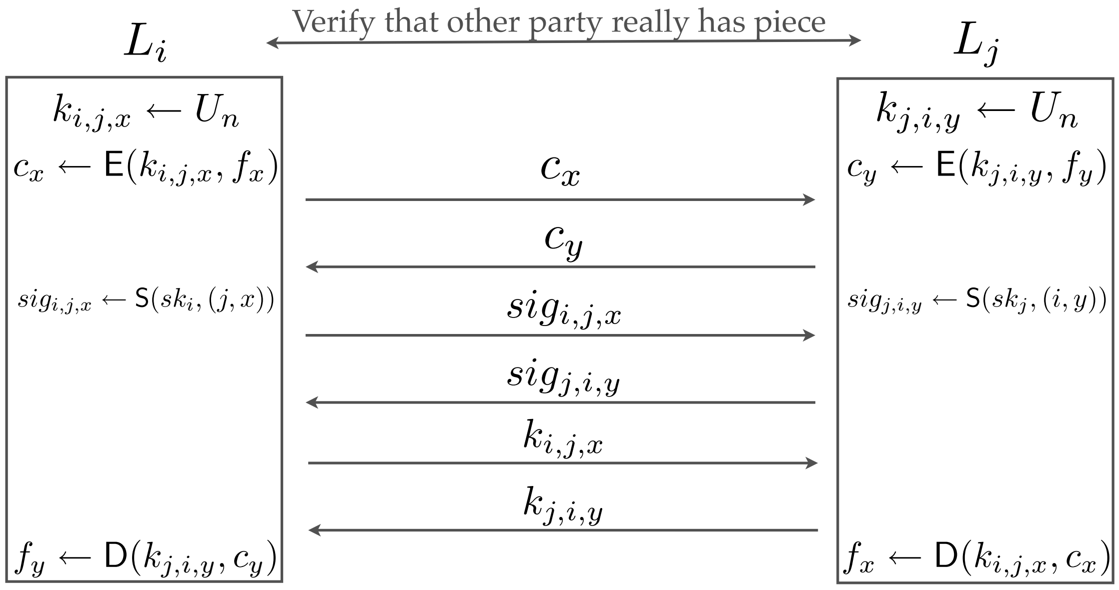
If a user notices that the other party does not follow the protocol, he will *blacklist* the deviating player. This means, from now on he will neither send nor accept messages from him.

Note that a player would not want to deviate in the last step of the exchange. He already wasted his upload to get an encrypted chunk of data and thus decreased his utility. As his payoff only gets bigger if he is able to decrypt the piece he will follow the key exchange without cheating. A user does not care if someone else gets a piece or not, so he has no incentive not to give the key, because the key size is sufficiently small, compared to the piece size.

3.2.3 Piece Selection Algorithm of Leechers

With the seeding algorithm from above, the piece selection algorithm is notably easy. For better understanding, consider the second round of the protocol. Every leecher L_i got his own piece f_i that no other leecher has. Further on, everyone has an upload capacity of $1/N$ of the file, which is N pieces of size $1/N^2$ per round. A leecher is now able to start an exchange with everyone else on the piece he got for the pieces the others received in the last round. While they are all exchanging, every leecher L_i is also getting piece L_{N+i}. Figure 3.3 contains an example for this.

This means, in the third round, every leecher L_i has pieces $f_1, \ldots, f_N, f_{N+i}$, and they can continue exchanging with the newly received pieces. As you can easily observe, this strategy gives a number of $N + 1$ rounds for each leecher in the system until he completes the download. Formally, you can read the algorithm in Figure 3.4.


Verify that other party really has piece
L_i
L_j
$k_{i,j,x} \leftarrow U_n$
$c_x \leftarrow \mathsf{E}(k_{i,j,x}, f_x)$
$sig_{i,j,x} \leftarrow \mathsf{S}(sk_i, (j, x))$
$f_y \leftarrow \mathsf{D}(k_{j,i,y}, c_y)$
c_x
c_y
$sig_{i,j,x}$
$sig_{j,i,y}$
$k_{i,j,x}$
$k_{j,i,y}$
$k_{j,i,y} \leftarrow U_n$
$c_y \leftarrow \mathsf{E}(k_{j,i,y}, f_y)$
$sig_{j,i,y} \leftarrow \mathsf{S}(sk_j, (i, y))$
$f_x \leftarrow \mathsf{D}(k_{i,j,x}, c_x)$

$\mathsf{Exchange}_i(f_x, j, y)$
If any of the following steps fails, the players blacklist each other.

1. Choose a random key $k_{j,x}$ and compute $c_x \leftarrow \mathsf{E}(k_{j,x}, f_x)$.
2. Send c_x to L_j.
3. Wait for L_j to send c_y. This step may take arbitrarily long to complete. (i.e., a player could send the corresponding piece much later than the other player)
4. Compute $\mathsf{S}_{sk}(j, x)$, yielding $sig_{j,x}$, and send $sig_{j,x}$ to L_j.
5. Wait for L_j to send $sig_{i,y}$, and verify that $\mathsf{V}_{pk'}((i, y), sig_{i,y}) = 1$, where pk' is the public key of L_j.
6. Send to L_j the key $k_{j,x}$.
7. Broadcast your new list of pieces and all obtained signatures to all players.

Figure 3.2: Algorithm for two leechers L_i and L_j who want to exchange pieces f_x for f_y. Above you see an illustrating picture whereas below the exact algorithm is given.

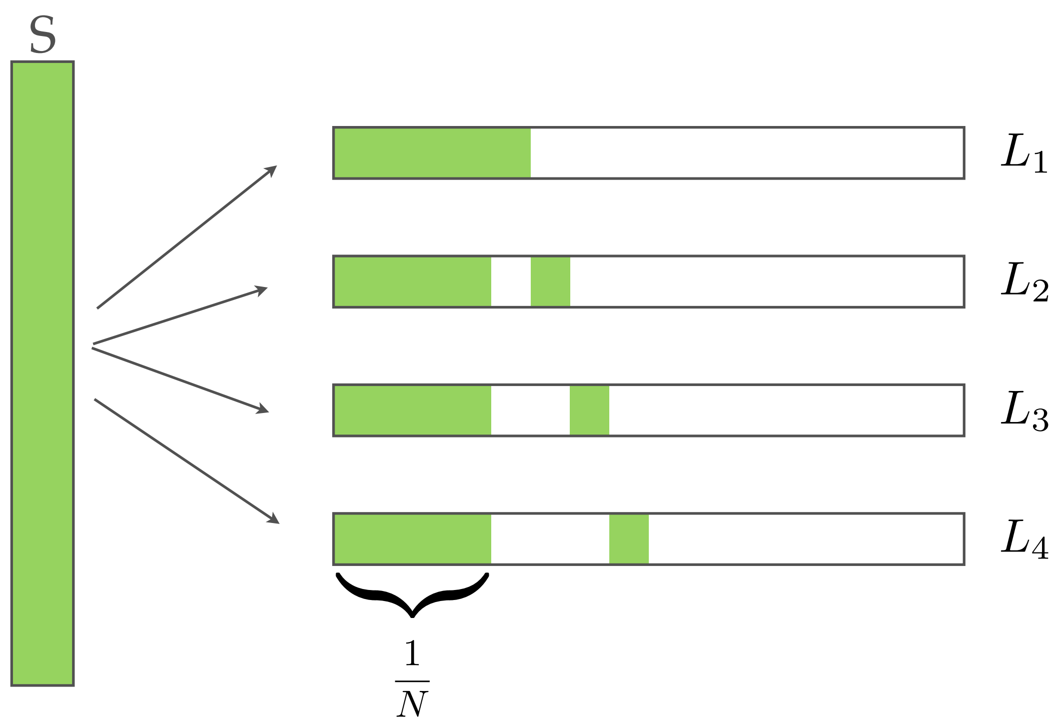


Figure 3.3: After the second round, the four leechers exchanged already $\frac{1}{N}$ of the file and have received an additional $\frac{1}{N^2}$ from the seeder. They will use this for exchanges in the next round.

```
PieceSel$_i$()

    1. If you got piece $f_{N \cdot j+i}$ from the seeder in round $j+1$, start an exchange in
       round $j+2$ with every leecher $L_k$ on the pieces $f_{N \cdot j+i}$ for $f_{N \cdot j+k}$.
```

Figure 3.4: Piece selection algorithm for a leecher L_i.

3.3 Proofs

First of all, we want to show that nobody will deviate during the Exchange protocol. Then, building upon this, we prove that our given strategies achieve the best results possible.

Lemma 10 (Exchange Phase)**.** *No leecher will deviate during an* Exchange *iteration.*

Proof. Assume that player L_i plays a deviating strategy M_i' instead of M_i. We show that a deviation will not increase his utility. Assume that player i, offering the piece f_x, deviates when he interacts with player j who in turn offers f_y. We examine the different possibilities in a case distinction.

