TÉCNICO LISBOA

Data Deduplication in BitTorrent

João Pedro Amaral Nunes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede

Supervisors: Prof. João Pedro Faria Mendonça Barreto

Prof. João Coelho Garcia

Member of Committee: Prof. Ricardo Jorge Feliciano Lopes Pereira

November 2013

Acknowledgments

I would like to thank Prof. João Barreto and Prof. João Garcia for their immense dedication and support throughout this project.

Resumo

BitTorrent é a plataforma de partilha de ficheiros P2P mais popular actualmente, com centenas de milhões de ficheiros a serem partilhados. Este sistema funciona melhor quanto mais fontes existirem para se descarregar o conteúdo. No entanto, o BitTorrent só disponibiliza aos seus utilizadores maneira de encontrar fontes dentro da mesma torrent, mesmo se um ficheiro se encontrar totalmente ou parcialmente presente noutras torrents.

Utilizando técnicas de Deduplicação, desenvolvemos um sistema que permite aos seus utilizadores encontrar essas fontes extra, e posteriormente medimos a sua eficácia a encontar novas fontes enquanto mantendo um custo adicional baixo para o utilizador final.

Palavras-chave: P2P, Peer-to-peer, BitTorrent, Partilha de ficheiros, Deduplicação, Deduplicação Distribuída

Abstract

BitTorrent is the most used P2P file sharing platform today, with hundreds of millions of files shared. The system works better the more sources one has to download from. However, BitTorrent only allows for users to find sources inside the same torrent, even if a file is totally or partially located in many other torrents as well.

Using Deduplication techniques, we develop a system that allows users to detect and take advantage of those extra sources, and measure its efficacy at finding sources with a low overhead for the end user.

Keywords: P2P, Peer-to-peer, BitTorrent, File-sharing, Deduplication, Distributed Deduplication

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Peer to Peer (P2P) file sharing applications are very popular, composing almost 50% of the upstream data usage in the US and 10 to 20% of the total internet traffic across fixed network connections, a monthly average of 6.15GB per subscriber on the US alone [48]. BitTorrent is the largest of the peer-to-peer file distribution networks currently used [48], with over 52 million peers connected to over 13 million active torrents, sharing almost 280 million files, totaling over 17EB of information[1].

In P2P filesharing, a user obtains a file he desires by connecting to the P2P network and downloading the file directly from his *peers*, that is the processes connected to the P2P network on behalf of the users, capable of both downloading and uploading files. For popular files, this model has several advantages versus the traditional Client-Server. There is greater fault tolerance, since there is no single point of failure, and faster download speeds may be achieved, since the network load is distributed through several machines.

One other advantage, offered by many P2P filesharing systems, is downloading simultaneously from multiple sources at once. Since downloads are usually limited by the uploader upstream bandwidth, this speeds up the downloading speed to near the user's downloading bandwidth. While server replication can provide these advantages as well, it is usually much more expensive than the P2P alternative.

BitTorrent has peers connected to sub-networks called swarms, instead of having all peers connected to the same network. Each swarm distributes only a small group of files, a *torrent*. Torrents are isolated from each other. Any user can be part of many swarms at once. For each torrent swarm he participates in, he is regarded as a different peer. Due to BitTorrent's method of file transfer, where a peer may share parts of a file even before owning the entire file, any peer that has at least part of the file is considered a *source* for that file. Peers in a torrent that are only uploading are called *seeders* while peers that are still downloading files from the torrent are called *leechers*.

A typical torrent has a small number of seeders and a large number of leechers. This still works fairly well mainly due to the usage of the choke algorithm, a tit-for-tat kind of algorithm where leechers receive

[1] Information extracted from the torrent indexing site 'IsoHunt.com' in September 2013.

more from other peers if they share what they have already obtained [12]. To finish downloading the torrent, a peer needs to upload almost as much as it downloads, keeping the torrent available for longer.

Having more sources for a file (or pieces of a file) has multiple advantages. Faster download speeds is one, but sometimes, download speed is less of an issue than download location. A user using a data plan that imposes a limit on international traffic may prefer to download only from sources on the same country. On locations with limited internet access, finding sources in peers from a local area network may be the difference between finishing a download or not. So, being able to find all available sources for a file would be advantageous for anyone downloading it.

BitTorrent uses the swarm of a torrent to identify the possible sources for that torrent's content. For very popular torrents, the number peers (be them seeders or leechers) is in the range of thousands, meaning thousands of possible sources. In those popular torrents, even if all the seeders leave, it is likely that the swarm, as a whole, contains a copy of the finished torrent. This happens due to BitTorrent's rarest-first transfer policy, where the pieces shared by the least amount of peers get traded first, trying to maximize the number of full copies available [28].

However, for smaller, not very popular torrents, the scenario is different. There are very few seeders for a few leechers. For these smaller swarms, the throughput is usually lower [15], and it's harder to complete a download due to less peers available [28]. This problem affects even torrents that are now popular, as sometime in the future they will decrease in swarm size, losing many of their seeders and leechers. This causes unavailability for that torrent [35].

If we could give the peer more sources from where to download the file from, it would allow for faster downloads and, in some cases, to finish download from a dying torrent. But the question is, how can we increase the number of available sources?

The answer lies in the torrents' content. The isolation between torrents means that torrents with the same contents, or very similar in content, cannot take advantage of each other.

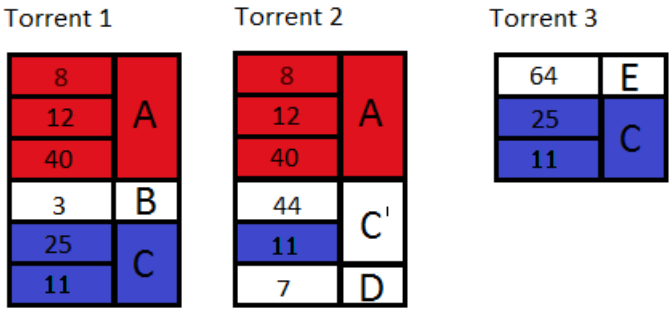

Figure 1.1: Three torrents with something in common. The smaller, left-hand boxes represent pieces, with the number being a piece hash. The larger, right-hand boxes represent files, with each letter representing a different file. Red means the the file is present in more than one torrent, while blue means only some chunks are identical. Peers can only see either Torrent 1, Torrent 2 or Torrent 3

If each torrent had 100 peers, a peer joining, for instance, Torrent 1, would see 100 sources for A, and 100 sources for C, while in reality, there are 200 sources for A, and 200 sources for C, with an added 100 sources for the second half of C. That means 200 sources completely ignored by that new peer, since he could not see them.

Such an example may seem far-fetched, but it is more frequent than it might appear. For example, Ferreira [20], cites 8.7% of the examined torrents contained over 75% of its content in common. Pucha et al. [42] found the median percentage of data shared between MP3 files (its *similarity*) in the eDonkey and Overnet P2P networks to be of 99.6%, with a median of 25 files in a set of 3725 MP3. Practices such as bundling [35] and repackaging [19] create even more duplicate sources for a file, which combined with the above findings, means there is an immense quantity of potential sources that is currently not exploited .

In order to exploit those extra sources, the file sharing system need to be able to identify and locate redundant data. To do it, we turn to the field of data deduplication.

Data deduplication aims to reduce stored or transfered data by identifying redundant data and storing only one copy or transferring only the non-redundant part. There is one group of data deduplication techniques called Compare-by-Hash, which share a similar methodology. They first take a set of data (such as a file) and divide it into smaller pieces, called chunks. Then, they calculating a unique identifier, called a fingerprint, for each chunk. Finally, they compare the fingerprints with each other, and all chunks whose fingerprints are equal are considered identical.

Taking this technique into account, there is a way of detecting duplicates across the torrent network. If we have the fingerprints of the chunks of a file, we can use that data to find similar or identical files through the network.

To make it clear, our objective is to allow the user to accurately find many more sources than BitTorrent currently allows, while minimizing overhead for that end user. Of course, we also need the system to be realistically implementable at the scale of BitTorrent.

In this document, we will present a novel solution. First, using a new approach at fingerprint storage, Multi-level fingerprints, we create a type of metadata that can be much smaller in size, yet keep the benefits of the small-size chunk redundancy finding. Then, we designed a set of services that allow our solution to find and expose redundancy to the user, all the while keeping the user overhead at a minimum as well as stay viable on the real world.

In this solution, the client wishing to find more sources for a file or a chunk of a file contacts the *lookup service*. The lookup service is responsible for storing the information about the all extra found sources, and providing it to the client.

In order to find new sources, the *insertion service* takes the metadata from files not present in the system and adds any chunks they have in common with other files to the lookup service.

Finally, to keep this solution scalable to the size of BitTorrent, we have the *maintenance service*, a service responsible for keeping the lookup information fresh and to keep the insertion service from becoming unsustainable in size.

We have fully implemented and tested the Insertion and lookup services, and part of the Maintenance service, and we drew results from experimentally evaluating the interaction between test clients and these services. We show that, for over 2/3 of the files downloaded, we could find some new sources, with over 2/3 of those having more 50% to download from, even if each file had only one peer supplying it.

We also show that we can detect at least as many sources as the closest related work, SET, with some files reaching 10 times more file sources than SET could find.

Finally, we show that our system has a low overhead for the end user, both in terms of time spent, as in exchanged message sizes.

The remainder of the document is organized as follows:

In Chapter 2, we will look at the related work, starting with P2P fiile sharing in general, then looking at BitTorrent and the closest related works to our own.

In Chapter 3, we will talk about the architecture and design of our solution, starting with a top-level view of the system as from the client point of view, then talk about each piece in more detail. Chapter 4 is about the decisons taken when implementing our solution.

Chapter 5 is where we evaluate our work, by measuring the time and message overhead of using our system, as well as compare it to the closest related work.

In Chapter 6 we present our conclusions, and we discuss Future work.

Chapter 2

Related Work

In the next Section, we will begin by talking about P2P file-sharing applications in general, raising the most important issues in such applications. We then look into the architecture of BitTorrent, to see how it approaches some of the common problems of the P2P file-sharing world, and which other problems are raised by it. Afterwards, we shall look at some attempts at improving BitTorrent, and how they can be improved upon.

Next, we look at deduplication, first explaining the most important architectures of its field, then looking at that area applied to distributed system, and end with an overview of which techniques would bring the most benefits to the BitTorrent world. Finally, we shall look at the nearest solution to our source-finding problem, SET, by looking at its architecture and design.

2.1 P2P file sharing

In the P2P architecture, any user of a P2P network is considered both a client and a server. Each program connected to that network is called a *peer*. In "A survey of peer-to-peer content distribution technologies" [4], P2P systems are defined as "distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources(...) capable of adapting to failures and accommodating transient population of nodes(...) without requiring the intermediation or support of a global centralized server or authority". The capacity of a P2P system is larger with an increase in the number of peers, which is the opposite of client-server, where the increase in users means a decrease in service quality.

It is usually assumed that all peers in a P2P network can perform the same tasks. P2P systems are capable of distributing the workload dynamically, scale with more clients and are capable of adapting very well to changes in peer population [4]. This makes P2P systems attractive to many types of applications, not only file sharing.

Having content scattered throughout the network raises an important question: how can we find what we are looking for? The answer to that question differs between P2P systems, based on its structure and degree of centralization. While this applies to all forms of P2P networks, let us focus on the more

specific case of P2P file sharing from now on.

Structure

Structure refers to the way the network between peers is organized. It can be either *structured* or *unstructured*.

In *structured* systems, the source for a given content can be determined by the network overlay. Content may be a file, or any other piece of data. One type of P2P structured system is the distributed hash table (DHT), where nodes choose a value that identifies them, and content is stored in the node whose ID more closely matches the content' identifier. Kademlia [34] is one of such systems.

Freenet [11] is a different kind of structured system, called loosely structured. In this system, when a piece of content must be located, the node IDs provide a likely location for where to locate the content, instead of determining where the content is located. The content is then routed back to the requester by a random path. This allows for the content source to remain anonymous.