1. L_i uploads $d \neq \mathsf{E}(k_{j,x}, f_x)$. Then again, we distinguish between two cases:
   (a) L_i did not receive f_x so far.
   $\Rightarrow$ L_i can provide a signature $sig_{i,x}$ from some player h only with negligible probability. Therefore, this case happens only with the same negligible probability.
   (b) L_i did receive f_x.
   $\Rightarrow$ L_i does receive f_y. However, L_i is blacklisted by L_j at the end of the Exchange protocol. We therefore obtain $u_i^l(M_i', \boldsymbol{M}_{-i}) \leq u_i^l(\boldsymbol{M})$.
2. L_i delays sending $\mathsf{E}(k_{j,x}, f_x)$ infinitely.
   $\Rightarrow$ L_i does receive $\mathsf{E}(k_{i,y}, f_y)$, but he will not get $k_{i,y}$. Therefore, player L_i is only able to decrypt f_y with negligible probability $\epsilon(|k|)$. Otherwise, he does not gain any information about f_y. Therefore, we obtain $u_i^l(M_i', \boldsymbol{M}_{-i}) + \epsilon(|k|) \leq u_i^l(\boldsymbol{M})$.
3. L_i does not send a correct signature on (j, x).
   $\Rightarrow$ L_i will not obtain a key on $\mathsf{E}(k_{i,y}, f_y)$, and we have $u_i^l(M_i', \boldsymbol{M}_{-i}) + \epsilon(|k|) \leq u_i^l(\boldsymbol{M})$.
4. L_i does not send $k_{j,x}$ to L_j.
   $\Rightarrow$ L_j will blacklist L_i, but L_i gets f_y. We therefore get $u_i^l(M_i', \boldsymbol{M}_{-i}) \leq u_i^l(\boldsymbol{M})$.

As one can easily observe, each case either happens only with negligible probability or yields

$$u_i^l(M_i', \boldsymbol{M}_{-i}) + \epsilon(|k|) \leq u_i^l(\boldsymbol{M}).$$

Therefore, no leecher has an incentive to deviate. □

Lemma 11 (Correctness of the Piece Selection Algorithm)**.** *When all N leechers in the system follow the* piece selection algorithm*, every leecher will complete the download after exactly $N+1$ rounds and upload exactly $N^2 - N$ pieces.*

Proof. By the seeding algorithm, every leecher L_i receives in round $j+1$ the piece $f_{N\cdot j+i}$. Note that we are counting both the rounds and the pieces from 1. Assume now that a leecher L_i already has pieces $f_1, \ldots, f_{N\cdot j}, f_{N\cdot j+i}$. Then, by construction, every other leecher L_k holds a piece that L_i does not own yet, namely $f_{N\cdot j+k}$. Because a leecher is able to upload $1/N$ of the file, he can upload N pieces of the file in a round. This is sufficient to start an exchange with every other of the $N-1$ leechers, thus receiving in the next round the pieces $f_{N\cdot(j+1)+i}$ (from the seeder) as well as $f_{N\cdot j+1}, \ldots, f_{N\cdot j+i-1}, f_{N\cdot j+i+1}, \ldots, f_{N\cdot(j+1)}$, which are exactly N pieces. He is able to receive those, because his download capacity is at least $1/N$ of the file by our assumption.

This means, in round $N+1$, a leecher holds the pieces $f_1, \ldots, f_{N^2}$, which is the whole file. In addition, he uploaded in every round but the first $N-1$ pieces, which in total makes $N(N-1) = N^2 - N$ uploaded pieces. □

To later on show that we have a Nash equilibrium, we need one additional Lemma that states that our above results are the best possible. Then, neither leechers nor seeder will have an incentive to deviate.

Lemma 12 (Bound on Time)**.** *It is not possible for any protocol to deliver the whole file to all $N > 1$ leechers in N rounds.*

Proof. Without loss of generality, assume the file was split into M pieces and the pieces are uploaded in the order of their index. By our definition of rounds, N rounds are needed to upload the whole file exactly once. Then, the piece f_M needs to be uploaded to a leecher in round N. However, after this round, only this leecher holds piece f_M, so at least one additional round is needed to distribute that piece among everyone else. □

Lemma 13 (Leecher's Protocol)**.** *No leecher will deviate in his protocol.*

Proof. By Lemma 11, every leecher needs at most $N+1$ rounds and needs to upload $N^2 - N$ pieces. Clearly, by Lemma 10, an upload of $N^2 - N$ pieces is the best a leecher can achieve, as the seeder gives away N pieces to him and for the remaining $N^2 - N$ pieces he wishes to download he has to offer the same amount for.

In addition, he can also not get better than $N+1$ rounds, because if everyone else sticks to the protocol there will exist pieces that are only available in the $N+1$st round for him download.

Therefore, as a leecher is not able to decrease neither C_i^l nor $T_{i,fin}$, he will not deviate. □

Lemma 14 (Seeder's Protocol)**.** *The seeder will not deviate from his protocol.*

Proof. As the seeder initially wants to share the file, he will, no matter what happens, at least upload the whole file one single time. Our idealized protocol ensures that he only needs to upload this file once and then the leechers will distribute the pieces among themselves. This means, the seeder is not able to decrease C^s. By Lemma 12 there is also no possibility to decrease the time needed T_{fin}. Of course, if all leechers get the file after $N+1$ rounds, you also cannot increase the availability $a(T_{fin}) = N+1$. Therefore, the seeder has no incentive to deviate. □

Theorem 15 (The Protocol is a Computational Nash Equilibrium)**.** *The protocol, given by the routines* InitSeed*,* Exchange *and* PieceSel*, is a* computational Nash equilibrium*.*

Proof. This follows directly by Lemma 13 which states that no leecher will deviate and by Lemma 14 which explains why the seeder will not deviate. Therefore, no computationally bounded player will deviate. □

4 Realistic Setting

4.1 Motivation

In the idealistic setting, we had many constraints both on the system and on the clients. Those assumptions clearly will never hold in reality. However, it was a good way to start and to establish a Nash equilibrium.

Now, we move to a new model, where the assumptions are close to reality. Again, we construct the protocol as well as the algorithms for the peers such that in the end we obtain a Nash equilibrium.

It is a question of preference whether the Nash equilibrium is good enough for "real world" applications. Clearly, an equilibrium which is robust against coalitions would be more desirable, however, this would also increase the metadata needed for the system. The protocol we offer allows some good mitigation techniques against coalitions. In the end, players in coalitions will be able to increase their utility by not following the protocol so little that it is not worth the effort to code a deviating client.

4.2 The Model

Following the BitTorrent convention, we call a player in the file sharing game a *peer*. A peer wants to download the *file* f, which is divided into *pieces* $f_1, \ldots, f_N$, each of size B. We require that $B \gg |k|$, where $|k|$ is the length of the keys for the encryptions we are going to use. All participants hold a publicly known *hash function* h as well as the values $h_1 = h(f_1), \ldots, h_N = h(f_N)$, to be able to verify the integrity of received pieces during the protocol.

A peer is either a *seeder* $S_i, i \in S$ from the set of all seeders, if he already holds the whole file and just stays in the system because he wants to share the file, or a *leecher* $L_i, i \in L$, if he still wants to download parts of the file. In addition, we require a trusted party called *tracker* whose IP is known, and who holds a signing key pair (pk, sk).

For convenience, we split the set of peers into two groups, the seeders and the leechers. Then, every seeder i has its individual *upload speed*, denoted by up_i^s. Note that a seeder does not download anything except for metadata, which is why we do not relate to its download speed. Every leecher also has individual upload and *download rates* up_i^l and $down_i^l$. When a leecher L_i is done downloading, we say that $T_{i,fin}$ is the overall time he spent.

Every peer has a *local neighborhood* of H people. To those people he has an open connection and with those he can exchange. We also assume, that each peer is only is able to maintain one connection to the system. For simplicity, each connecting or leaving has to take place in between two *rounds* which last T seconds. We also assume that nobody is able to forge messages which seem to come from another peer than himself.

Further on, we assume that the overall seeding capacity does not exceed the overall upload capacity of the leechers too much. As long as we have

$$\frac{\sum_k up_k^l}{\sum_k up_k^s} \geq \frac{1}{2}, \tag{4.2.1}$$

we can guarantee that no leecher has an incentive to deviate. We believe that this is realistic, because usually only a few people are altruistic while a lot of them leave after having downloaded the file.

Finally, we want to exclude some pathological cases where it is in general difficult to give assertions. First, we require that there is always at least one seeder in the system. Second, we want that for all leechers, every piece is immediately accessible. This basically means, that the availability of the file is big enough and that even the rarest pieces are distributed fast, be it through other leechers or through seeders. Also, we want that the system gives every leecher who follows the protocol a download speed of at least $5\frac{B}{T}$. The last two assumptions may seem strong, but they are later on just used to exclude the case where a leecher is *stuck*, i.e. that he has no pieces to offer to other users. These two assumptions are more backed up by practice than by theory.

4.2.1 Utility Functions

Leecher's Payoff

In contrast to the small model, we need to strengthen the assumptions on the utility functions. As our system provides a few pieces to newcomers, we have to ensure that this mechanism is not misused. I.e., a leecher should not always reconnect (*whitewash*) and gain advantage by claiming that he is new to the system. However, BitTorrent is used because it offers fast download rates. This means, if your download stops, we can safely assume that the payoff for a leecher will decrease rapidly, disregarding how much he uploads. We therefore require that a download speed of 0 is considered to be too low by the leechers. This means, that the utility of a leecher increases if the download speed increases above 0, even if the leecher needs to upload (more).

Other than that, we continue to stick to the utility of the small model, which means that a leecher tries to minimize the amount of uploaded data and the time needed for downloading the file.

Definition 16 (Leecher's Utility). *We say that u_i^l is a* utility function for leecher L_i *if the following two conditions hold. For two outcomes o, o' it holds that $u_i(o) \geq u_i(o')$ if and only if one of the following cases holds:*

- *L_i holds the file in o but not in o'* or
- *$\int_0^{T_{i,fin}} up_i^l(t)\,dt$ in o is smaller than in o' (which means that L_i uploaded less).* or
- *$T_{i,fin}$ in o is smaller than in o'.*

Second, let o, o' be two outcomes which have the same transcript up to some point T_{diff}. We want that $u_i^l(o) \geq u_i^l(o')$ if in o' we have $down_i^l(T_{diff}) = 0$ and $T_{i,fin}$ in o is smaller than in o'.