In *unstructured* systems, any peer may have any piece of content. There is no way for a peer to know the location of a piece of content from this network's topology. P2P file sharing systems are usually unstructured, and content location must be performed in some other way. Some P2P file sharing systems, however, also use a secondary structured P2P network to locate peers and content sources.

Centralization

Centralization refers to the reliance of a P2P network on a client-server architecture for one of more of its services.

A P2P system that uses a central server for functions such as to locate peers and content is *hybrid decentralized*. This central server contains a database with peers that are currently logged in and may contain a database with files and their sources. The peer must communicate with the server if he wishes to join or leave the network, must tell the server which files he wishes to share with the other users, and must request from the server the location of the content he wishes to download. The file exchange, however, occurs directly between peer pairs.

The hybrid decentralized approach allows to offload the bandwidth-consuming part of the file distribution to the peers, but keeps most of the problems of the client-server architecture as well. There is still a single point of failure, and the network capacity does not increase as well when new peers join. However, this was the approach used by many early P2P systems, such as Napster [44]. The original BitTorrent specification also places it in this category.

Such systems require that a peer wishing to join the network register itself in the central server before communicating with other peers. If the peer wishes to share a file, it must first register it in the central server as well, which keeps a database of shared files. If the peer wishes to download a file, they must

ask the central server for its information, including which currently online peers have that file. Only then will the peer contact the other peers to request that file.

On the other end, we have *purely decentralized* systems. These P2P systems have no centralized structure, relying only on their peers to work.[1] However, locating both peers and file sources is much more difficult. Some purely decentralized systems such as Gnutella [3] contact all known peers (up to a certain range) in order to search for more peers and files. This practice, query flooding, is not a very efficient method for content location, but if allowed to reach the entire network, will locate all files. Much work has been developed in an effort to increase the efficiency of peer and source locating in such systems, such as using random walks to find peers instead of flooding [32].

With the introduction of the DHT tracker, BitTorrent is now capable of becoming purely decentralized.

In a mixed approach that tries to get the best of both worlds we have the *partially centralized* systems. In these P2P systems, such as Kazaa [30] and Gnutella v0.6 [27], some nodes are considered special (usually called supernodes) that have added responsibilities. These nodes act as the central servers from the hybrid decentralized approach, keeping source locations and a group of known peers. However, these supernodes are simply peers that have been selected to fulfill that role, and not special machines (or processes) dedicated to keep these structures. They are selected based on criteria such as apparent bandwidth, processing capacity and uptime. They are not a critical point of failure, as if a supernode fails, another peer will simply take its place.

Supernodes form a second network on top of the normal P2P network they belong, composed only of supernodes. To find sources for a file, a normal peer must ask its supernode for the sources. If the supernode knows the source for that file, it replies with the peers that have that file. If not, it sends the query to the other supernodes it knows about. This messaging scheme is similar to the hybrid decentralized case when dealing with peer-supernode interactions, but is similar to purely decentralized when between supernodes.

2.1.1 Challenges in P2P File Sharing Systems

P2P file sharing systems have some core problems that must be addresses. The first one is the problem of content availability.

Content availability

P2P Systems are appealing due to their high fault tolerance. However, achieving that fault tolerance is a problem. P2P systems are usually composed of a peer population that is unstable and always

[1] Servers may be involved in the bootstrapping phase, as the client needs to learn about at least one active peer to join

changing. To keep content available, there must be enough sources across all peers that, after some leave, it is still possible to retrieve that content.

One possible solution is *passive replication*, where a file is replicated whenever a peer downloads it. This happens automatically in P2P file sharing.

Another solution, mostly used in structured P2P networks, is *cache-based replication*. In this solution, when a file passes through a node while being routed to their destination, the routing node keeps it in its cache. The node is then able to reply to requests for that file with the file, even if the node that is responsible for storing the file is down. This solution is used by Freenet [11].

Free riding

P2P Systems in general are based on distributing the workload equally, where all peers contribute to the effort. However, there are always some peers that use the system and don't give anything back. Those peers are called *free riders*. In the context of file sharing, a free rider is a peer that only downloads files, never uploading any.

In a *trust-based* solution, the user is given a reputation that is shared across all interactions with the other peers [26] [49]. That reputation can be use to provide benefits to well-behaved peers, or punished mis-behaving peers. Benefits include reduced waiting times and higher priority in download queues (as used in Kazaa [30]). Punitive measures include refusing to upload a file to a peer with too low of a reputation. Reputation may also be used simply as an aid for peers to decide which peers to download from.

Anonymity and security

Another problem in P2P systems concerns security and anonymity. There are entire content distribution P2P systems concerned mainly with allowing peers to remain anonymous, store files in a secure manner, and provide deniability about the source of the content, such as *Freenet* [**?**]. We will not cover these subjects in this document.

2.2 BitTorrent

BitTorrent [13] is somewhat different from older P2P file sharing networks. When first introduced, it was a hybrid decentralized system, with a peer having to contact a server, called a *tracker*, that contained a list of peers that shared that torrent. However, thanks to the addition of the DHT extension [31], it is possible for it to be a purely decentralized architecture.

BitTorrent follows a different approach to file sharing from other P2P file sharing networks. It uses the concept of a *torrent*, which describes a file or set of files. Peers that join a torrent share and download only the files it describes. BitTorrent does not have a single large peer network where all peers interact. Instead, each torrent could be considered a P2P network of its own, isolated from all the others. For

each torrent, the group of peers sharing it form a network mesh of their own. The peers present in that torrent mesh are called that torrent's *swarm*. Users may share several torrents at once, but are a different peer for each swarm. This means that, even if two peers are sharing multiple torrents, they will not be able to see that fact.

BitTorrent was designed to ease the transfer of very large files from a content provider by allowing other users that had started to download the file to redistribute parts of it. Those parts are called *pieces*. A peer that has all the pieces is called a *seeder*, and all others are called *leechers*.

Files in BitTorrent are not downloaded entirely from a single peer in the swarm, but from several peers in parallel, downloading pieces individually, and then assembling the entire file when completed.

Due to the isolation of torrents from one another, and the lack of a common P2P network to all torrents, BitTorrent does not have an internal content finding system[2]. A torrent is identified uniquely by its *infoHash*, the SHA-1 hash of the metadata that defines the torrent's contents. In order to obtain a infoHash, a user must either receive it from some other source, by means of a link (a *Magnet Link*), or must download the metadata file (the *.torrent* file) and calculate it from the downloaded metadata.

Some websites, called torrent indexers, allow for users to publish their .torrent files in their servers, and search the web for other .torrent files. This allows for users to learn about torrents for the content they want, and join the torrent.

2.2.1 BitTorrent protocol

As a way of explaining the BitTorrent protocol, we will provide an example that shows a simple torrent T1 where a peer P1 joins for the first time and tries to download it, to the end.

Illustrative Example

Let's consider a user who just downloaded the .torrent file for torrent T1. He will join the torrent as Peer P1. T1 contains 4 pieces, from 0 to 3.

First, P1 must read the torrent to find the Tracker (or trackers) for that torrent. P1 will then contact each tracker provided in that .torrent to join the system. The trackers provided 2 peers, P2 and P3. The following image describes the starting state of all 3 peers:

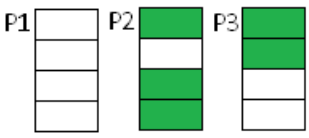

Figure 2.1: Starting state of the torrent in each peer. Pieces in green are chunks that are already present in the peer, while empty, white pieces have yet to be downloaded.

[2]While it can be implemented with the aid of current BitTorrent extensions such as DHT and the metainfo extension, it is not possible to simply search all shared files from inside BitTorrent, in same manner as older P2P file sharing systems allowed querying. There are some clients, such as Vuze, that allow to search from inside the BitTorrent client, but that search is made in the same way as with indexers

P1 contacts every peer he can, sending a handshake message. P2 and P3 reply with their own handshake messages, which also contains a bit array indicating which pieces they have.

After that initial handshake, P1 sends 'interested' messages to all the peers that have pieces he does not have. Because P1 has no pieces, he will send the messages to P2 and P3. Then, he just patiently waits for an 'unchoke' message, that will allow P1 to request a piece.

P2 sent an 'unchoke' message to P1. P1 may now request any piece P2 has. P1 will request the rarest piece. From the image, we saw that piece 0 is present in 2 peers, yet 1, 2 and 3 only have one source. P2 does not have piece 1, so P1 will randomly choose between pieces 2 and 3, and requests piece 3. P2 sends P1 piece 3, that P1 will download until it finishes, or is choked by P2.

Meanhile, P3 is unchoked by P2, and the rarest pieces are 2 and 3. P3 randomly selects piece 3. At the same time, P1 is unchoked by P3, and the rarest piece it has is piece 1. So, P1 downloads piece 1 from P3.

After a while, P2 chokes P1 and P3. P1 continues to download piece 1 from P3. The state when that happens is illustrated in the next image.

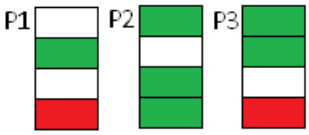

Figure 2.2: Current state: Red pieces are being downloaded by the peers.

P1 then announces to all his known peers that he has piece 1, using a 'have' message. P2, that needs piece 1, sends an 'interested' message as a response to P1. P1, having downloaded from P2 already, and having free upload slots to give, sends an 'unchoke' message to P2. P3 did the same, so P2 can download piece 1 from P3 as well.

P2 unchokes P1, and requests piece 1, as well as unchoke P3 and request piece 1.

P1 and P3 upload piece 1 to P2. Both P1 and P3 decide to request the remaining length of piece 3 from P2. When P3 finishes downloading piece 3 from P2, he requests piece 2. As P1 is still unchoked by P3, P1 requests piece 0 from P3. P3 sends the data. P1 then finishes piece 3 from P2.

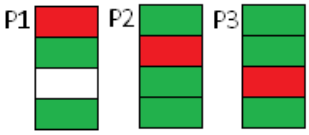

Figure 2.3: Current state: P2 is downloading from both P1 and P3

Right now, P1 has pieces 1 and 3 finished, and is downloading 0. P2 finishes downloading the last remaining piece. P3 does so as well. P2 and P3 send 'have' messages for the last pieces they downloaded, and finished downloading the torrent. They each send a 'completed' message to the tracker, and are now seeders for this torrent.

P1 now requests from both P2 and P3 piece 2, the last remaining piece. P2 and P3 start sending piece 2. As soon as P1 finished downloading piece 2, he sends to the tracker the 'completed' message and also becomes a seeder.

This simplified interaction reveals many aspects from the BitTorrent system. First of all, most of the work is done by the peers. The tracker interaction is limited to the first interaction with the tracker when the peers joins (*started* message), periodic announcements to tell the tracker that peer is still live (*empty* message) and request new peers, when the peer finishes the torrent (*completed* message) and when the peer leaves the torrent (*stopped* message). Other than that, the peers are the ones that exchange most messages.

Also, this interaction shows a bit about two important algorithms in BitTorrent, the Choking algorithm and the piece selection algorithm.

Choking algorithm

The choking algorithm [13][28] is an algorithm that allows BitTorrent clients to choose to which clients they are going to upload. The choking algorithm is the most used as the 'tit-for-tat' algorithm mentioned in the base Protocol. It works as follows:

Every 10 seconds, a leecher will unchoke 3 interested peers that had the highest upload rate to the unchoking leecher during that torrent interaction. Every 30 seconds, a random peer from the remaining interested choked peers is unchoked. This last peer is said to have been 'optimistically unchoked'. Optimistically unchoked peers exist for two main reasons. The first is to allow the unchoking peer to find information about peers with which it never exchanged torrent data. The second is to allow new peers with no pieces to trade to get their first pieces.

For seeders, the original protocol had a similar unchoking policy to the leechers, but newer versions order peers by the last time they were unchoked, and unchokes the 4 that had the most time pass since they were last unchoked. This method was found to be very efficient in reducing free riding [28].

Piece selection algorithm

The piece selection algorithm implemented is the rarest-piece-first algorithm for most of the interaction. In this algorithm, the first step is to gather information about how many pieces are available on the network, which is called that piece's *availability*. The algorithm gathers the availability of pieces by counting the 'have' messages the peer has received so far. When the peer is unchoked, it will request the piece with the lowest availability.

In the start, when peers have no pieces to offer, they use a random-first policy, where they simply request a random piece, without caring for its rarity. This is usually done until 4 pieces were downloaded. In the end, when few pieces are missing, a special endgame mode is activated, where the peer requests everyone for all remaining pieces, regardless of rarity[28]. However, for the sake of demonstration, these rules were skipped in the example.

Metadata

Going deeper into BitTorrent, we will talk about the three main pieces of this system: the .torrent file (metadata), the trackers and the peers.

The metadata is composed of a bencoded [13] dictionary containing several entries:

- *announce* - the tracker to contact for this torrent;
- *announce-list* - the list of trackers to contact to track this torrent. May replace *announce*;
- *info* - the dictionary containing the important information that makes this torrent. The *infoHash* is the SHA-1 hash of this entry. It contains:
  - *path*- A list containing the relative file path for each file in the torrent, as well as its size. The location of a file X in the torrent is given by the sum of the lenght of each file until X is declared. Only present in multi-file torrents;
  - *length* - The total size for a single-file torrent;
  - *private* - A flag that indicates this torrent is a private torrent. This entry may not be present in the dictionary;
  - *name* - The suggested name of the file, or a identifier for the torrent. Only required in single-file torrents;
  - *piece length* - The size for each piece of data to be transferred (equal for every piece);
  - *pieces* - the SHA-1 hashes of each piece, used to confirm the integrity of the piece after arriving, in a byte sequence with 20*numOfPieces bytes.

The *private* field, when it appears is usually a 1 integer. This marks the torrent as private, meaning only private trackers are allowed to track that torrent and control the peers that can join. This makes that torrent untrackeable by DHT, not shareable using Peer Exchange and and unobtainable with the metainfo extension. It is not part of the BitTorrent standard, but is accepted and respected by most BitTorrent client designers.

All hashes in the BitTorrent protocol are SHA-1 Hashes. SHA-1 is a cryptographic hash, meaning that for a very small variation in input data, a large difference in the produced output happens. Cryptographic hashes are commonly used as message digests, that allow to detect if a piece of data was altered in any way since the digest was created. They are designed so that, given a file F1, it is very hard for an attacker (or a random error) to produce a second file F2 where Hash(F1) = Hash(F2). These types of hashes have been determined as safe for use as identifiers [7]. BitTorrent uses hashes both as an identifier on the torrent infoHash, and as a way to detect errors, between pieces exchanged in the *pieces* field.

Trackers

Trackers recognize only four requests from a BitTorrent client: *started*,*empty*,*completed* and *stopped*. Trackers may also accept HTTP connections requesting for statistics, through a *scrape* command. The

tracker replies to the *empty* and *started* requests by returning an updated list of peers that the client may connect to. That list may come in one of two formats: bencoded and compact Peer List [22]. The connection to the tracker can be using TCP [13] or UDP [40], with UDP adding handshake messages.

Using multiple trackers for the same torrent is possible, as to start tracking a torrent, all that is needed is the infohash of the torrent and at least one peer. Sending a *started* message makes the tracker track the torrent. Trackers are isolated from each other, and do not share peer lists. This causes a problem, where peers downloading the same torrent from different trackers cannot see each other. Consider the following example. P1, P2 and P3 are all downloading torrent T, but P1 and P2 are using Tracker Tr1, and P3 is using tracker Tr2. P3 would not be able to download from P2 and P1. However, if P2 had joined the swarms from both trackers, it would be able to download from both P1 and P3, while P1 and P3 would not be able to download from each other.

In some cases, the independent swarm behavior is the intended one. There are some special trackers, *private trackers* [10], that track torrents that only special users registered with that specific tracker can use. Torrents for those trackers are marked with a *private* flag on the metadata, and cannot be tracked by trackers that are not declared in the *announce* or *announce-list* fields, or use any other type of peer discovery other than those trackers.

Messages

Messages between peers are exchanged using TCP [13] or uTorrent Transfer Protocol (uTP) [38], a specialized transfer protocol built on top of UDP. The first message exchanged between peers is the handshake message. Handshake messages contain the sending peer ID, the infoHash of the torrent they are sharing and 64 bits containing BitTorrent reserved flags. The handshake message may also contain one array of bits, with as many bits as there are pieces in the torrent, with 1 bits indicating the sending peer has that piece. After the handshake, the following messages are sent between peers:

- *choke* - Informs the peer that it will be choked by the sender from that message on;
- *unchoke* - Informs the peer that it is no longer choked by the sender;
- *interested* - Informs that the sending peer is now interested in the receiver's contents;
- *not interested* - Informs that the sending peer is no longer interested on the receiver;
- *have* - Followed by a part number, informs the receiving peer that it now contains a certain part;
- *bitfield* - Only sent immediately after the handshake, indicates the pieces (in a bit packed array) that the peer currently has. Not required to send, and only sent when the peer has at least one part to share;
- *request* - Asks the peer to send a specific piece. Contains the requested piece number, the offset from where to start sending the piece, and the total length of the sent data. The length is limited to $2^{16} - 1 (< 64KB)$, so pieces may need more than one request to be fully downloaded;
- *piece* - Reply to a *request* message, contains the piece number, the begin offset and the requested data;

- *cancel* - Sent after a *request*, and with the same fields, it is sent by the requester to ask the sending peer from sending the requested piece.

BitTorrent Extension protocol

The extension protocol provides a way for programmers to create their own additions for BitTorrent without breaking compatibility with the mainline BitTorrent protocol. The extension protocol provides a way for clients to learn which extensions are understood by their peers, and allow to send messages using the BitTorrent protocol through *extended* messages.

The extension protocol specifies that each extension must have a unique name, and is assigned a ID byte in each client, that may be different between clients. The first part of the extension protocol is, right after the handshake message, to send an extended message with that client's local mappings. That message may also include a port number to be used for the reply, and an IPv4 or IPv6 address where the peer is located. The reply sent will be identical in format, containing the mappings from the receiver, and the port or IP for further replies.

After that handshake, other *extended* messages sent will start with a byte that matches the ID of the extension desired in the receiving peer, followed by the payload the extension creator wished to deliver.

For instance, if peer "A" knows 'UT_pex' extension (standard Peer Exchange) as id 2, and peer "B" knows it as id 7, messages from "A" to "B" would be sent as $\{20, 7,$[peer exchange request]$\}$ and the reply from "B" to "A" would be $\{20,2,$[peer exchange reply]$\}$.

With the extensions to the BitTorrent Protocol, new methods for a client to get new peers were added. They were Distributed Hash Tables (DHT) [31] and Peer Exchange (PeX) [39].

DHT extension

BitTorrent DHT was designed to allow an alternative to centralized trackers, and can be used to implement a totally distributed P2P system. In this method, a DHT table is used as a form of tracker, where the key is the infoHash of the torrent and the values are the peers connected to that torrent. Clients wishing to join a torrent contact any known node of the DHT to find out the location of other peers that belong to that swarm. To solve the problem of inital connection to the DHT, there are usually one or more well-known nodes that are shipped with the client distributions.

There are two important DHT implementations in BitTorrent, the Azureus DHT and Mainline DHT. They are incompatible with each other, but all known DHT-implementing clients can use Mainline DHT, with Azureus DHT being used together with Mainline DHT in Vuze, Azureus successor. Both are based on the Kademlia DHT with some differences between their implementations [14].

Peer exchange extension

Peer Exchange is a gossiping algorithm that allows for peers to trade their list of known peers directly with each other. This list of known peers is unique for each torrent, but may contain peers from multiple

swarms for that torrent.

After a peer sends a first message with all known peers, he will send an update message periodically, indicating the peers he has added or removed from the list since the last update.

There are several implementations of PeX, with the most common being the Mainline PeX implemented using the BitTorrent extension protocol.

Peer Exchange allows for cross-swarm integration. In our tracker isolation example, P1 and P3 could not see each other. However, with Peer exchange, P1 could learn of P3 by exchanging lists with P2, even though P3 still does not know about Tr1 or P1 about Tr2.

Metadata exchange extension

In order to join a torrent, a user only needs the infoHash for that torrent. However, without first downloading the .torrent file from some other place, the user would have a very hard time exchanging any meaningful information. The user would not be able to download pieces. Not only would the user not know how to reassemble the pieces received, it would also be impossible to check if they were received correctly.

The usual way to obtain a .torrent file is to download it from a torrent indexing site. Sites such as“thepiratebay.se” or “isohunt.com” allow users to search for content available in the BitTorrent network. So, to obtain a .torrent file, a user either searched for them in the web or received them directly from another user, by other means such as an official website or e-mail.

With the adoption of the metainfo exchange protocol [23], however, that changed. This protocol extension allows for peers to download the info dictionary directly from other peers, allowing them to join the torrent with only an infoHash and a tracker. Combined with DHT, it is even possible to join a torrent with only the infoHash.

MagnetLinks are URI that contain the InfoHash of the torrent and other information such as the *announce* tracker for that torrent. In the absence of an announce field, it is implicit that the tracker is the mainline DHT. To start hosting such torrents, a client only needs to create the .torrent file for itself, connect to the DHT or other announce tracker, and start seeding as usual. Clients that wish to join that torrent simply copy the MagnetLink, and if their client supports MagnetLinks and the metainfo extension (and DHT in trackerless torrents), paste the MagnetLink in the Client input area and join the torrent.

The simplicity of use and smaller sizes has caused a large adoption by the community, with the most commonly used BitTorrent clients[3] supporting them. Some torrent indexers, like 'thepiratebay.se', stopped hosting .torrents, and offer only MagnetLinks.

[3] uTorrent, BitComet and Vuze

2.2.2 Finding sources in BitTorrent

Finding new peers

Using one torrent's infoHash, a peer is capable of finding all sources and peers a tracker knows about[4]. However, this is limited by how many peers and sources a tracker knows. Each tracker follows a torrent individually, forming their own swarm for that torrent. This problem was already mentioned in the tracker subsection above, and shown with the tracker isolation example.

This multiple swarm per torrent problem was already studied by Dán, G et al. [15]. They propose an algorithm called DISM, that would balance the swarms in each tracker by making making sure each swarm was not too small and not too large.

Other solution for this problem comes from the use of DHT and PEX.

Any peer connected to DHT shares the same tracker(the DHT tracker), which can be accessed by any peer that knows the DHT extension for any non-private torrent.

However, the use of PEX can be much more powerful in finding new peers.

In the torrent isolation example, if all peers have Peer Exchange, P1 can learn of P3 by sharing the peer list with P2. Even if no other peer of Tr1 or Tr2 had PEX, it would be possible for P1 to download from any peer from Tr2 that P3 knew about. If, for some reason, a peer P4 connected to a tracker Tr3 and Tr2 also had PEX, and talked with P3, P1 would eventually find peers from Tr3. With PEX, it is possible to find peers from any tracker, provided at least one peer from that tracker is also present in our swarm.

Another solution is presented by certain indexing sites, such as 'isoHunt.com'. These sites scrape a list of known trackers about a infoHash, and add all the trackers that reply with some peers to the *Announce-List*. If the peers could exchange that list with an extension, similar to the metadata extension, that would allow all peers to access any tracker.

Although the above solutions would give us all the sources for torrents, this would not give us all the sources for files in BitTorrent.

Finding file sources

BitTorrent differs from other P2P file sharing algorithms by allowing the user to download a group of files as a single task, instead of dealing only with individual files. This is particularly convenient for the end user, as it allows transferring entire collections of files (such as music albums and series) in a single transaction, without needing to use archiving programs. However, the way BitTorrent identifies a torrent's content allows for multiple copies of the same file to be distributed under different torrents, and those

[4]While the protocol limits the response to 50 peers, constant querying of the tracker allows for a peer to eventually obtain all peers known by that tracker

extra sources are not possible to find using BitTorrent alone. This file duplication may occur in several cases.

One first case that can happen is if one or more users upload the exact same torrent, but changing something that alters the info dictionary of the torrent's metadata. The content will be the same, but the infoHash will be different.