The second point basically just means, if a player is able to avoid having a download speed of 0, he will do so. Note that the first condition does *not* contain the second one: In the latter, we say that the utility decreases *in any case* if your download speed drops to 0, whereas the first condition only talks about one variable while the others are fixed.

Seeder's Payoff

The seeder still wants to maximize the efficiency of his upstream. Here, we reformulate the seeder's utility in contrast to the small model.

Definition 17 (Seeder's Utility)**.** *We say that u_i^s is a* utility function for a seeder S_i *if for two outcomes o, o' it holds that $u_i(o) \geq u_i(o')$ if and only if*

- *$\frac{1}{|L|}\sum_{i \in L} T_{i,fin}$, which is the average time to complete a download, in o is smaller than in o'* or
- *$\int_0^{T_{i,fin}} up_i^s(t)\,dt$ in o is smaller than in o'.*

4.3 The Protocol

In this section, we will present the protocol that will lead to a Nash equilibrium.

4.3.1 Tracker's Protocol

The job of the tracker is to keep track of everyone in the system, i.e. to know all valid IPs and to know who has what parts of the file. When a new peer i joins, he asks the tracker to assign him a set of connections, and the tracker replies giving him a random subset of size H of all peers which are currently in the system. This random subset will become the local neighborhood of L_i. We assume the ID i to be given. In reality, this will be the IP of the newcomer. In the next subsections we will look in detail at the responsibilities of the tracker.

Connect Protocol

Firstly, the tracker is responsible for the forced wait of newcomers. Intuitively, a forced wait is a simple defense mechanism against whitewashing. If you join the system once, then a small wait will not hurt you. However, if you try to whitewash and therefore rejoin all the time, then the waits will accumulate and your payoff gets smaller. We also implement a mechanism such that people can later on prove that they already downloaded some pieces.

The tracker holds the variables $A_i^m, m \in \{1, \dots, N\}$, where $A_i^m = 1$ if leecher L_i has piece f_m, and $A_i^m = 0$ otherwise. This is basically just an array describing what pieces every leecher has.

```
TrackerConnect($i$)

  1. Compute $\mathsf{S}_{sk}(i, T_c)$, where $T_c$ is the current time, yielding $sig_t$, and send to $L_i$ the message "TIME $(T_c, sig_t)$".

  2. If the leecher claims to have pieces by sending the message $(a_1, \dots, a_N, id, sig_p)$, verify that $\mathsf{V}_{pk}((a_1, \dots, a_N, id), sig_p) = 1$ and that you have $p := (a_1, \dots, a_N, id)$ stored. Remove $p$ from your local storage and then set $\forall m \in \{1, \dots, N\} : A_i^m := a_m$.

  3. Send to $L_i$ a random subset of size $H$ of all known IPs.

  4. Notify a seeder by sending him "NEWPEER $(i, T_c)$".
```

Figure 4.1: Algorithm executed by the tracker when a new peer joins

The tracker's protocol for a joining leecher is described in Figure 4.1. A more detailed protocol is located in the appendix, in Figure A.1. The waiting time is not enforced by the tracker, but by the seeder who is supposed to wait a fixed amount of seconds before he starts uploading to the newcomer. He uses T_c to determine when he can start uploading. The tracker also signs the timestamp T_c such that a leecher cannot fake it when connecting to other parties. Similarly, we implement a method PeerLeave(i) in Figure 4.2 which is invoked when a leecher leaves before he downloaded the full file, such that he can later rejoin and is still able to use the pieces he already downloaded. Note that the tracker allows to reconnect, but each reconnect is bound to an *id*. More specifically, a leecher cannot "fork" his state by reconnecting twice using the same reconnect *id*. This especially is important against coalitions which we will discuss in Chapter 5.

Reward Protocol

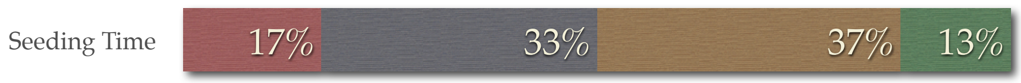
The system is designed to work in rounds of T seconds. During each such round, the tracker, seeders and leechers follow the protocols shown below. After T seconds, the

```
PeerLeave($i$)
    1. Select a random number $id$ and let $p := (A_i^1, \ldots, A_i^N, id)$. Store $p$ locally.
    2. Compute $\mathsf{S}_{sk}(p)$, yielding $sig_p$.
    3. Send to $L_i$ the message "LEAVE $(id, sig_p)$" and disconnect from him.
```

Figure 4.2: Tracker's algorithm invoked when a leecher wants to leave and rejoin later

tracker will be responsible for giving out the rewards for people following the protocol. The idea behind the system is to reward leechers who are exchanging. This way, we can provide strong incentives to exchange, because if one does not exchange, one is not able to download both from seeders and from other leechers.

For illustration, assume the following setting. Leecher L_4 had 5 exchanges in the last T seconds. Similarly, L_{25} had 10 exchanges, L_{10} achieved 11 of them and L_{13} just made 4. Then, the seeding time of all seeders is split according to how much percent of all exchanges a leecher had. In our example, it looks like this:



For example, because L_{10} had more than one third of all exchanges, he also deserves 37% of the overall seeding capacity.

The protocol in Figure 4.3 is used to determine whether an exchange between two leechers actually happened. Here, the tracker internally holds a variable X_i for every leecher L_i indicating how many exchanges he made during the last round of T seconds. As you can see, the tracker uses the CheckExchange call also to update the completion of the leechers. But as leechers can also get pieces from seeders, every time a seeder sends the tracker the message "SENT (i, x)", the tracker updates $A_i^x := 1$, too.

In Figure 4.4, the algorithm is shown which the tracker uses to reward the leechers according to the number of their exchanges. Note that a leecher cannot have more than half of the overall exchanges, because this would mean that for some exchanges he did not have a partner.

4.3.2 Seeder's Protocol

The seeder's protocol becomes very simple in this case, as the tracker does all the computation which is needed for establishing the rewards. Therefore, the seeder simply does what he is told by the tracker. There are just a few things he needs to take care of:

```
CheckExchange()
If any of these steps fails, do nothing.

  1. Upon receiving a message "EXCHANGED $(j, x, y)$" from $L_i$, check if both $A_i^x = 1$ and $A_j^y = 1$. If the check passes, ask $L_j$ whether the exchange really happened by sending to him "ACKNOWLEDGE $(i, x, y)$". Otherwise, send to $L_i$ the error and cancel.

  2. If $L_j$ replies with "OK $(i, x, y)$", meaning that he acknowledged the exchange, increase $X_i$ by one and set $A_i^y := 1$.

  3. Send to $L_i$ the message "OK $(j, x, y)$".
```

Figure 4.3: The check exchange algorithm implemented by the tracker

- As soon as the seeder joins the system, (i.e., he just became a seeder or he connected to the system), he notifies the tracker of his upload speed.
- When he is told by the tracker to upload to a leecher, the seeder follows the order, because the tracker is trusted. The leecher requests pieces to be uploaded.
- When a new leecher joins the system and the seeder is notified by the tracker, the seeder lets the newcomer request 5 pieces which he then uploads after waiting T seconds.
- Every time a seeder uploaded piece f_x to L_i, he sends the message "SENT (i, x)" to the tracker such that he is able to track the owned pieces of the leechers.

We give the detailed connect protocol in the appendix in Figure A.2.

4.3.3 Leecher's Protocol

The Leecher's protocol is the most important protocol, because whereas seeders and the tracker can be regarded as at least partially honest, a leecher would try to do anything to decrease his download time or his overall upload.

Connecting to Other Parties

When a leecher connects to a new party, they need to exchange some information first. For the sake of readability, we just briefly mention what needs to be sent.

- *Connecting to the tracker.* If you are new, just wait for the tracker to give you a neighborhood and a signature on the time (T_p, sig_t). If you are rejoining, additionally provide the tracker a signature on the held pieces.

RewardLeechers() (called every T seconds)

1. Calculate for every leecher the percentage of the upload speed he deserves. $r_i := \min\left\{\frac{X_i}{\sum_k X_k}, \frac{1}{2}\right\}$
2. Distribute the upload speed of every seeder such that for the overall average download speed of every leecher L_i holds $down^l_i = r_i(\sum_k up^s_k) + \theta_i$, where θ_i is the download speed the leecher L_i acquires from exchanges with other leechers.
3. For all i, set $X_i := 0$.

Figure 4.4: The tracker's algorithm to reward leechers according to the number of their exchanges

- *Connecting to a seeder.* Request 5 pieces from the seeder and wait until he uploads them to you.
- *Connecting to a leecher.* Do not connect until you have $T_p + T < T_c$, where T_c is the current time and T_p is the timestamp you received from the tracker. Send $(a_1, \ldots, a_N)$ to the other leecher, where $a_m = 1$ if you hold piece f_m. Wait for the other party also to send their availability as well. Also, exchange your signed timestamps, receiving (T'_p, sig'_t). Verify the signature. If it holds $T_c < T'_p + T$, this means that the leecher just joined. In this case, ignore him until $T_c > T_p + T$.

Note that a leecher always needs to keep track of the availability in his neighborhood. He does this, as in the original BitTorrent, by observing the "HAVE x" messages which leechers send when they acquire a new piece. Again, for better readability, we specify the exact connect protocols in the appendix in Figure A.3.

Uploading

To fix the problems with the strategic client that was proposed by [PIA+07], we change the uploading strategy of the leechers. As no leecher wants to upload more than he receives, we propose a simple rule. Upload to everyone exactly as fast as he uploads to you. Theoretically, this obviously gives you an equilibrium, because no matter how one changes his uploading strategy, one will always download as fast as upload, possibly even slower. The equilibrium here is therefore intuitive.