If a user uploaded the same torrent, but change the name of one or more of the files, that would result in two different torrents, with different infoHashes. That happens because the 'path' entry, that contains the name and location of a file, is part of the info dictionary, meaning a change in a 'path' entry changes the hash of the info dictionary.

This case is commonly seen when two or more rival groups compete to release the same content. In those cases, they may simply change the name of the content file to reflect their group identification, or simply add an empty file with an URL for the group's web page.

This case can be detected rather easily if we calculated the infoHash only over the 'pieces' entry.

Another, more common case is when a file is distributed in two different torrents, yet is mixed together with other, distinct files [19].

As an example, consider an album containing six songs, where one of the tracks was once part of a single. That music may be the exact same on both the album torrent and the single torrent.

This case would be very hard to find using only BitTorrent. However, BitTorrent calculates the pieces of a torrent by first joining together the entire content as a single, big block of data, with the fies ordered as the uploader requests. The way BitTorrent divides the data into pieces makes it very easy for them to have a different hash over the slightest shift in the data (fixed-sized chunking)[37]. As such, the only way to detect a file is if it started exactly in the same position inside of a piece, the pieces had the exact same size, and the data before the file was identical on both torrents.

If the file always started at the beginning of a piece, it would be possible to identify identical files by looking at their piece hashes.

BitComet implements a system that allows torrents made with their client to be padded at the end of a piece using a dummy file, in a way that each new file would start at the beginning of a new piece. That dummy file would be ignored by their client. Despite clever, it was heavily criticized by the community because all other clients downloaded the dummy files, sometimes wasting considerable bandwidth. In a large file collection, that result in hundreds of files, the padding summed up to 10% of the downloaded data[5]. However, this helped identify identical files across different torrents.

An even more common case is that of files that are nearly identical, but are not the same file [42]. Due to the way files are distributed in BitTorrent, it would be beneficial to download the pieces in common from both torrents, and download only the rarest, single torrent chunks from the original torrent.

[5] http://torrentfreak.com/bitcomet-pollutes-bittorrent-with-junk-data/

Both above cases cannot be easily spotted due to the way BitTorrent builds its pieces. Otherwise, we could have used a compare-by-hash technique (which we will discuss briefly in the 2.4) in order to find the common pieces and download them.

2.3 Deduplication

Deduplication is the area of study that tries to exploit duplicated data, by storing only one copy of that data. Distributed deduplication is a subset of the deduplication area of study that deals with techniques to identify identical data between two remote points, in order to reduce the amount of data transfered.

2.3.1 Delta encoding

One popular category of deduplication algorithms is *delta encoding*. Delta encoding algorithms allow to describe data as a sequence of modifications to a known piece of data.

In Distributed Deduplication, we can use delta encoding when a file is already present on both ends, and a new file to be transfered can be described as a modification of that file. The file present in both ends is called the original file, or source file. The sending point, sends the sequence of modifications instead of the entire new file. The receiving end then takes the received modification sequence and applies it to the source file, obtaining a copy of the file.

These algorithms are used in source versioning softwares such as CVS[6] and GIT [9].

In order to find sources using this technique, we would have to generate a modification sequence for all files or pieces we suspect would be similar. First, that would require us to have some heuristic to detect potential similarity to choose which data we would apply delta encoding to. Then, we would need to download all those files or pieces, and generate the modification sequence for each pair. Not only would this be very hard to use, due to massive bandwidth costs, it would also require a very good heuristic in order to detect most similarity.

2.3.2 Compare-by-hash

Many of the most popular deduplication techniques fall in the *Compare-by-hash* category. This category warrants a much deeper look.

Compare-by-hash techniques make the base assumption that each block of data can be uniquely identified by a smaller, finite identifier. That identifier is called the *fingerprint* of a block of data. A Compare-by-hash algorithm starts by dividing the block of data into smaller blocks, if needed. These smaller blocks are called *chunks*. Then, for each chunk, the algorithm calculates the fingerprint of the chunk, and then compares it to all the fingerprints stored in the system. If the fingerprint matches a previously stored fingerprint, the algorithm assumes the block identified by the fingerprint was already present.

In storage deduplication, this is used to reduce the amount of data stored, by storing the block of data only once.

In the Distributed Deduplication case, after calculating the fingerprint, the data uploader sends the fingerprint to the receiving end. If the fingerprint was already present, that block of data is not uploaded.

Choosing a chunk's identifier

Compare-by-hash takes its name from the most commonly used function for calculating fingerprints, cryptographic hash functions.

Cryptographic hashes make good identifiers. The reason is that, by definition, a cryptographic hash has a very low random collision rate, or a high collision-resistance [7]. In the case of SHA-1, that collision rate is 2^{-160} for any given pair of chunks to collide, and the probability to find any 2 chunks with the same hash is 2^{-80}. That chance of collision is so low (lower than the chance of random hardware error), it is somewhat safe to assume that collisions do not happen.

A severe concern is raised in [24], to whom [7] was a response: if collisions do happen, the system becomes corrupted.

TAPER [25], a compare-by-hash system for reducing data redundancy in replicated server synchronization, addresses the problem by computing a second, different hash over the entire file after such a collision is found. Only if both match, is the chunk is considered identical. However, such attempts can only reduce the risk of corruption, not eliminate it [24].

Choosing a chunk's boundaries

When a Compare-by-hash algorithm receives a block of data, the algorithm will look at its size. If the fingerprint is calculated over a too large block of data, there will be very little redundancy detected. In that case, the block of data must first be divided it into smaller chunks. However, those chunks cannot be too small, or the overhead they introduce into the system may bigger than the advantages of detecting the redundant data.

Deciding on how to choose a chunk's boundaries (where does one chunk ends and another begins), is a popular study subject. Essentially, there are two different approaches: fixed-sized chunks and dynamic-sized chunks.

With fixed-sized chunks, a chunk size deemed appropriate is used for all chunks, and every chunk will end when it has exactly the expected size or the end of the data is reached.

In RSync [47], a popular file synchronizing protocol, chunk size was determined to be between 500 and 1000 bytes. RSync also allows for the last chunk to be of a different size, smaller than all the previously found chunks.

Fixed-size chunks have a problem, however. If a file has a single byte inserted somewhere, every byte after that has its offset increased by 1. As the chunks would still start and end at the same place, the chunks would have a different content, and the fingerprinting function would return completely different values.

This means that the insertion of a single byte causes all chunks after it to look different look like a completely different chunk, even though its contents are the exact same. Every chunk that starts from that byte onwards would then seem completely different from the ones in the original file, even though only one byte changed. For example , if a file with 100 chunks had a random byte inserted at the beginning of chunk 1, all 100 chunks would be different. If that file was 1GB in size, +1 byte difference would require the full 1GB of data to be transferred again.

Dynamic-sized were created to fix the above flaw. Their boundaries are not delimited strictly by size, but instead a function that tells when a chunk should end. The most common way to divide chunks dynamically is using content-defined chunking, as introduced in the Low Bandwidth File System (LBFS)[37].

In this approach, a sliding window is passed over the entire file, calculating a function over the last N bytes. When that function returns a desired value, the chunk's boundary is defined. In the LBFS file sysem, this limit is found when the last *n* bits of the value returned by applying a Rabin Fingerprinting [43] algorithm to the sliding window is equal to 0. Those n bits determine the average size of chunks in this system. In the LBFS case, the average expected size is 8KB, so n = 13.

To prevent chunks too small or too large caused by extreme cases, a chunk boundary must always be between a certain minimum and maximum threshold value (2KB to 64KB in LBFS). The boundary function is ignored until the minimum threshold is reached, and a boundary is immediately defined if the maximum threshold is reached.

Other approaches were devised based on LBFS chunking system, such as *fingerdiff* [8] or the *Two Thresholds Two Divisors* algorithm [17].

Storing a chunk's fingerprint

Another challenge raised by compare-by-hash is how to store and retrieve the chunk identifiers.

Taper [25] uses a tiered lookup approach, where it starts by matching the fingerprints of the entire directory trees and subtrees by means of an hierarchical hash tree, whose nodes are directories and leafs are files. Any file or directory to be stored that has a fingerprint match on that tree is already present, and needs not be stored.

If no matching file is found, the next step is to searches the chunk hashes, to try and find files that share content.

Finally, for a file that shares blocks in common, they run a delta encoding algorithm on the blocks that were unmatched. They then store only the necessary data to build the new file from the found duplicates.

This solution is very heavy in stored metadata, but reduces to a minimum the amount of data needed to be stored and transfered.

In Jumbo Store [18], hashes are stored in Hash Directed Acyclic Graphs, and graphs are themselves chunked in order to reduce the amount of data changed when one chunk is modified. Using this second

level of hashes, reduces the amount of data stored in the file metadata list. The second level also allows for slightly modified files to reuse part of the old metadata, saving disk space. This solution manages to share less metadata and redundant data, while managing to reduce metadata stored.

In the Data Domain Deduplication File System [50], the system first looks up the hash against a vector called a Bloom filter, that can quickly and safely predict if an fingerprint is not present in the system.

Additional considerations about compare by hash

Compare by hash algorithms are only efficient at detecting duplicate information if, when given the same block of data, the same set of chunks is produced. As such, the rules for creating a chunk must be the same for all the data where finding redundancy is expected.

If fixed-sized chunks are used, the chunk size must always be the same.

If using dynamic-sized chunks, the minimum and maximum boundaries must stay the same, and the function used to detect the boundary must always return the same value given the same input.

2.3.3 Applying Deduplication to P2P file sharing

Applying compare-by-hash techniques to P2P file sharing files is not a new idea.

In Gnutella, finding a file is done by querying the system over a set of terms, which are then used to select files that match that query [3]. Those terms are usually matched to a file's name. Two identical files with different names would never be seen as the same file, and could not be used as the same source, even though they were identical. However, an attempt to use hashes as URN in Gnutella was introduced in version 0.6 [27]. The HUGE plugin allowed to find more sources to download a file from by identifying it by its hash.

This approach allows for users of Gnutella to locate files that have the exact same content, however does not allow the users to download files that are very similar but not identical.

BitTorrent itself already uses compare-by-hash to identify torrents, to make sure torrents with the exact same content (the info dictionary) are not published by different users changing only the tracker list or the comments. This allows a user to join any tracker he knows and download that torrent content, regardless of where he got the torrent metadata.

Like the Gnutella example above, this allows users to find extra sources for identical files (or in this case, torrents). This method does not allow for a user to find similar torrents, which share much of their content.

In the area of compare-by-hash techniques applied to file chunks, most is applied locally to already downloaded files. BitTorrent only uses the piece hashes to verify the integrity of the downloaded pieces. However, borrowing from compare-by-hash, uTorrent, a very popular BitTorrent client, uses these same hashes to identify pieces that have already been downloaded by the user, by using a sliding window and comparing existing files in the download destination folder to the pieces to be downloaded.

However, not much work can be found in using compare-by-hash to find more sources for existing file chunks. The closest works we could find in that area were Partial Swarm Merger (PSM) [19] and Similarity Enhanced Transfer (SET) [42].

2.3.4 Partial Swarm Merger (PSM)

PSM [19] was a system designed in order to exploit the similarity already detectable in current BitTorrent. PSM's main objective is to provide additional sources in order to save Inter-ISP traffic.

In order to find the extra sources, PSM keeps information about the swarms that currently exist for that torrent, in order to allow peers to join all of them. PSM also detects and exploits similarity at a piece-level, using a compare-by-hash technique to the existing BitTorrent pieces. This allows users to download shared pieces from any torrent that contains them.

To start using PSM, a peer contacts the service by providing a magnet link for the torrent he wishes to download. The PSM service then looks at the infoHash present in the magnet link and adds all trackers in that link to the entry of that infoHash. If the infoHash was not present, the system will join a tracker (such as the DHT) and downloads the metadata via the metadata extension. Then, it would store that information in a database in a way that the pieces shared between torrents would have the magnet links of each similar torrent associated.

The PSM service would then reply the peer with all magnet links for each torrent found similar to the one sent.

Finally, the peer then contacts each of the trackers provided and joins each torrent, announcing itself as looking only for the pieces it shares in common, and downloading and sharing only those pieces from those other torrents.

There is one shortcoming detectable from analyzing PSM's design. The system uses pieces to find duplicates. As mentioned before, pieces are fixed-size chunks of a size defined by the torrent uploader. This makes it very difficult to find most existing similarity between torrents.

The wildly varying piece size makes it so that chunks of data that are similar may be masked by the chunk division. Two identical torrents, one with 4 250KB pieces and another with 10 100KB pieces could share 100% of their content, yet would not be detected because the piece hashes would not match.

Another problem is, with pieces being fixed-sized chunks, the chunks are subjected to the byte shifting problem mentioned in Section 2.4.2.

2.3.5 Similarity enhanced transfer (SET)

SET [42] was originally proposed in order to take advantage of the high similarity between files in P2P systems.

Pucha et al. defined a way to determine chunks, called *file handprinting*. A handprint of a file is defined as the first k sorted fingerprints of the chunks of a file. The handprint creation process starts by

dividing the file in variable-sized chunks using Rabin Fingerprints as delimiters. Then, the fingerprint for each chunk is calculated. Finally, the found fingerprints are sorted into increasing order, and the first *k* are retrieved as the handprint.

Handprints are used as an heuristic to determine if two files are similar to each other (similarity being the percentage of a file that is equal to another, second file). Two files are deemed similar if their handprints share at least one fingerprint in common. Handprints can detect files that have a similarity of up to *n*%, where *n* is determined by the *k* size of the fingerprint. The size k is a system parameter, with a logarithmic relation with n. With a *k* = 30, we can find files that have at least 10% in common detected, while to detect files with at least 5% in common requires nearly double that (*k* = 58).

To publish a file on SET, the peer must first calculate the fingerprints for every chunk in the entire file (each chunk with an average of 16KB). The generated list is the metadata for that file. The whole file hash is used as the Object identifier(OID) for SET. After generating this metadata, the user must order the hash list and form the handprint for that file. The handprint is then sent to the SET service, to the global lookup table. The global lookup table keeps a table matching fingerprints to sources. For each fingerprint on the handprint, the global lookup table will store the submitted file OID. The OID is then later used to locate the actual source of the file.

When a peer wishes to download a file from SET, it must first obtain the file metadata from somewhere. This is the same on BitTorrent, where the metadata file must be downloaded from somewhere before downloading the torrent.

To begin downloading, the peer must first recreate the handprint of that file, using the metadata obtained. For each fingerprint in the handprint, the peer requests the OIDs that contain that fingerprint to the global lookup table. The table will return the OIDs associated with each fingerprint looked up.

The peer must then request the source for all OIDs he finds interesting to the global lookup table. The table then returns the actual sources for each requested file. For each of those files, the user may then request and download the metadata from any of sources that has it. When the metadata is obtained, the peer must compare it to the metadata for the file the peer is interested to download. The peer may then download any chunks the peer is interested in from any source that provides it.

SET then uses a protocol quite similar to BitTorrent for downloading the files and verifying chunk integrity. The similarity between SET and BitTorrent is stated in the ”Design and Evaluation” sections of SET’s paper.

SET’s shortcomings

SET, while a a good solution, is not perfect, specially if we consider the BitTorrent environment.

First of all, SET requires all files to share the same average chunk size. While this allows for similarity to be detected across all files, file sizes in BitTorrent range from the small (less than 1KB) to the very large (more than 40GB). This generates a massive amount of fingerprints for the very large files, and

nearly no redundancy detection on very small files. Trying to solve either case would make the other end worse. In SET 's paper, its 16KB average size was defined to minimize overhead on their test set data, that was composed of files ranging from 1MB to 800MB.

Let's examine the very large file problem in more detail. The size of each fingerprint is 20 bytes, being a SHA-1 hash. Assuming a case where all chunks have their average size, the metadata size is given by Msize, while the ratio between the file size and the metadata size is given by p in the following equation:

$$Msize = 20 * \lceil \frac{fileSize}{ChunkSize} \rceil; \quad p = Msize * fileSize^{-1} = \frac{20}{chunkSize} \tag{2.1}$$

. For SET's 16KB chunks,p=0.001. This is small percentage-wise (0.1%). For a 1GB, it is about 1MB. This is not a bad size for a metadata set, if the user only needed to download it once. However, that is not the case in SET.

In SET a peer must download the entire metadata not only for their file, but also for each file the global lookup table returned an OID.

Let us consider a scenario where all files present in the SET system are the same size. A peer is downloading one such file from a source, and while downloading, that file appears to disappear from the network. The peer, that so far was satisfied with the file download, needs only 10 specific chunks to finish its file. The peer makes the file handprint, contacts the SET lookup table and finds 100 file OIDs. For each of those files, the peer must then download the metadata, which is 0.1% of the size of the download. This adds up to an overhead of 10% of the size of the file.

After searching each of the new file metadata, however, he found none of the chunks he required. The peer has just wasted significant bandwidth for no profit. If we consider all the files had a size of 1GB, the user would have wasted approximately 130MB, a very sizable amount for someone with a limited data plan.

This scenario is more likely than it appears. In the cases presented in SET's paper, movies and trailers that differed only in their languages (audio and/or subtitles) were found to share 15% of the data, enough to be found by SET. However, these same video files would not share 85% of their contents. That means that, on average, 85% of the metadata obtained was wasted bandwidth.

Chapter 3

Architecture

In this Chapter, we will describe our solution. We will begin by looking at how the user interacts with our system. This will be done by looking at the system from the user's perception, considering each piece of the system the user interacts with as a black box. Then we will be looking at every piece in more detail, looking at their internal architecture and the reasoning behind it.

We begin by remembering the problems at hand.

In BitTorrent, and many other P2P file sharing networks, users download files from as many different sources as possible, in the hopes of speeding up their download speed to near their maximum downstream bandwidth capacity. However, they are limited by the number of sources they can find.

In a different case, a peer may be in a position where downloading from far away sources may be troublesome or downright impossible, and finding a source nearby may be the difference between finishing a download or not.

Both cases mentioned above would benefit greatly from more sources found.

However, it is important that any solution found has minimal negative impact on the end user. The overhead introduced by this solution must be outweighed by the benefits of using the solution on an average case.

Our solution is based on the compare-by-hash class of algorithms. We will need to face the challenges from using these class of algorithms, such as how to manage the trade-off between higher redundancy detection and lower metadata sizes.

System composition

Our system is composed by several distinct pieces of software, called *Services*. A service may be hosted at a single computer or in multiple computers. Services may need to store data. Data stored inside a service may be stored in a conventional table or using a DHT, as long as the needed precautions are taken.

There are three services in this architecture:

- The *insertion service*, responsible for finding extra sources, for both chunks and files.
- The *lookup service*, responsible for keeping the information found about extra sources, and deliver them to any process that requests it.
- The *maintenance service*, responsible for making sure the lookup and insertion services are working correctly, and that the information in both these services is up-to-date and fresh.

Users of our system may be divided into two types. One type of user is the uploader, a user that wishes to insert files into our system. The other type of user is the common peer, that wishes to find new sources for the chunks it is downloading.

Uploaders only need to know how to contact the *insertion service*. Other peers only need to know how to contact the *lookup service*. The maintenance service cannot be contacted by normal users.

Our system is built in order to deliver *sources* of duplicate pieces of data to anyone that requests them. Sources are ID/offset pairs. What the ID is may vary depending on the type of source. For a file source, the ID is the torrent's infoHash. For chunks inside a file, the ID is the whole file hash, the *file ID*. The offset is the location inside the container where the block of data we wish to find is located. For a file, the offset would be where inside the torrent that file begins. For a chunk, the offset is where in the file the chunk begins. From this moment on, whenever we say sources we mean the above definition.

In our system, we try to identify and store information about all existing duplicate chunks and files in the network. This would allow for a peer to quickly check whether a chunk or file it is looking for has extra sources or not. Keeping all duplicate chunks and files presents an additional challenge, as the size of the BitTorrent network is immense.

Using the service: uploader

In order to use our system, the uploader must first create the metadata for the file or set of files he wishes to upload. This metadata contains chunk fingerprints, as well as the file ID. The entire metadata composition will be discussed in Section 3.1 in more detail.

Next, the uploader must send each file's metadata individually to the insertion service. The uploader sends a message to the insertion service containing the whole file node of that file's metadata and the source for that file (the torrent infoHash and the offset where that file starts inside that torrent).

The insertion service will reply to the uploader with a message requesting more information if the system had never seen that file before, or reply with a already present message if there was a copy of the file already present in the system.

If more information was requested, the uploader will send the metadata to the insertion service. The insertion service will use that metadata to identify chunks that are already present somewhere in the network, and make that information available through the lookup service. The way this chunk detection is done will be detailed in Section 3.3.

After the insertion is completed, the insertion service informs the uploader that the insertion has been completed.

Using the service: peer

A peer wishing to find a source needs only to send a single message to lookup service.

To find sources for a file, the peer only needs to send a message containing the the file ID to the lookup service. The lookup service will reply with a list of all known sources for that file.

If the peer has some of the metadata, and wishes to find sources for one of the chunks, the peer sends a message with the fingerprint of that chunk, and the lookup service replies with the sources for that chunk. Then, for each chunk source found, the peer may need to look up sources for the files that contain the chunks. That falls in the first case described.

There is a good reason all sources contain an offset. By having an offset where the a file is located, the peer only needs to join the torrent and download from the offset provided in order to obtain the file. In the chunk case, it is pretty much the same, only that the offset of a chunk in a torrent is the offset of the chunk inside the file plus the offset of the file inside the torrent.

The peer can then use its own metadata to verify the chunk, by calculating the downloaded chunk fingerprint and matching it to the fingerprint the peer was expecting. This requires no additional metadata download, saving greatly on overhead.

We will now look at each piece in more detail, starting with the metadata, and then following with the lookup service, the insertion service and finishing with the maintenance service.

3.1 Metadata

Our system uses compare-by-hash to find identical chunks to use as additional sources. This requires us to create a set of fingerprints for the chunks of a file. This metadata will then be used to identify the duplicate chunks. The peer will also need to have this metadata, in order to identify the chunks the peer is downloading, and to verify the integrity of the downloaded chunks.

From the start, one of our objectives was to minimize the load that metadata imposes on the peer. Traditionally, decreasing the size of the metadata is accomplished by increasing the chunk size. However, that makes it harder to find redundant data, and as such reduces the number of possible sources detected.

If we ignored the small metadata size requirement, however, we could instead maximize the number of sources found by choosing a much smaller chunk size. That will drastically increase the number of fingerprints.

Our objective is only to minimize the metadata size on the peer. The metadata size on the uploader is irrelevant to us. Ideally, we would distribute as little metadata as possible to the peer, while finding many new sources from the uploader's metadata. If we could provide the peer with a much smaller metadata, yet use the uploader large metadata to provide the redundancy detection, our problem would be solved. We could have the large redundancy detection with a small metadata size.

Metadata tree

Our solution was to use a multi-level fingerprint structure, where each level provides a different level of resolution.

The structure representing the metadata is a tree, where each node represents a chunk of data. The root of that tree is the whole file node, containing the file ID. Each level of the tree represents a *level range*. The level range is a pair of values that identify the smallest and biggest possible value for the size of a chunk in that specified level. The level ranges get smaller as the depth of the tree increases.

Nodes in the tree represent a chunk of a size inside the level range. The children of that node are the chunks resulting from splitting that node into chunks of a size inside the next level range.

In order to better understand this structure, we provide an example in the figure below.