In practice, this is enforced anyway, because in the piece exchange (which we will discuss in the next paragraph) both parties wait until they have received their encryptions. Practically, this means that both parties upload with the same speed.

Piece Exchange

The piece exchange in the realistic setting works now somewhat different than in the small model. The basic idea remains the same, namely that encryptions are used to ensure that nobody cheats, and that $|k| \ll B$, where $|k|$ is the key length and B is the piece size.

The difference is that now, leechers want their exchanges to be counted and rewarded. The simple idea behind this is to prevent people from free riding. Again, as one can easily observe in Figure 4.5, if one does not upload, one also does not get any reward. We therefore used the altruism given by the seeders cleverly to provide additional incentives for exchanging and thus for participation in the protocol. The exchange protocol can be found schematically in Figure 4.5 and in detail in the appendix in Figure A.4.

Please note that this protocol is presented just from the viewpoint of one leecher. Of course, the other leecher has to do the same steps, but this would make the diagram too crowded.

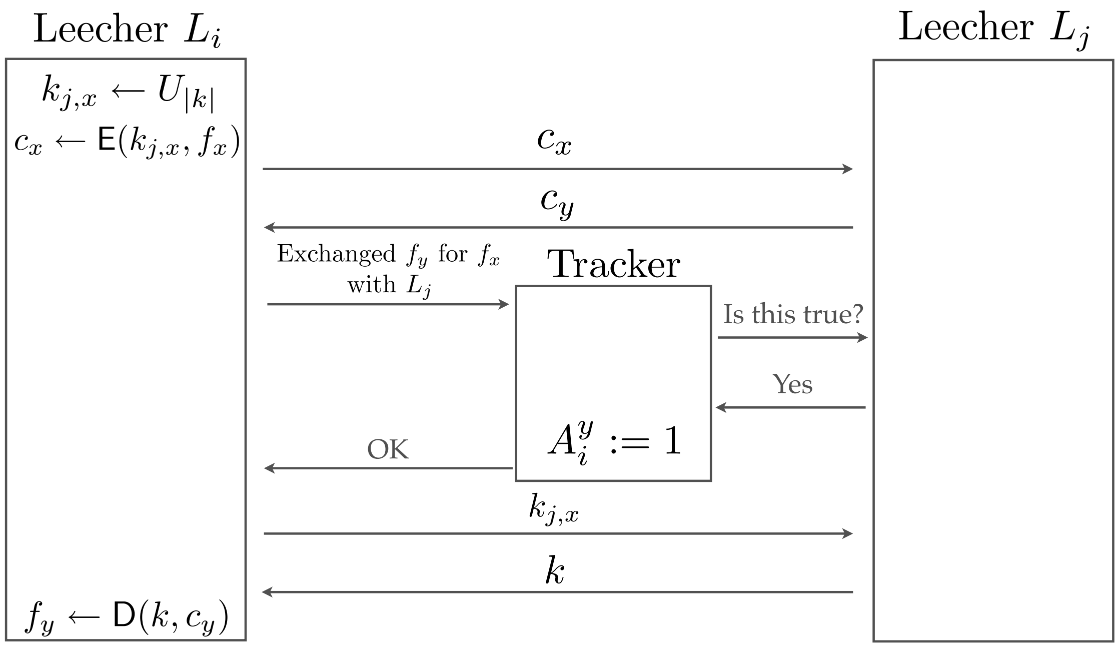

Leecher L_i
Leecher L_j
$k_{j,x} \leftarrow U_{|k|}$
$c_x \leftarrow \mathsf{E}(k_{j,x}, f_x)$
c_x
c_y
Exchanged f_y for f_x with L_j
Tracker
Is this true?
Yes
$A_i^y := 1$
OK
$k_{j,x}$
k
$f_y \leftarrow \mathsf{D}(k, c_y)$

Figure 4.5: Piece exchange protocol for leecher L_i

4.4 Proofs

To show that our protocol constitutes a computational Nash equilibrium, we prove that no party has an incentive to deviate. This is only necessary for seeders and leechers, because the tracker is trusted.

4.4.1 Leecher's Protocol

A leecher has several possibilities of deviating, because he has a rather complex strategy. We will now prove step by step that no deviation increases his payoff if all other participants follow the protocol. First, we handle all simple deviations that are possible and where you immediately can tell that this will not increase the payoff. Afterwards, we discuss the single more involved case, namely when a leecher does not acknowledge another leecher's reward. From this, it follows that no leecher has an incentive to deviate.

Lemma 18 (Simple Deviations)**.** *Let $|k|$ be the key size of all keys chosen in the protocol. Let M_i' be the strategy for leecher L_i who plays at some point one of the following deviations:*

1. *It does not announce the correct timestamp he got from the tracker.*
2. *It connects to leechers before the penalty wait of T seconds is over.*
3. *It accepts connections from leechers whose penalty wait is not over.*
4. *It announces to hold more pieces than it actually does.*
5. *It announces to hold fewer pieces than it actually does.*
6. *In the exchange phase, it does send $d \neq \mathsf{E}(k, f_x)$ upon being requested to exchange piece f_x.*
7. *It requests an invalid reward from the tracker.*
8. *In the exchange phase, it doesn't send the key in the end.*
9. *It reconnects at time T_D on purpose while it is participating in the protocol.*

Then, we can construct a strategy M_i'' which does not deviate precisely in that particular point, and it holds

$$u_i(|k|, M_i', \boldsymbol{M}_{-i}) - u_i(|k|, M_i'', \boldsymbol{M}_{-i}) \leq \epsilon(|k|).$$

This means, a leecher cannot increase his payoff more than negligibly by deviating in a way listed above.

Proof. Assume that the deviating leecher L_i plays a deviating strategy M_i' as described above. Using a big case distinction, we show that neither of the cases does increase his utility.

1. *M_i' does not announce the correct timestamp he got from the tracker.*

   Because L_i always has to provide a valid signature from the tracker on the timestamp, this deviation will be undetected only with a negligible probability $\epsilon(|k|)$. If it is detected, he is blacklisted by the other party, which does not increase his utility. Therefore, we obtain that

   $$u_i(|k|, M_i', \boldsymbol{M}_{-i}) - u_i(|k|, M_i'', \boldsymbol{M}_{-i}) \leq \epsilon(|k|).$$

2. *M_i' connects to leechers before the penalty wait of T seconds is over.*

   Because other leechers will not reply to L_i unless he provides a timestamp $T_p < T_c - T$, and this is not possible to fake as we have shown above, this again will only help him with negligible probability. Thus, we have

   $$u_i(|k|, M_i', \boldsymbol{M}_{-i}) - u_i(|k|, M_i'', \boldsymbol{M}_{-i}) \leq \epsilon(|k|).$$

3. *M_i' accepts connections from leechers whose penalty wait is not over.*

   Because those leechers will not connect to him, this does not change the outcome of the game. Therefore, we obtain that

   $$u_i(|k|, M_i', \boldsymbol{M}_{-i}) = u_i(|k|, M_i'', \boldsymbol{M}_{-i}).$$

4. *M_i' announces to hold more pieces than it actually does.*

   There are two possible cases. Either L_i tries to convince the tracker upon connecting that he holds more pieces, or he has sent wrong "HAVE x" messages. In the first case, he would have to provide a valid signature from the tracker, which is only possible with negligible probability $\epsilon(|k|)$. In the second case, this will not affect anything until some other leecher requests a piece f_x from him that he does not have yet. Then, in the exchange step, the tracker will send the other leecher the message that he does not have f_x. The exchange will be canceled and L_i will be blacklisted by the other leecher. This does not increase his utility, so we have

   $$u_i(|k|, M_i', \boldsymbol{M}_{-i}) - u_i(|k|, M_i'', \boldsymbol{M}_{-i}) \leq \epsilon(|k|).$$

5. *M_i' announces to hold fewer pieces than it actually does.*

   This does not increase his utility, because the amount of pieces is used by other leechers to determine whether they can exchange with L_i. Fewer parties are willing to exchange with him if he claims to have fewer pieces. So we obtain

   $$u_i(|k|, M_i', \boldsymbol{M}_{-i}) \leq u_i(|k|, M_i'', \boldsymbol{M}_{-i}).$$

6. *In the exchange phase, M_i' sends $d \neq \mathsf{E}(k, f_x)$ upon being requested to exchange piece f_x.*

Assume L_i does not hold piece f_x. Because other leechers only request what he announced with "$\mathsf{HAVE}\ x$", this means he wrongly announced that he has too many pieces. This case is already discussed above. Therefore, L_i has the piece the other party requested and the other party expects a message of the length $|\mathsf{E}_k(f_x)|$. But then, L_i can also send the encryption of the requested piece, because sending a wrong encryption still requires him to upload exactly as much as uploading the right encryption. Thus, this deviation leaves him with

$$u_i(|k|, M_i', \boldsymbol{M}_{-i}) - u_i(|k|, M_i'', \boldsymbol{M}_{-i}) \leq \epsilon(|k|).$$

7. *M_i' requests an invalid reward from the tracker.*

   By assumption, M_i' cannot forge a message from a different player than himself. However, his request will not be acknowledged by any other party, as they are sticking to the protocol. Therefore, this does not increase his utility:

$$u_i(|k|, M_i', \boldsymbol{M}_{-i}) = u_i(|k|, M_i'', \boldsymbol{M}_{-i}).$$

8. *In the exchange phase, M_i' doesn't send the key in the end.*

   Again, by assumption we said that the key size does not increase the uploaded amount. However, L_i is blacklisted by the other party afterwards. Therefore, we have

$$u_i(|k|, M_i', \boldsymbol{M}_{-i}) \leq u_i(|k|, M_i'', \boldsymbol{M}_{-i}).$$