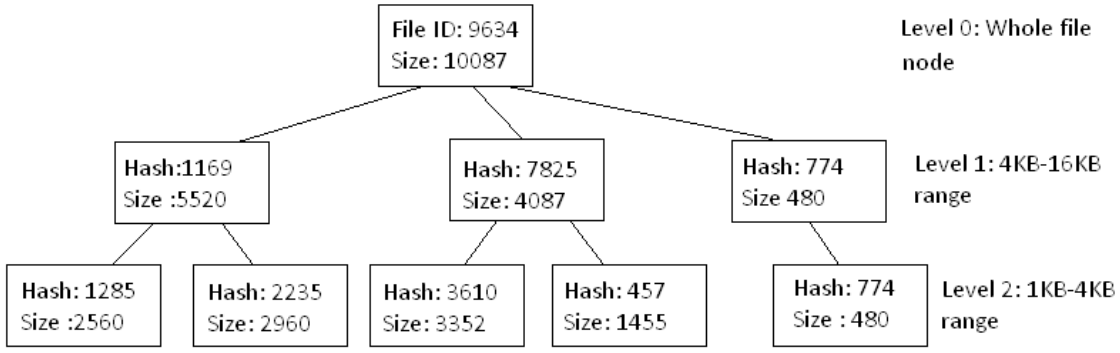

Figure 3.1: Example of a small tree

There are several reasons why we use this parent-children approach to chunking.

The first reason is that this structure allows a user to obtain more information about a chunk, by looking at its children. Imagine a peer does not find any extra sources for a node. With the parent-children approach, the peer can look at the children and search for sources for each of them. If sources are found, the peer will have extra sources for at least part of the chunk the peer originally looked for. In the best case, the peer may even find sources for all children, and will be just like having extra sources for the parent.

The second reason is that this structure allows for a more compact storage of sources in the lookup service and of nodes in the peers that download them. The storage in the lookup service will be discussed in Section 3.2.1.

The third reason is that this structure allows for peers to download only lower level nodes only when

interesting to them. Imagine a peer that needs more sources for a chunk. That peer knows the node has children, but has no information at all about the next level. With the parent-children approach, the peer only needs to download the children node. Without the parent-children approach, the peer would have to download the entire new level. He must do so in order to verify the validity of the newly downloaded level.

The parent-children approach at chunking has some minor problems. The chunks are created from the bottom up, and then joined together to form their parents. This causes cases where dividing by the given range generates smaller chunks than adding them together.

As an example, let us consider that running the chunking algorithm without minimum restrictions would generate four chunks, C_1 with 500B, C_2 with 350B, C_3 with 250B and C_4 with 750B. We have two level ranges, $\{750 - 1100\}$ and $\{1100 - \infty\}$. In both joining and dividing, the first level would be $C_{1,2}$ with 850B and $C_{3,4}$ with 1000B. The second level obtained by joining would have a single chunk,$C_{1,2,3,4}$, with 1850B. However, if they were divided from the start with the traditional algorithm as described in LBFS, the second level would be $C_{1,2,3}$ with 1100B and C_4 with 750B.

We believe, however, that this shortcoming does not outweigh the benefits gained by the parent-children approach. Finding a better way to join chunks together to form their parents is, however, something worth looking at in the future (see Chapter 6).

We will now look into the constitution of a node in this tree.

Metadata nodes

Each node in this tree is composed of four important pieces of information.

- the chunk fingerprint;
- the offset of the chunk in regards to its parent;
- the size of the chunk;
- the child hash.

Since chunks are created in a variable-size approach, chunks may have any size inside their level range. This information is important to allow the downloading peer see where a chunk ends, and allows the peer to verify the data they downloaded against the fingerprint for corruption.

The *child hash* is the hash of a block containing the chunk fingerprint and the child hashes of the children nodes. The only exception to this field are the last and before last levels. In the last level, there are no children, hence no child hash. In the before last level, the formula uses the chunk fingerprint of their children as the child hash of those nodes.

The child hash exists because our metadata tree is a Merkle tree[36]. Merkle trees provide protection to the tree's data against accidental change or malicious tampering. However, it also provides some useful side effects, which will be discussed further on.

Auxiliary metadata

Besides the tree, there are two more pieces of metadata that aid in the proper functioning of this system, the level range specification and the level hashes. Collectively, they are the *auxiliary metadata*.

The level range specification tell the peers that downloaded the metadata which are the level ranges for each level of the tree. The discussion of what is affected by a change in level range is discussed in Section 3.1.1.

There is only one fixed rule that must be followed when choosing the level ranges: the range for the level N-1 must start in a value equal or bigger than the end of level N. There are default value for ranges, which is starting at 1KB, ranges in the format $[2^n, 2^{n+2}]$ until a value close to the total size of the file.

Changing the metadata level ranges is dangerous. The same file, chunked using different ranges, will be vastly different from one another.

A *level hash* is a hash of a block containing all the fingerprints of the same level for a given file. This set of data allows a user to download an entire level and then verify the obtained nodes without knowing the rest of the tree. Each level, except for the whole file node one, has its own level hash.

Distribution

Metadata may be distributed between peers or obtained from an external location, such as a torrent indexing website.

A peer wishing to download the file, but not interested in searching for extra sources, may simply download the first 2 levels and use them as BitTorrent uses piece hashes, as a checking mechanism for corruption. This metadata could act as a replacement to the BitTorrent's metadata, and should be much smaller when torrents have multiple files. If the peer is interested in using the deduplication aspect of the system, it may then download the next level for each chunk he needs more sources for.

This is the major advantage of the parent-child approach. The peer only needs to download the children of each chunks the peer is interested in. If a peer does not need more sources for a particular chunk, the entire sub-tree belonging to that chunk is not downloaded. If a chunk node is pruned early on, that may have a very big impact on the amount of metadata downloaded.

The level hashes allow for a peer to download a level from the middle of the tree, knowing only the file ID and the level hash for the desired level. A peer may, from that level onwards, download the metadata as before, even without knowing about the previous levels.

The bare minimum the peer needs to use the system is the file ID. However, because a tree may vary wildly if the level range is changed, the Level Ranges and Level hashes are also required to correctly see the tree the uploader created. As such, we identify the metadata content using the hash of the block containing the level Ranges, Level hashes and the whole file node. This hash is called the *metadata ID*.

Downloading parts of the metadata dynamically in the future requires some extra precautions. In a real-world scenario, peers may not always be trustworthy. As such, it is important to provide some assurance to the downloading peer that the new piece of metadata he received is the one he requested, and was not altered in any way. This assurance is given by the level hashes and the child hashes.

As the data is distributed dynamically, it is possible that some sub-trees are never shared between the peers. If that happens, the sub-tree may disappear from the network entirely when the last peer that owns it leaves. However, due to the child hash, it is possible to recover those lost sub-trees. If a peer has the entire chunk that generates that sub-tree, the peer may recalculate the tree and distribute it. The receiving peers can validate the sub-tree using the child hash.

There are ways to prevent the metadata from disappearing from the network, by forcing the distribution of parts of the metadata to other peers. How to choose which data to be forced, and which to let disappear, is a question for further study.

The algorithm to create the metadata tree will be presented in Algorithm 1,in the analysis section. Besides that algorithm, a second step uses the created tree to generate the level hashes, by going through the tree and gathering the needed fingerprints.

3.1.1 Analysis

In this Section we will present a in-depth analysis of the metadata, starting with the actual structure and then the tree construction algorithm.

Metadata Space Analysis

We will now look at the space occupied by the tree. Each node contains four fields, as mentioned before. Both hashes (fingerprint and child Hash) are SHA-1 hashes, with 20 bytes each. For the size and offset of a chunk, we used a special number format, the formatted unsigned number format.

The formatted unsigned number format tries to reduce the number of bytes needed to store a number. This format stores numbers on the lower 7 bits of a byte, and uses the last bit to indicate if a number still needs more bytes to be stored. So, for instance, the number 127 is represented by $\{$0x7f$\}$, yet 128 is stored as $\{$0x80, 0x01$\}$. Using this format, the biggest size storeable in 4 bytes is $2^{28} - 1$, but 5 bytes can store numbers up to $2^{34} - 1$, nearly 16GB of size.

Not all nodes are the same. Nodes can be divided into four distinct kinds:

- File Node - the root node. It can have much more children than any of the other nodes types, and requires no offset;
- Top Node - the nodes closest to the Root. it may require a larger offset field than the rest of the nodes;

- Middle node - the most common type of node, requires slightly less space in the offset field;
- Leaf Node - has no child hash;

Each of this types of node has their own space requirements, shown below:

	Fingerprint	Child hash	Size	Offset	Child	Total
File node	20	20	6	0	4	50
Top node	20	20	4	6	1	51
Middle node	20	20	4	5	1	50
Leaf node	20	0	4	5	1	30

Table 3.1: Size of each type of node

The node sizes presented above make some assumptions. They assume no chunk is bigger than 256MB. They also assume that the level ranges increase in an exponential pattern, that is, all level ranges can be defined as [2^n, 2^{n+k}].

In order to round the values, the offset field was slightly enlarged on the middle and leaf nodes (which would have an offset, at most, the size of the offset field).

Before we begin, let us define *N* as the last level, 2^s as the size of the smallest possible chunk, that is, the minimum size for the range of *N*, and *size* as the total size of the original file. The general formula for the size of the metadata is as follows:

$$treeSize = 30 * \lceil \frac{size}{2^s} \rceil + \sum_{i=1}^{N-1} 50 \lceil \frac{size}{2^{s+ki}} \rceil + 51 * \lceil \frac{size}{2^{s+kN}} \rceil + 50) \tag{3.1}$$

This formula only applies, however, if $size > 2^s$, $N > 2$ and $k > 0$.

In cases where the size is less than 2^s, only the fileID is needed, as it is the only hash calculated.

For $N <= 2$, this formula does not apply because there is no middle level. Removing both middle fractions will give you the correct formula for that case.

For all other cases, the size of the metadata grows with the size of the file. The ratio of growth can be determined by dividing the above formula by *size*. To simplify the mathematics, let us ignore the ceiling functions. The ratio formula, after simplified, becomes as follows.

$$ratio_{treeSize} = \frac{30}{2^s} + \frac{50}{2^{k+s} - 2^s} + \frac{51 - 50 * 2^k}{2^{kN+s}} + \frac{50}{size} \tag{3.2}$$

Let us evaluate this function by taking its limit when the size reaches infinity.

Size is only present in the last fraction, and that fraction converges to 0. As all remaining values are constant, the entire function converges, and its limit is given by the remaining three fractions.

The value of the ratio as it size tends to infinity is then controlled by the size of the smallest chunk, the size of the range interval increments and by the number of levels chosen. Let us study each of those parameters individually.

Size of the smallest chunk

The size of the smallest chunk has the greatest impact, being present in all remaining fractions.

With the remaining parameters being identical (k = 2, N = 8),a chunk of 1024 bytes has a maximum ratio of approximately 4.56%, while a chunk size of 2048 has a maximum ratio of 2.28%.

With 32 byte chunk size, the ratio reaches close to 144%, meaning that the total maximum size usable in order to keep the size lower than the original file is 64 bytes, with a 72% ratio.

Level range increase

The change on level range increase (k) has a smaller impact than the size of the smallest chunk. In relation to 2^k, the function is inversely proportional, and has a limit as k tends to infinity of $\frac{30}{2^s}$. If we have s=10 and N = 8, we get a ratio of 7,67% for k=1, 4.56% for k=2 and 3.63% for k= 3.

It is important to note that the larger k is, the more children nodes level N-1 will have. Each node in level N-1 will have range between 2 and 2^k children, except for the last node in a level, that may have only one child.

The reason the minimum number of nodes is 2 is because each node of the previous level will have, at most level $N.rangeMin$ - 1 bytes, needing to add two at least to reach level $N-1.rangeMin$.

The reason the maximum number of nodes is 2^k is because, even if all children of level N have the minimum possible value, 2^k*level $N.rangeMin$ = level $N-1.rangeMin$, by definition.

Number of levels

The number of levels, N, is negligible for any N above 4. That because N is a multiplier to k. A change in either N or k has an exponential impact on the denominator, reducing its impact on the overall formula.

With s=10 and k=2 we have $\frac{251}{1024*2^{2N}}$, which for N = 4 is equal to 4.5% , and from N=5 onwards , the number rounds to 4.55%.

Formula limitations

This formula shows a very simplified, worse-case scenario.

For one, it assumes all chunks would be sized exactly the same as the minimum value for their level range. The previous assumption implies that all nodes will have their maximum number of possible children (explained below).

It also assumes all nodes would be sized exactly the same in each level, while the way they are stored should make children nodes vary from 1 to 6 bytes depending on their position inside the parent's children list.

As such, we expect the metadata size ratio on real files to be smaller.

Choosing values for s, k and N

After this look at the space occupied by the tree, let us go deeper into the subject of the choice of values for those constants. We begin with the level number, N.

As we have seen, the number of levels has very little impact on the total size ratio of the metadata tree. That does not mean N has no impact in other pieces of the system.

In the lookup service, the lookup of sources for a chunk may require N accesses to find all the sources for that chunk. That happens when the chunk is from level N, but a parent in level 1 had sources in the system. This is due to the parent chain method of storage in the lookup service, which will be explained in Section 3.2.

The maintenance service is also negatively affected by the number of levels, in all of its functions (see Section 3.4).

We recommend that N be smaller or equal to 10. However, the number of created levels should be chosen as a consequence of the level ranges chosen and the smallest chunk.

For k, we have seen that $k > 1$ provide size ratios of less than 5%.

Increasing k increases the maximum size a chunk may have, but also reduces the minimum possible size for every chunk before level N.If we take a tree starting at 2^s=1024, using k = 2 makes level $N-1$ start at 4KB, and level $N-2$ start at 16KB. With the same start, but using k = 4 makes level $N-1$ start at 16KB, and level $N-2$ start at 64KB. As such, it is beneficial to keep k lower in order to detect more redundancy.

As for values for k, we recommend k=2. We do so because the gain from wider apart levels is not as great as the increased advantage of having more intermediate levels to match. We also do not recommend 2 times the previous value, as the loss in that case is better put to use in a smaller, lower starting chunk size.

Finally, the smallest chunk detected (2^s),the most important parameter. We saw that the size doubles with each unit decrease in s. However, the bigger s is, the less redundant data can be detected. As such, we recommend 1024 bytes as the minimum chunk size.

Using 1024 as the minimum chunk allows for any file which size can be calculated by the equation to have at most under 5% of the original file size, while at the same time being a common low-granularity size for compare-by-hash chunks.

If more granularity is desired, we would recommend using 256 bytes at most, which has a 18% ratio, and while using a 4 times increase would allow for a N-1 level very close to those of the 1024 bytes level.

input : File to process, f;level ranges $range$, optional

output: root of the metadata tree

$counters \leftarrow$ `MakeArray(size(`$range$`),` *0*`)` ;

$nodes \leftarrow$ `MakeArray(size(`$range$`),` *tree nodes*`)`;

$byteToRead \leftarrow$ 1;

$startFileOffset \leftarrow$ 0;

while $byteToRead <$ `size(`f`)` **do**

    increaseCounters();

    while $byteToRead <$ `size(`f`)` **and** $byteToRead$ - $startFileOffset < range$*[*`size(`$counters$`)`*].*min

    do

        increaseCounters(); $byteToRead \leftarrow byteToRead + 1$;

    end

    if *(*`RabinFingerprintFindDelimiter(`f, $startFileOffset$,$byteToRead$`)` **and**

    $counter$*[*`size(`$counters$`)`*]* $>= range$*[*`size(`$counters$`)`*].*min*)* **or** $counter$*[*`size(`$counters$`)`*]* $= range$*[*`size(`$counters$`)`*].*max*)* **then**

        `addNewNode(`$nodes$*[*`size(`$counters$`)`*]*, $startFileOffset$,$byteToRead$`)`;

        for *i* $\leftarrow$ `size(`$counters$`)` $- 1$ **to** 1 **do**

            if $counter$*[i]* $>= range$*[i].*min **then**

                `addNewNode(`$nodes$*[i]*, $byteToRead$-$counters$*[i]*,$byteToRead$`)`;

                `addNodesAsChildren(`$nodes$*[i].last*, $nodes$*[i+1]*`)`;

                resetCounter($counters$, i);

                emptyNodes($i + 1$);

            end

        end

        resetCounter($counters$, `size(`$counters$`)`); $startFileOffset \leftarrow byteToRead$;

    end

    $byteToRead \leftarrow byteToRead + 1$;

end

$root \leftarrow$ makeWholeFileNode(f);

`addNodesAsChildren(`$root$, $nodes$*[1]*`)`;

Algorithm 1: Metadata tree generation algorithm; *addNewNode* creates the node, calculating the SHA-1 fingerprint for that chunk; *addNodesAsChildren* also creates on the adding node the childHash based on the children added

Metadata generation algorithm

The algorithm presented in the previous page is how this metadata is currently created. It uses the last n bits of a Rabin fingerprint to delimit chunks.

$$n = \frac{log_2(range[N].min) + log_2(range[N].max)}{2} \tag{3.3}$$

The algorithm reads the file only once, being in that regard O(n) to the number of bytes read. The main slowdown for this algorithm is due to the amount of cryptographic hashes being calculated. In order to reduce that time, hashes are only calculated when absolutely needed.

The Rabin fingerprint is only run after we have read enough bytes to fill the minimum chunk size for level N. After that minimum, the Rabin fingerprint is calculated over the sliding window for each byte until a match is found or the maximum is reached. Either way, after the chunk is set, the Rabin fingerprint is only calculated again after the minimum bytes were read again.

Only when a chunk is found is the SHA-1 fingerprint for that chunk calculated. The SHA-1 function is run once for level N nodes, and is run twice for all other types of node, once for the fingerprint and once for the child hash of that node. This means the function is called at most 2 times the number of nodes created.

This finishes the analysis of our system's file metadata. Creating a file's metadata is the first step towards allowing a file to be used in our system. Our metadata structure allows smaller chunks than even SET's metadata. Because users only need to download the first level to use the service, the metadata may be the same size or even smaller that the BitTorrent metadata.

3.2 Lookup service

The lookup service is responsible for storing the extra sources found by the deduplication process. The lookup service is also responsible for replying to any peer or service that asks for extra sources.

From a peer perspective, the lookup service receives a request containing either a file ID or chunk fingerprint for which the peer wishes to find more sources, and replies with a list of sources for that data. If no sources were found, the lookup service replies with an empty list.

In the case of a chunk fingerprint, we defined a source as a (fileID, offset) pair that indicates where we can find that chunk inside the identified file. For files, the source is a (infoHash, offset) pair, indicating the torrent that contains that file and where in that torrent the file starts.

In order to reply to lookup requests, the lookup service must also store the sources found by the deduplication process. The sources are provided by the insertion service, and the lookup service needs only keep this data safely stored.

In order to store the sources, the lookup service maintains two tables. The *file lookup table* stores sources that contain found files. The *chunk lookup table* stores sources for chunks found duplicate by the system.

In our design, the file lookup table simply stores the file ID and a list of sources. The chunk lookup table, however, has a much more complex design.

Chunk lookup table

The chunk lookup table must store the fingerprint that represents the chunk, as well as the list of sources to find that chunk in a file, called a *source list*.

We also have a second list of sources, each indicating the source for a chunk inside another chunk. This second source list is called a *parent list*. It represents the chunks stored in the table that had that chunk as a child in their metadata tree.

In order to reduce the size of the entries, sources were only stored on the entry for the largest of the chunks that contained that piece of data. That is, chunks with parents did not have a copy of the source their parents had.

The following expression states the rule for storing a source. With F as the source file ID, and $children$ as the set of children nodes to a node in the metadata:

$$F \in sourceList_C \Leftrightarrow \nexists_{C_P \in lookupTree} C \in children_{C_P} \wedge F \in sources_{C_P},$$

$$sources_{C_P} = sourceList_{C_P} \cap (\forall_{C_n \in parentList_{C_P}}, sources_{C_n}) \tag{3.4}$$

In order to keep the system working correctly, some nodes will be stored without any sources. However, those nodes must have at least one parent. We can separate nodes that have no sources from the nodes with sources, and create two distinct tables, the chunk lookup table, and the sourceless chunk lookup table. This division may increase the time required to find chunks with only parent sources, but allows to reduce the size of the table.

In the a chunk entry we can also keep a *children list* containing a list of the fingerprints of the children chunks for that entry's node.

Having a child list has several advantages. First of all, it allows us to find partial chunk sources, sources that only contained part of the chunk. We could provide those sources together with the chunk's own sources, in order to increase the peer's options. Another advantage is that this structure would make cleaning up the chunk lookup table much easier. This, however, will be discussed in Section 3.4.

One feature possible with the addition of children lists would be the possibility to partially recreate the trees for stored chunks.

The children list has its drawbacks, however. Keeping the extra information about the children can increase the space taken by a node considerably.

Peer lookup scenarios

A peer interacting with the service may require either a file source or one or more chunk sources. We will exemplify the interaction between the peer and the service using an example.

Peer P1 wants to find sources for file F1. P1 asks the lookup services for sources for F1.

The lookup service looks for F1 in the file lookup table, and replies with the list containing all the sources it could find.

Upon receiving the message, P1 may choose to look for more sources for the entire file. P1 first chooses a level k. That level is chosen by the peer, and may be the level the peer believes will be the most successful at finding sources, or simply the last level the peer has entirely downloaded.

P1 then sends the level, from chunk $C_{k,1}$ to $C_{k,n}$, to the lookup service, who follows Algorithm A2 to find sources:

input : a fingerprint of a chunk $cHash$
output: List of sources $sources$
Function `findChunkSources(`$cHash$`)` : $sources$ **is**
    $tableEntry \leftarrow$ `lookupChunkOnTable(`*cHash*`);`
    if $tableEntry = \emptyset$ **then**
        $sources \leftarrow \emptyset$;
    else
        $sources \leftarrow tableEntry.sources$;
        for $parent \in tableEntry.parents$ **do**
            $sources \leftarrow sources \cap$ `correctSourcesOffset(findChunkSources(`$parent.ID$`),`
            $parent.offset$`);`
        end
    end
end

Algorithm 2: Chunk lookup algorithm. The $lookupChunkOnTable$ function may search across multiple tables

After receiving these sources, P1 may decide that, for some specific chunks, P1 needs more sources. For those, P1 may then look up the next level. He may repeat this interaction until the final level is reached.

3.2.1 Analysis

We will begin by taking a closer look at the way the data is stored.

Size concerns

This thesis faced a dramatic challenge from the start: the size of the BitTorrent universe. BitTorrent contains 280 millions of files, with a total of 17EB of data. Any solution that tried to index that network would take a tremendous amount of space to store that information. So, whatever the solution we created, it would need to take as little space as possible per chunk.

We designed this solution with that problem always in mind. One way we reduced the amount of data needed to store was by storing in the lookup tables only fingerprints for chunks or files that have at least two sources. Another way we reduced the size was by trying to store the minimum amount of information on each chunk that would allow us to find all sources.

Parent chain method

We call the method used to store sources in the chunk table the *parent chain* method. This method allows us to save space while still finding the same number of sources.

The key insight behind it is simple: if a chunk is present at one source, its child chunks are also present at that source. Therefore, we can use a parent node source to calculate a child node source.

As such, in the parent chain method, a source is stored only in the entry corresponding to the biggest chunk that contains the data. The children of that chunk share the parent's sources. Their offset inside that source will be equal to the sum of the offset stored in the source with the offset where the chunk is located inside the parent.

If we did not use the parent-chain method, whenever a new source was found for that parent chunk a new entry would also appear on all their children, all the way down that chunk's sub-tree. If a top-level chunk with 5 levels below it, and each level filled with children (4 per node), a single source found for the top-level chunk would cause 256 inserts, and take 256 times the space our solution has.

The parent chain method has its drawbacks as well. First, this solution only has considerable savings in space if there are more than two sources in the top level nodes. Otherwise, the space taken will be roughly the same without using the parent chain method.

The second, and more serious drawback, is that the time it takes to lookup a single chunk's sources is increased from 1 table lookup to N table lookups, being N the total number of parents a chunk has. However, due to the dimension of the BitTorrent dataset, we believe this trading smaller storage size by access time to be acceptable.

Split between chunk-level and file-level redundancy

We make a clear distinction between chunk-level redundancy detection and file-level redundancy detection. Even without mentioning how duplicate chunks and files were stored, we mentioned that files and chunks are separate and treated differently. There are many good reasons for such a division.