9. *M_i' reconnects at time T_D on purpose.*

   We are looking at two different outcomes. In o, the leecher did not rejoin at T_D while in o', he reconnected at time T_D. Then, in o', L_i has $down_i^l(T_D) = 0$. Additionally, he starts receiving new pieces after a penalty wait of T seconds. We split the leecher's download speed $down_i^l$ in two components. Let $down_{i,E}^l$ be the 5 pieces extra received from seeders upon joining the system and $down_{i,0}^l$ be the speed received through normal exchanges and rewards. Assume that the leecher already holds Y_i pieces of the file.

$$\begin{aligned}
B \cdot (N - Y_i) &= \int_{T_D}^{T_{i,fin}} down_i^l(t)\, dt \\
&= \underbrace{\int_{T_D}^{T_D+T} down_i^l(t)\, dt}_{=0} + \int_{T_D+T}^{T_{i,fin}} down_i^l(t)\, dt \\
&= \underbrace{\int_{T_D+T}^{T_{i,fin}} down_{i,E}^l(t)\, dt}_{\leq 5B} + \int_{T_D+T}^{T_{i,fin}} down_{i,0}^l(t)\, dt
\end{aligned}$$

Because we assumed that every piece is accessible to the user, it makes no difference whether a user is downloading from a seeder or from a leecher. Therefore, he will only benefit from this deviation if he is able to decrease $T_{i,fin}$ in o'. However, in o, the leecher always maintains a download speed above $5\frac{B}{T}$ by assumption. More precisely, in o the following holds:

$$\begin{aligned}\int_{T_D}^{T_D+T} down_i^l(t)\,dt &\geq \int_{T_D}^{T_D+T} \frac{5B}{T}\,dt \\ &\geq 5B.\end{aligned}$$

It follows that in o, $T_{i,fin}$ is smaller than in o'. Further on, the transcripts of o and o' are the same up to time T_D, where in o', the leecher has a download speed of $down_i^l(T_D) = 0$. This exactly matches the second condition in the utility function of a leecher, resulting that $u(o) \geq u(o')$. Therefore, the deviation did not increase the utility:

$$u_i(|k|, M_i', \boldsymbol{M}_{-i}) \leq u_i(|k|, M_i'', \boldsymbol{M}_{-i}).$$

This concludes our proof. □

Now we only mentioned some cases. Apart from the most interesting deviation, intuitively it is now clear that by cheating you cannot increase your utility. However, one would need to examine all other cases which are not discussed so far, e.g. that a leecher does not ask the tracker whether another party has a piece before exchanging. Or, a leecher simply sends random data. However, these deviations really are pathologic. To save time and space, we therefore will omit them.

Lemma 19 (Deviations)**.** *Let $|k|$ be the key size of all keys chosen in the protocol. Let M_i' be the strategy for leecher L_i that plays at some point any deviation besides not acknowledging another's exchange in the* Exchange *protocol. Then, we have*

$$u_i(|k|, M_i', \boldsymbol{M}_{-i}) - u_i(|k|, M_i'', \boldsymbol{M}_{-i}) \leq \epsilon(|k|).$$

Proof. Omitted. □

Theorem 20 (Leecher's Protocol)**.** *Let $|k|$ be the key size of all keys chosen in the protocol, and let M_i be the strategy for leecher L_i given by the protocol. Then, we have that for every strategy M_i' it holds that*

$$u_i(|k|, M_i', \boldsymbol{M}_{-i}) - u_i(|k|, \boldsymbol{M}) \leq \epsilon(|k|).$$

In other words, a leecher cannot get any better payoff than using the strategy our protocol describes.

Proof. So far, for every deviation we discussed, we immediately have shown in Lemma 18 and Lemma 19 that this does not increase the deviating leecher's utility. It is left to show that M_i cannot increase its utility by not acknowledging the other party's exchange during an Exchange iteration.

Here, we cannot show that this deviation immediately does not increase the utility of L_i. But we can prove this by looking at one round, which is T seconds long. Assume therefore, L_i plays a deviating strategy M_i' for the whole game. Then, he deviated $Z' > 0$ times. Assume that $Z \leq Z'$ deviations were not to acknowledge the other party's exchange, and in $Z' - Z$ cases he deviated in one of the ways discussed above. Then, we construct a strategy M_i'', which only deviates in Z exchanges by not acknowledging the other's exchanges. It holds that $u_i(|k|, M_i'', \boldsymbol{M}_{-i}) \geq u_i(|k|, M_i', \boldsymbol{M}_{-i})$ because of Lemma 19, which says that those deviations do not increase the utility.

Therefore, if $Z = 0$, we are done, as the constructed strategy M_i'' is the strategy M_i given by the protocol.

For the case where $Z > 0$, we show that $u_i(|k|, \boldsymbol{M}) \geq u_i(|k|, M_i'', \boldsymbol{M}_{-i})$ holds. To reach the phase of acknowledging the other's exchange, L_i first needs to upload the encrypted data. Therefore, the amount of uploaded data stays the same both in the deviated strategy and the protocol's strategy. The only possibility to increase the utility is then to decrease the overall time needed, which is equivalent to increasing the amount of data downloaded in the same time span. We will show that this is not possible.

With M_i, the leecher's download amount corresponding to one round of T seconds equals

$$BX_i + T(\sum_k up_k^s) \cdot \min\left\{\frac{X_i}{\sum_k X_k}, \frac{1}{2}\right\}.$$

Note that in the second summand, we are relating to the download of the next round, because the tracker rewards exchanges always one round later. Using M_i', he gains ZB bytes less from leechers, because those exchanges are cancelled, but he obtains a higher reward from the seeders. Together, this sums up to

$$B(X_i - Z) + T(\sum_k up_k^s) \cdot \min\left\{\frac{X_i}{(\sum_k X_k) - Z}, \frac{1}{2}\right\}.$$

Therefore, his utility will not increase if

$$ZB + T(\sum_k up_k^s)\frac{X_i}{\sum_k X_k} \geq T(\sum_k up_k^s) \cdot \min\left\{\frac{X_i}{(\sum_k X_k) - Z}, \frac{1}{2}\right\}.$$

Note that the tracker does not allow someone to have more than half of all exchanges and therefore caps according to the protocol specified in RewardLeechers the reward at one half of the seeder's upload.

Now, we can simplify the inequality, using how the number of exchanges in T seconds is computed. Namely, we can express the number of our own exchanges X_i by

$$X_i = \frac{T}{B} up_i^l.$$

Similarly, we can say that the number of all exchanges $\sum_k X_k$ equals the overall download speed obtained from exchanges divided by the block size.

$$\sum_k X_k = \frac{T}{B}\sum_k up_k^l$$

Now, we distinguish two cases. In the first case, the leecher increases the benefit from the seeders, but still stays below $\frac{1}{2}$ of the overall capacity of the seeders. In the second case, we examine what happens when he would exceed $\frac{1}{2}$ but is capped by the tracker. Note that those computations strongly rely on the fact that seeding power is at most twice as fast as leeching power.

1. $\frac{X_i}{(\sum_k X_k)-Z} < \frac{1}{2}$. Then we obtain $min\{\frac{X_i}{(\sum_k X_k)-Z}, \frac{1}{2}\} = \frac{X_i}{(\sum_k X_k)-Z}$. Therefore, the utility of a deviating leecher does not increase if the following holds:

$$\begin{aligned}
ZB + T(\sum_k up_k^s)\frac{X_i}{\sum_k X_k} &\geq T(\sum_k up_k^s)\frac{X_i}{(\sum_k X_k) - Z} \\
\Leftrightarrow \frac{ZB(\sum_k X_k)((\sum_k X_k) - Z)}{T(\sum_k up_k^s)} &\geq \underbrace{X_i(\sum_k X_k) - X_i((\sum_k X_k) - Z)}_{=ZX_i} \\
\Leftrightarrow \frac{B(\sum_k X_k)((\sum_k X_k) - Z)}{T(\sum_k up_k^s)} &\geq X_i \\
\Leftrightarrow \underbrace{\frac{(\sum_k up_k^l)}{(\sum_k up_k^s)}}_{\geq\frac{1}{2}}((\sum_k X_k) - Z) &\geq X_i
\end{aligned}$$

   But because we have
   $$\frac{X_i}{(\sum_k X_k) - Z} \leq \frac{1}{2},$$
   the last inequality holds.

2. $\frac{X_i}{(\sum_k X_k)-Z} \geq \frac{1}{2}$. Here, we have $min\{\frac{X_i}{(\sum_k X_k)-Z}, \frac{1}{2}\} = \frac{1}{2}$. The deviating leecher's utility will not increase if

$$\begin{aligned}
ZB + T(\sum_k up_k^s)\frac{X_i}{\sum_k X_k} &\geq \frac{T}{2}(\sum_k up_k^s) \\
\Leftrightarrow \frac{\frac{B}{T}\sum_k X_k}{\sum_k up_k^s} Z &\geq \frac{1}{2}(\sum_k X_k) - X_i \\
\Leftrightarrow \underbrace{\frac{\sum_k up_k^l}{\sum_k up_k^s}}_{\geq\frac{1}{2}} Z &\geq \frac{1}{2}(\sum_k X_k) - X_i
\end{aligned}$$

Again, because we are in the case

$$\frac{X_i}{(\sum_k X_k) - Z} \geq \frac{1}{2},$$

the last inequality holds.

Therefore, the utility can not increase by deviation, which concludes our proof. □

Overall Upload Speed of a Leecher

One might wonder why we did not discuss that a leecher could fake his total upload speed to be lower than it really is. This is indeed possible, but this is not considered to be a deviation. Rather than that, we regard this as a feature of our protocol. Every leecher is then free to choose his desired total upload speed. Depending on that, he will get an appropriate download speed. Of course we could e.g. assume that every leecher wants to download as fast as possible in the first place, but leaving this open, every leecher is able to find his own desired balance between uploading not too much but still getting good rates. In any case, our proofs hold if we fix the total upload speed of a leecher to a certain value.

4.4.2 Seeder's Protocol

Now we want to prove that a seeder will not deviate from his given strategy.

Theorem 21 (Seeder's Protocol)**.** *No deviation of the seeder from his protocol will increase his payoff if the other parties play their designated strategy.*

Proof. If we leave open how long and how much of his upload speed a seeder wants to offer, we have only one thing left to show. We need to prove that no matter how the seeder would distribute his upload speed over the leechers, the average time to completion does not decrease.

This follows directly from the fact that we consider all other participants in the game to stick to their strategy. Therefore, if a seeder seeds to someone more and to someone else less, the average completion time is still the same.

Let up_i^s be the total upload speed of the leecher, $\omega_1, \ldots, \omega_n$ be the weights on the upload speed to all leechers given by the tracker and $\phi_1, \ldots, \phi_n$ arbitrary other weights. We require, as usual for weights, that $\sum_{k=1}^n \omega_k = \sum_{k=1}^n \phi_k = 1$ and that $\forall k : \omega_k, \phi_k \geq 0$.

Then, we obtain for the average download speed of all leechers in one round

$$\begin{aligned}
\frac{1}{|L|}\sum_{k\in L}(down_k^l + \omega_k \cdot up_i^s) &= \frac{1}{|L|}\left(\sum_{k\in L}(down_k^l) + up_i^s \sum_{k\in L}\omega_k\right)\\
&= \frac{1}{|L|}\left(\sum_{k\in L}(down_k^l) + up_i^s\right)\\
&= \frac{1}{|L|}\left(\sum_{k\in L}(down_k^l) + up_i^s \sum_{k\in L}\phi_k\right)\\
&= \frac{1}{|L|}\sum_{k\in L}(down_k^l + \phi_k \cdot up_i^s).
\end{aligned}$$

Because the average download speed in every round stays the same, the average completion time for every leecher does not differ if the seeder deviates from his protocol. Therefore, the seeder's utility does not increase. □

4.4.3 Computational Nash Equilibrium

Now, when we have the Lemmas stating that neither the leechers nor the seeders will deviate, we can finally conclude with showing that we have a computational Nash equilibrium.

Theorem 22 (The Protocol is a Computational Nash Equilibrium)**.** *The protocols we present for tracker, leechers and seeders form a computational Nash equilibrium.*

Proof. The tracker is trusted and will therefore not deviate. According to Theorem 20, a leecher's utility will only increase by a negligible factor by a deviation. In Theorem 21 we have shown that a deviation of the seeder will not increase his utility. Therefore, no computationally bounded player has an incentive to deviate as long as all the other parties play their designated strategies. □

5 Further Mitigation Techniques

The protocol we present is proven to work in our model. As long as this model holds, nothing can harm it. It immediately follows, that attacks on the protocol will try to violate this very model. In this chapter, we want to discuss possible flip sides of our protocol and say how to protect from them. Because those cases are outside our model, we will give neither definitions nor proofs, but only reason why a certain defending mechanism could make sense. Our biggest concern is the forming of coalitions, but we have also to discuss minor violations of the model, like too many seeders or heavily rejoining peers.

It is interesting to note that most of the mitigation techniques we mention do not need to be deployed in the first place. The reason lies within the rationality of humans itself. If we are able to convince people that even attacks outside our model do not work because we *can* implement mitigation techniques, most likely nobody will even try. Because if someone writes a client that exploits such a fact and we notice that, we can afterwards e.g. release new tracker software which defends against the new attack. Therefore, the people who implemented the cheating client wasted their time. This is important as every mitigation takes away resources and therefore costs money.

5.1 Coalitions

Coalitions are not unusual, neither in reality, nor in game theory, nor in rational cryptography. On the contrary, most protocols in rational cryptography try to achieve at least immunity against coalitions before looking for stronger equilibria than Nash. Hence, the question arises why we did not formally discuss this point.

The answer lies within the mechanism of BitTorrent itself and can be understood best in [HP06], by Hales and Patarin. They give reasoning why nobody uses cheating clients although they are freely available. According to them, BitTorrent has this – on the first glance negative – side effect that there is no metadata search included in the protocol. Therefore, each torrent creates its own small network, called *swarm*. Interestingly, exactly this already *forms* coalitions, namely between you and all other people who want to share the same file, and this coalition is usually not too big. Now you already have some incentives to be altruistic to other users, because you share common interests. And although you cannot talk to them, this creates some kind of bond between all users.

The same holds of course for RatFish. To now successfully cheat, you would need to find a coalition *in* the coalition who is willing to betray other users. But as you cannot talk to anyone, the client should look for other instances of itself, running on other machines, and he might be scolded by other parties already during the search for coalition partners. For example, as soon as a client is released who looks for other instances of himself by whatsoever means, one can patch "honest" clients to watch if they are contacted that way. Then, they can safely blacklist the client who is looking for possibilities to cheat.

There is however a second way to create a coalition, namely to concurrently connect twice to a swarm. For a normal user, this might already be infeasible as a tracker notices two connections coming from the same IP. Therefore, this has to be done using a proxy, but especially free proxies usually have low throughput. Nevertheless, we will assume for the forthcoming subsections that somehow a coalition was established.

5.1.1 Coalition With Fake Identities

Assume, a cheating client managed to connect twice to the system, with distinct identities. However, his overall upload and download speed is subdivided into two parts for the connections, and both of the identities he created only have access to a common pool of pieces. Therefore, as long as he is exchanging with other parties, everything still works the same. The only possibility which opens to the cheating client is to fake exchanges between his two identities and therefore get an unproportional reward from the seeders.

However, what options does he have in detail? The tracker knows what pieces both identities have, so he somehow needs to convince the tracker that they really *have* pieces to exchange. This cannot work by claiming to have more pieces, because for this he would need to forge at least at some point the signature of the tracker. Therefore, he needs to say that he has less pieces than in reality. This is only achievable by starting collecting distinct pieces with both identities, and at some point "exchanging" the pieces gotten so far and then move along. Note that the two identities do not really exchange because they use the same pool of pieces anyway.

As already pointed out, this is unlikely to happen, because you would need a rather good proxy server. If this happens you can get at most the following bonus.

An honest peer L_i on average receives a reward of $\frac{up_i^l}{U^l}U^s$, where $U^l = \sum_k up_k^l$ and $U^s = \sum_k up_k^s$. A cheating peer who is able to use every 2 pieces he gets to fake an exchange obtains a reward of

$$\left(\frac{2up_i^l}{U^l + up_i^l}\right)U^s.$$

For large seeding capacities, this is a reasonable increase. However, we think that such behavior can be detected by the tracker.

An ad hoc solution to this problem is the following: The tracker observes whether two parties are downloading distinct pieces and then exchanging them. Normally, this should not happen to leechers who just download from whoever got something for them. This puts more workload on the tracker, but maybe better solutions can be found by experimenting.

5.1.2 Coalitions With Other Clients

Again, there is only one interesting attack point, namely the number of exchanges. However, for two different users, cheating becomes even more difficult when we implement a simple mechanism: The tracker only counts exchanges when a user downloads a *new* piece. Or, if a leecher downloads from a seeder a piece he should already have, the tracker subtracts one exchange for that. This means for our cheating users:

- Either you claim to receive a piece you already have. Then, this will not be counted.
- Or, you can pretend to receive a piece which you do not have yet. Then, this will be counted. However, you do not really have the piece and therefore need to download this later. Then, you either need to reconnect and claim that you are completely new which gives you more restrictions again. Or you download the piece in the same session later but do not get a reward for this particular exchange.

Practically, this means that two different leechers cannot help themselves by claiming they have fake exchanges.

5.2 Frequent Rejoining

In our protocol, we did prove that rejoining does not increase your utility, given strong assumptions. However, if those do not hold, it *can* help you.

For example, if you really got stuck during the protocol and have nothing to offer anymore to other parties, a simple rejoin can help, as you are getting 5 new pieces again. However, this comes at the expense of being forced to wait for a whole round, T seconds. Now getting 5 pieces of size 256 kB during more than 256 seconds means that you effectively have a download speed of below 5 kB/s.

BitTorrent was built to give people a high download speed. If a user is willing to endure 5 kB/s just to not upload data, he could have used a different method for file sharing in the first place.

Of course you could also try to dynamically rejoin, if you e.g. are getting such a low upload speed that even 5 kB/s would help. However, for this to work you first need a seeder in the system. Second, if you have a seeder it is already improbable that you suffer from such download rates. To be more precise, this can only happen if the seeder is forced to serve such a lot of other users, that even upon rejoining, you would get your new 5 pieces very slowly. In most cases possibly even slower than the download speed you had before.

5.3 Border Cases

At last, we want to look at what happens if our quantity assumptions, e.g. having a seeder, do not hold.

5.3.1 Missing Seeders

This case is almost pathologic. If a system has no seeder, it is close to having an availability which is smaller than 1. This means, the file which was shared at some point is not completely available in the system anymore. However, under some circumstances maybe a system is able to recover. In such cases we have a problem with our protocol: It only allows people to join when seeders are available, as seeders are responsible for giving out the initial pieces.

There is not much we can do about this problem besides somehow involving leechers to help. Several techniques are possible.

1. We simply rely on altruism again. If a newcomer joins a system with no seeder, some leecher helps him out for free.

2. We can give additional incentives to help newcomers. E.g., a tracker could assure a leecher some “free exchanges”, should he help and should there ever be a seeder again.

3. The tracker allows the newcomer to cheat by assuring that he has certain pieces although he just joined. Then, the newcomer can cheat some leechers and get some real pieces.

Clearly, every method has its drawbacks. The first approach needs altruism and can of course be exploited. However, if one keeps the forced wait, the damage should be minimal.

The second approach maybe produces more altruism. Again, this is exploitable, especially by faking distinct identities and rejoining all the time. Then, the tracker gives out a lot of free exchanges to leechers who want to help the newcomer, but this newcomer again immediately rejoins. And of course you never know whether a seeder will come back.

The third method is the least applicable one, as the cheating newcomer gets blacklisted already at the beginning by some parties. Considering that the system was almost dead anyway, this might be fatal for him.

5.3.2 Too Many Seeders

At some point in our proof, we required that we have at most twice as much seeding than leeching power. In the exchange protocol, one has to acknowledge the other party’s exchange in return for the key for the encrypted piece. However, having too much seeding power, it might be better for a user to decline the exchange, because he then gets a lot more reward from the seeders.

Although we did not prove it, this is still not an option, even when the seeding power vastly exceeds the leeching power. First of all, you get blacklisted by your cheated partner, and this decreases your local neighborhood. Therefore, if you do this too often, you eventually do not have other parties to exchange with. Second, if you already have a lot of seeding power it might be that you do not even want to cheat, because your downspeed is fully utilized. And third, for knowing whether this deviation helps you, you need a good picture of the overall leeching and seeding power. Otherwise, you do not know when exactly the point is reached where not giving an acknowledgement improves your download speed.

6 Feasibility Study

Now that we have introduced the new protocol in theory, we want to check whether the workload on the participants is not too high. In theory, everything is acceptable as long as we only have polynomial complexity, but in reality we have to take a good look even at the constant factors. We want to look at all different parties participating in the system and discuss whether their new protocol is acceptable. Often, we will refer to BitTorrent by saying that something uses as much resources as BitTorrent. We do this for convenience, as BitTorrent is widely deployed and therefore gives good insight into what is feasible.

To really be able to argue about the feasibility, we would have to implement the new protocol and test it exhaustively, using a system like planet-lab [KHMP06]. We leave this however open for future work as this thesis' main aspect lies on the theoretical part and just give brief reasoning on why it should work in practice.

6.1 Seeder's Overhead

We start with the most simple party to discuss, the seeders. Concerning network or computational complexity, almost everything stays the same. To summarize briefly, we added to the BitTorrent protocol that

- every seeder tells the tracker its upload speed,
- a seeder is told by the tracker to whom he should upload and
- a seeder notifies the tracker upon having sent a piece to a leecher.

Everything else is basically the same as in BitTorrent. It is easy to see that not much overhead was added. The only addition we need is an open TCP connection to the tracker, where a seeder can send down its updates. For example, an update for a sent piece to a leecher should contain

- the IP of the seeder himself (4 bytes),
- the IP of the receiving leecher (4 bytes) and
- the ID of the piece sent (4 bytes),

which overall give 12 byte of upload overhead compared to a 256 kB piece which has to be sent first. This is clearly no problem.

Also, the coordination with the tracker should be easy. Upon joining the network, the seeder notifies the tracker of its current upspeed. As soon as a user changes the upload speed, another update should be sent. What is more important is that a seeder should, upon leaving the network, at any rate notify this to the tracker. Otherwise, the tracker's calculation for the rewards gets faulty as he expects more upload speed from the seeders than there exists. We can however not change the fact that some users kill their client or their connection, and that the tracker might therefore occasionally miss such a quit message.

A possible countermeasure to this would be if e.g. the tracker requests a specific keep-alive message from a seeder as soon as he is missing some uploaded pieces. For example, a seeder with an upload speed of 256 kB/s, which is known to the tracker, should upload one piece in a second. If the tracker does not get any message for several seconds from the seeder, it is probable that he has quitted.

A more serious problem are large-scale networks with few seeders, e.g. the initial seeding phase of a popular file. Assume that we have 10,000 leechers trying to download from a seeder who has 100 kB/s upstream. Because in the initial seeding phase, this upstream is the bottleneck of the system, we can assume that all leechers exchange approximately the same amount. Hence, the seeder has to split his upload capacity among 10,000 leechers, resulting in 10 B/s for everyone. Obviously, this makes no sense. In such a case we have to accept that we can give no perfect fairness to the leechers if we want the system to stay reasonable, at least not in such a small time frame. Probably, one should switch to a more simple initial seeding protocol in such a case, e.g. as described in the idealistic setting. A seeder then simply cycles through all leechers and uploads to everyone a piece.

6.2 Leecher's Overhead

Compared to the seeder's new protocol, a leecher faces more differences in his protocol:

- He encrypts pieces to exchange and
- he interacts with the tracker to
  - get a signature on the current time,
  - get a random subset of other leechers,
  - check during an exchange whether the other party has a piece,
  - request rewards and
  - acknowledge the other's rewards.

Encrypting Pieces

In practice, we do not need a proven CCA-2 secure encryption algorithm for our protocol to work. Rather than that, we need an encryption scheme just strong enough that a participant would rather upload data instead of brute forcing it. For this, a simple block cipher like AES suffices. Using short keys, one can encrypt several hundreds of Megabytes per second, which is computationally fast enough for our purposes. More importantly, the cipher text is as long as the message, which means that we do not have any network related overhead by encrypting pieces (except of course the transmission of the key, which is however only a few bytes).

Interaction with the Tracker

On the leecher's side, basically everything stays the same. The signature on the current time and the subset of leechers he receives in the same message as in the handshake of the original protocol. Also, a request whether a party really has a piece is just a few bytes, as well as the request for rewards. Compared to the 256 kB per piece, the overhead is still small.

Even for large-scale networks, from the viewpoint of a leecher, he still interacts just with his local neighborhood and the tracker. Therefore, a large-scale network does not affect the leecher and the performance is comparable to BitTorrent.

6.3 Tracker's Overhead

This is the most interesting part. The tracker has a lot of new tasks to perform, and the question is whether modern server architectures are fast enough to cope with this. Once again, we first want to skim briefly over the additions to the protocol. The tracker is required to

- compute signatures on timestamps for arriving leechers,
- compute and verify signatures on the completion of leechers,
- select a random subset of stored IPs,
- store all seeders' IPs as well as their upload speed,
- store all leechers' IPs as well as their completion and their number of exchanges,
- look up whether a leecher has a certain piece,
- compute the rewards for the leechers and
- notify the seeders whom to upload to.

Clearly, this is a lot more work than before. Additionally, we face the problem that a single tracker does not only serve one torrent but usually many thousands of torrents. However, only a few of those are exceptionally popular. For our calculation, we assume having 1.000.000 leechers on the tracker. Further on, we say that they download files of an average size of 1 GB with an average speed of 100 kB/s. This gives us approximately 100 leaving/joining peers per second. Assuming that a signature needs 1 millisecond to be computed, we obtain that 100 ms are used to compute signatures for joining peers. For leaving peers, we only need to compute signatures if they are not done downloading. Therefore, we need at most 200 ms for computing signatures. This is feasible.

The rest are basically hash table lookups, but quite a lot of them happen at one time. With 100 kB/s for 1,000,000 leechers, we have 390,625 exchanges happening per second. For every exchange, we have to look up two times the completion, and increment an exchange count. We assume that fetching a hash value consumes 10 nanoseconds, whereas a replace costs 20 nanoseconds. This means, for every exchange we utilize 40 nanoseconds. This results in 15 milliseconds of overall hash table lookups during one second. This is feasible. Also, we claim that the reward computation is feasible, as you only perform basic operations on the values.

Also, what needs to be discussed is the network overhead. Every such request is just a few bytes, but as the tracker gets millions of those requests, we need to look whether it still scales. A request whether another party has a piece usually just consists of the ID of the piece (4 bytes) and the IP of the other party. Possibly, you have to include your own IP as well, but this the tracker can also deduce from the IP header. Nevertheless, we count it in, resulting in 12 bytes per request. Such a request is sent in an IP/TCP packet, which additionally give us 40 bytes of overhead. A reward request basically has the same size, we only need to count additional 4 bytes in for the second piece.

Again, for one exchange we therefore have to receive a check request of 52 bytes (the response is considerably smaller), and a reward request of 56 bytes, making it 108 bytes in total. Thus, we get in one second 40 Megabytes of metadata just for the exchanges. Additionally, we are sending to newcomers their required information which we approximate with 10 kB per newcomer, giving us additional 1 Megabyte. However, as we are speaking about server architecture, this should be feasible for good servers.

Similarly, the tracker has to notify all seeders after the rewards are computed. Having 500,000 seeder means that every seeder serves approximately two leechers. Therefore, the tracker has to send to each seeder the IPs of these leechers (2 times 4 bytes) as well as the capacity every leecher should obtain (4 bytes per leecher). Again, this is sent using TCP/IP, so we have to add 40 bytes of overhead. This gives us a packet size of 56 bytes which has to be sent to every of the 500,000 seeders. This results in 27 MB every T seconds, which is no problem.

7 Conclusion

7.1 Summary

In the beginning, we have presented the drawbacks of BitTorrent which could someday make it unusable. This popular file sharing protocol is still used by most people, but it also is easily cheated. Should this happen, there will be the need for a protocol for *rational players*, which is not cheatable at all.

We presented this protocol, RatFish. First of all, we have shown in Chapter 3 how a simple approach would look like and gave a strong proof that in an idealistic setting, a rational player is not able to increase his utility by deviating. This was mainly to give the reader intuition on how the protocol will look in the forthcoming chapter.

Then, in Chapter 4, we presented a model which is close to reality along with a more involved protocol which, as long as our assumptions on the utility functions hold, cannot be cheated. Our protocol defends effectively against the main weaknesses of BitTorrent:

- *Free riding.* As we do not have optimistic unchoking, free riding becomes increasingly difficult. Even white washing does only help up to a very small degree, as one always is forced to wait upon rejoining the network.

- *Strategic clients.* Whereas in BitTorrent you could by strategically selecting the peers you upload to achieve a medium performance gain of 70%, this is not possible in our client. The strategic choice is already inherent: Every client uploads exactly as fast to his exchange partners as they upload to him.

- *Garbage upload.* In BitTorrent, although garbage is detected and discarded, it still counts as valid incoming traffic. In our protocol, you can only upload what you really hold. This is always verified by the tracker, who plays a major role in our new protocol.

Afterwards, we examined what happens if our model is violated in Chapter 5 and reasoned that our protocol should also defend against those attacks. Along with the feasibility study in Chapter 6 whether the protocol can be used in practice, this gives us a pretty clear picture that we could replace BitTorrent.

Although nobody will use new file sharing programs when the old ones work perfectly well, there *do* exist possibilities to cheat BitTorrent heavily. We also would like to point out that this might happen on behalf of certain companies, for which popular file sharing methods are always a thorn in their flesh.

On a side note, we want to explain how the name “RatFish” is formed. It is an abbreviation of the thesis and we think that sounds pretty catchy. The name derives from *Rational Filesharing*. Interestingly, there really exists a fish who is called like that. Should *RatFish* therefore ever be deployed, there is a lot of potential for artistic design choices. A possible logo which is an ambigram you can see on the right.

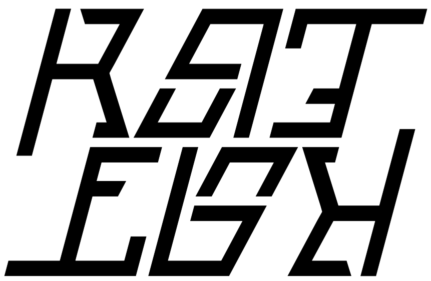


Figure 7.1: The RatFish logo

7.2 Future Work

Of course, before we are ready to deploy our protocol, there is still a lot of work to do. This thesis only examined the theoretical aspects of a new protocol. Before we therefore are able to release a working client, the following steps should be taken.

1. *Complete the proof.* Our proof in Chapter 4 is missing one piece, namely that during the case distinction in Theorem 20, we really covered all existing cases. This is not obviously clear, and to fix this, the whole proof needs to be restructured. For example, this can be done by an induction proof over the steps of the protocol. However, this would make the proof complicated and less intuitive, which is the reason why we omitted this step and just listed the most important cases.

2. *Implement the protocol.* There needs to be software written for clients and for trackers alike. We can build upon existing implementations for BitTorrent, but there are still a lot of changes to be done. Especially, one needs to care for the implementation for the tracker, where every little speedup can help reducing the workload significantly.

3. *Test the implementation.* We discussed in Chapter 6 that the protocol *should* work in practice. However, this is just common sense we used and it could very well be that we missed some important fact. Therefore, our protocol needs to be tested exhaustively using networks like planet-lab [KHMP06], and the statistics have to be checked carefully whether the metadata overhead is really acceptable.

4. *Make the protocol immune against coalitions.* We reasoned why coalitions should not pose any threat to *RatFish*. Nevertheless, for a real proof, a lot is missing and most likely the protocol needs to be rewritten partially. Whether this is necessary or not we only will see in the future. Should there appear clients that can effectively and heavily abuse *RatFish* using coalitions, this case needs to be investigated. Otherwise, we can focus on the first three points.

Having said this, the roadmap for the future is clear, and we really hope that someday *RatFish* describes a serious alternative to BitTorrent.

A Detailed Protocols

Here, we want to give a more precise specification of the protocols that we mentioned in Chapter 4.

$\mathsf{TrackerConnect}(i)$
If i is a leecher:

1. Compute $\mathsf{S}_{sk}(i, T_c)$, where T_c is the current time, yielding sig_t, and send to L_i the message "TIME (T_c, sig_t)".
2. If the leecher claims to have pieces by sending the message "PIECES $(a_1, \ldots, a_N, id, sig_p)$", verify that $\mathsf{V}_{pk}((a_1, \ldots, a_N, id), sig_p) = 1$ and that you have $p := (a_1, \ldots, a_N, id)$ stored. Remove p from your local storage and then set $\forall m \in \{1, \ldots, N\} : A_i^m := a_m$.
3. Send to L_i a random subset of size H of all known IPs.
4. Notify a seeder by sending him "NEWPEER (i, T_c)".

If i is a seeder:

1. Wait for the message "SPEED up_i^s" from the seeder and store this value.

Figure A.1: Algorithm executed by the tracker when a new peer joins.

SeederConnect$_j(i)$

If i is the tracker, then seeder S_j does the following:

1. Send to the tracker the message "SPEED up_j^s".

If i is a leecher, then seeder S_j does the following:

1. If the tracker sends you the message "NEWPEER (i, T_p)", wait until the current time $T_c > T_p + T$ and then repeat the following steps 5 times.
2. Wait for the message "WANT x" from the leecher L_i.
3. Send f_x to L_i.
4. Send the tracker the message "SENT (i, x)".

Figure A.2: Connect protocol for a seeder.

$\mathsf{LeecherConnect}_j(i)$
If i is the tracker:

1. If you are rejoining, send the message "PIECES $(a_1, \ldots, a_N, sig_p)$", where $a_m = 1$ if you hold piece f_m, and sig_p is the signature you received when quitting last time.
2. Receive from the tracker the message "TIME (T_p, sig_t)", and a set of size H containing IPs.
3. Wait until $T_c > T_p + T$, then connect to the IPs got from the tracker.

If i is a seeder: (Repeat 5 times)

1. Send S_i the message "WANT x", if f_x is the rarest piece in your local neighborhood which you do not have yet.
2. Receive piece f_x from the seeder.

If i is a leecher:

1. Send L_i the message "MYTIME (T_p, sig_t)".
2. Receive from L_i the message "MYTIME (T'_p, sig'_t)". Verify that $\mathsf{V}_{pk}((i, T'_p), sig'_t) = 1$. Verify that $T_c > T'_p + T$. If any of these checks fails, abort.
3. Send to L_i the message "$\mathsf{AVAILABILITY}$ $(a_1, \ldots, a_N)$", where $a_m = 1$ if you hold piece f_m.
4. Receive from L_i the message "$\mathsf{AVAILABILITY}$ $(a'_1, \ldots, a'_N)$". Store these values.

Figure A.3: Connect protocol for a leecher.

$\mathsf{Exchange}_i(f_x, j, y)$
If any of the following steps fails, blacklist L_j, meaning that you neither will send nor read any messages from L_j from now on.

1. Ask the tracker whether the other party really holds the piece by sending him "HAS (j, y)" and wait for him to answer positively.
2. Choose a random key $k_{j,x}$ and compute $c_{j,x} \leftarrow \mathsf{E}(k_{j,x}, f_x)$.
3. Send $c_{j,x}$ to L_j and wait for c_y from L_j. This step may take arbitrary long to complete (i.e., the other player could respond with the corresponding piece much later).
4. Send the message "EXCHANGED (j, x, y)" to the tracker and wait for him to reply "OK (j, x, y)".
5. Upon receiving the message "ACKNOWLEDGE (j, y, x)" from the tracker, reply "OK (j, y, x)".
6. Send L_j the key $k_{j,x}$.
7. Upon receiving k_y from L_j, retrieve $f'_y \leftarrow \mathsf{D}(k_y, c_y)$ and verify that $h_y = h(f'_y)$.
8. Broadcast to the local neighborhood "HAVE y".

Figure A.4: Piece exchange protocol for leecher L_i

Bibliography

[Bac09] Michael Backes. Lecture notes for advanced cryptography. Online under `http://infsec.cs.uni-sb.de/teaching/WS08/AdvancedCrypto/index.html`, 2009.

[Coh03] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[Dou02] John R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[Gol01] Oded Goldreich. *Foundations of Cryptography*, volume I: Basic Tools. Cambridge University Press, 2001.

[Gol04] Oded Goldreich. *Foundations of Cryptography*, volume II: Basic Applications. Cambridge University Press, 2004.

[HP06] D. Hales and S. Patarin. How to cheat bittorrent and why nobody does. In *European Conference on Complex Systems*, 2006.

[Kat08] Jonathan Katz. Bridging game theory and cryptography: Recent results and future directions. In *TCC*, 2008.

[KHMP06] Aaron Klingaman, Mark Huang, Steve Muir, and Larry Peterson. PlanetLab Core Specification 4.0. Technical Report PDN–06–032, PlanetLab Consortium, June 2006.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free riding in bittorrent is cheap. In *Fifth Workshop on Hot Topics in Networks (HotNets-V)*, Irvine, California, November 2006.

[PIA+07] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in bittorrent? In *Proceedings of 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 2007)*, Cambridge, MA, April 2007. USENIX.

[QS04] D. Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks, 2004.

[SPM04] J. Shneidman, D. Parkes, and L. Massoulie. Faithfulness in internet algorithms. In *Proc. SIGCOMM Workshop on Practice and Theory of Incentives and Game Theory in Networked Systems (PINS'04)*, Portland, OR, USA, September 2004. ACM SIGCOMM.

[TC05] R.W. Thommes and M.J. Coates. Bittorrent fairness: Analysis and improvements. In *Workshop Internet, Telecommunication and Signal Processing*, Noosa, Australia, December 2005.