The first, and most obvious one is that chunks are not files. Users may be interested in downloading files, but they are looking for chunk sources only to be able to download files.

The next reason is that cross-file chunk redundancy exists regardless of how many copies of a file are present, and where those copies are located. Extra file sources, however, are dependent on the network distributing the file. This separation allows for the redundancy detected between files to remain unchanged regardless of the number of times that a file is uploaded to (or removed from) the network.

The storing of files and chunks separately also saves space, for the same reasons as presented in the parent-chain method. If we had to store the infoHash of every chunk, for each new copy of the file, all chunks would have to be updated. Separating file and chunks allows for new files to appear in new torrents at any time, and cause minimal impact in the system. Also, as files don't have parent chunks, that field may be discarded in the file table.

The maintenance service needs to periodically check the existing files to see if they still exist. If files were not separate from chunks, the maintenance service would have to make that check for each chunk instead, which would be much more costly.

Compatibility with other file-sharing systems

One consequence of distinguishing between file sources and chunk sources is the ability to use the same Lookup service to serve multiple P2P networks. Our system is largely *network agnostic*, as it does not need to know which network the files came from. In fact, the only piece of the system that cannot be totally network agnostic is the maintenance service, which connects to the network the file is in to check for its availability.

If we were to deploy our solution in Gnutella, it would work as well as in does in BitTorrent. File sources for Gnutella would simply include the file URI instead of the infoHash from BitTorrent.

This network agnosticism allows us to even find cross-system file similarity, and use that similarity across multiple P2P networks. On the best case scenario, a peer that can use several P2P networks to find and download a file would be able to take advantage of our system to find much more sources than a single P2P network using peer.

Another effect of this system agnosticism is that the same chunk table can be used by multiple, different file tables.

It would be possible to create a shared resource for all networks, containing an index of sorts for all duplicates, and when a file is introduced in a new network, that file could benefit directly from any chunks found before it.

Messages in the lookup service

The lookup service replies to two different sets of messages: one set is free for everyone to use and the other requires some form of authentication.

Any message that only looks up data, such as file and chunk source lookups, is free to use. As they do not affect the internal state of the lookup service, they require no authentication.

There are 8 free to use messages, all lookup requests. There is a lookup file and a lookup chunk message.

Both messages have a single-lookup variant and multiple-lookup variant, that allow to look up multiple chunks or files at once.

Each of those messages has two variants, based on the type of response expected. The source reply variant requires the service to send a list of sources, while the boolean reply mode requires the service to reply with a bit-packed array containing a 1 bit for every identifier found.

The boolean reply is used mainly by the insertion service, which requires it for its insertion process, but can be used by any other users.

Authentication-only messages are messages that can alter the lookup service's tables. The only authorized users are the insertion service and the maintenance service.

The ”Find and insert/update” messages send a file or chunk node and one or more sources. If the lookup service finds the node identifier in the table, the sent sources are added to that chunk's (or file's) source list. These messages reply either with a bit-packed array containing 1 bits for each update that took place, or with that bit-packed array and the list of the sources that were already present.

The second kind is the ”Insert” type message, which forces the lookup service to insert the given file or chunk node into its table. The message contains the node to be inserted and respective sources. Insert messages also accept entire sub-trees as the node. If a sub-tree is sent, all the nodes are inserted, with the top of the tree containing the source list, and the remaining nodes having a parent reference.

Finally, the ”Delete” messages are sent by the maintenance service when one or more nodes are no longer useful, and must be erased.

Security concerns with the lookup service

The lookup service is the only piece of the system where the content is supposed to be permanent. A node, after inserted into a lookup table, is considered real. That entry will be used as a reliable source

by all the peers that connect to the system.

Information in the lookup service is considered safe from loss while relevant, and is irreplaceable if lost. As such, special care must be taken with who can manipulate the data inside it.

Only trusted entities are allowed to manipulate the lookup service's data. In our design, only the insertion service and maintenance service are considered trusted. We currently do not support for a peer to report new information to the lookup service directly.

In the design, we state that data manipulation messages for the lookup service must be authenticated in some way. We did not specify, however, any form of authentication to be used between entities in our system.

To deploy this solution in the real world, security measures should be added to minimize the impact of malicious users. However, it is out of the scope of this thesis develop any solution in this area, and leave this for future work.

3.3 Insertion Service

The insertion service is responsible for finding duplicate data and inserting the results into the lookup service.

The insertion table contains two tables, a *file insertion table* that contains information about files that were submitted to the insertion service,and a *node insertion table*, that contains the fingerprints of nodes submitted to the insertion service that have no duplicates so far .

The insertion process starts when an uploader decides to upload a file, and sends the metadata root node with the source where that file will be located.

After receiving the root node and source, the insertion service will check if the file was already present. If the file was present, the insertion service sends the new source to the lookup service and finishes the insertion. Otherwise, the insertion service will require more metadata from the uploader.

The insertion service may request the uploader sends the entire metadata at once, or that the upoader sends the metadata one level at a time. Algorithm 3 shows how insertion is handled. It will follow the case where levels are sent one at a time.

Function description

The following section will describe the workings of eah important function defined in the algorithm.

The *findAndInsertChunksOnLookup* sends a ”FindAndUpdate” message to the lookup service, and removes from the level list all chunks that were reported found by the lookup service. The way the insertion is made was already described in Section 3.2.1.

```
Input          : client contact address, fileID the file to be inserted
Instance Data: levelList : ordered set of chunk fingerprints
Instance Data: fileList : collection of file IDs
Instance Data: recentlyInserted, insertedChunks : set of chunk fingerprints with sources
Instance Data: notInserted : set of chunk fingerprints
if lookupFileID(fileID) then
    end insertion
end
level ← 1;
levelList ← requestNextLevelFromClient(client, fileID);
while levelList ≠ ∅ do
    fileList ← fileList + findAndInsertChunksOnLookup(levelList);
    insertedChunks ← findDuplicatesOnNodeInsertTable(levelList);
    insertDuplicatesOnLookup(insertedChunks);
    recentlyInserted ← recentlyInserted ∩ insertedChunks
    notInserted ← notInserted ∩ levelList;
    insertNodesOnNodeInsertTable(levelList, level);
    levelList ← requestNextLevelFromClient(client, levelList);
    level ← level + 1;
end
if notInserted ≠ ∅ & fileList ≠ ∅ then
    for file ∈ fileList do
        fileNode ← getFileInsertNode(file);
        if fileNode.smallestHanprintEnd ≠ largestHash then
            newSources ← requestDuplicatesFromUsersWithFile(file,
            {∀ x ∈ notInserted ∩ recentlyInserted ∧ x >= fileNode.smallestHanprintEnd});
            insertDuplicatesOnLookup(newSources);
            deleteNodesFromNodeInsertTable(newSources);
            recentlyInserted ← recentlyInserted ∩ newSources;
        end
    end
end
for node ∈ recentlyInserted do
    subTree ← requestSubTreeFromClient(node);
    insertTreeIntoLookupTable(subTree);
end
```

Algorithm 3: Insertion algorithm. Both $find$ functions modify the level set by removing the found chunks.

The *findDuplicatesOnNodeInsertTable* function does what its name implies. It finds duplicates using the node insert table. Each fingerprint found in the table is removed from both the table and the level list. Then a ”Insert” message is generated containing the fingerprint, the source for that file and the source contained in the removed table entry.

The *insertDuplicatesOnLookup* function sends the previously generated ”Insert” messages to the lookup service for each chunk recently found duplicate.

The *insertNodesOnNodeInsertTable* function inserts each not found node into the Node Insert table. It finishes by creating or updating the file entry in the file insertion table.

Handprint-based insertion

After all nodes are inserted in either the lookup or insert tables, a new phase of the algorithm starts. Before we discuss this in more detail, we need to understand the reason why this second stage exists.

The amount of data needed to be stored by the insertion service, in order to keep all fingerprints for all files in BitTorrent, would be gigantic.

To try and grasp the magnitude of the size the table could take, let us assume each entry in that table would have approximately the size of a node from the metadata tree. We could approximate very roughly the size of the table to the size of the metadata generated for a file of 15.3 EB. With the ratio being 4.56% of the original file size, that would be approximately 714TB.

It is very possible that, at any given time, we would not have space to store new data.

In order to make space, we should erase old, useless fingerprints. However, we have no way of knowing which fingerprints are, in fact, useless. A fingerprint that was never used may be needed in the near future to detect a duplicate. If we choose an intelligent way to delete hashes, however, we can minimize the losses.

Looking at SET, we find a promising heuristic to determine if a file has chunks in common with another, the file handprint. Based on the way handprints are created, we can choose some nodes we can delete and recover at a later date if required.

We delete nodes that have the biggest fingerprints. We then store the biggest remaining fingerprint for that file in the file insert table. That biggest fingerprint is called the *handprint end*.

Later on, if we suspect a chunk have been one of the deleted nodes, we can request the metadata for that file from someone that has it, and check our hashes against the deleted fingerprints.

The second part of the algorithm checks if a file might have some more fingerprints in common, for every file that already shared at least one fingerprint.

We accomplish this by selecting a subset of the recently inserted or not found nodes where the fingerprints are larger than that file's handprint end. If there are any fingerprints that match that, the *requestDuplicatesFromUsersWithFile* function comes into play.

In this function, the insertion service starts by contacting someone that may have the metadata. Peers from the network the file is shared, or metadata distribution sites are possible sources.

The service may then either request the entire metadata and search for the data by itself, or request that the metadata provider sends information about the fingerprints being looked for. In either case, the insertion service will receive a group of nodes. After validating the findings each new chunk found is added to the lookup service.

Inserting whole sub-trees

The third phase of the algorithm is in charge of inserting all child nodes for each newly found duplicate chunk.

The *requestSubTreeFromClient* function requests to the client the entire sub-tree for each node inserted into the lookup service, one at a time. After receiving the sub-tree, the insertion service sends a "Insert" message to the lookup service. The lookup service will then place the received sub-tree in a parent chain.

Table entries

In order to minimize the size of the node insert table, due to the massive number of entries it might reach, each entry contains only the fingerprint for a chunk, its source and the level it came from.

The file insertion table contains much more information, however. Besides the fileID that identifies the entry, the entry must also contain a timestamp (for the cleanup process, see Section 3.4), and a small table containing one row per metadata tree level.

Each row in the small table contains the total number of fingerprints of that file's level still present in the node insertion table. and the handprint end for that level.

The small table can also contain other fields, such as the level hash and the level range. Storing the additional metadata may reduce the time it takes to find fingerprints that were deleted from the node insertion table.

The information stored in the node insertion table can be considered volatile, and can be erased at any time. However, the loss of all data in the node insertion table would greatly reduce the amount of duplicate data that could be detected.

3.3.1 Analysis

Detection of intra-file redundancy

The system is capable of detecting intra-file redundancy. When the level list is created, we can detect chunks that share the same fingerprint in the same file. The system now has two sources for that chunk inside the same file.

The decision of storing this type of redundancy in the lookup service is a difficult one.

Detecting these chunks is beneficial to the peer. A peer only needs to download one copy of that chunk in order to finish the download. After downloading the copy, the peer only needs to copy that chunk to all the locations in the file where it would appear.

Storing this type of redundant chunks, on the other hand, may not be beneficial. On a worst case scenario, we may be storing chunks in the chunk lookup table that have no sources outside the file they were found in.

This type of redundancy could be easily detected by the peer, or even be detected during the creation of the metadata. However, storing this type of redundancy would ease the load on the peers. We believe the best solution would come from storing the information about internal redundancy in the metadata.

Detection of padding chunks

A problem related to he internal redundancy detection above, detecting padding bytes is beneficial to the peer, but storing information about those chunks may not be beneficial to the system.

Padding is used to trim a file to a certain size, or to align a piece data to a certain boundary. Padding is commonly present in disk image files such as .iso rips of CDs and DVDs, where unused space is treated as if it was useful data.

The supposedly empty space is then filled with a set of irrelevant bytes, commonly 0x00 or 0xff. These sections of data, while completely useless to the peer, are required to finish the download, and are very common across the network.

Detecting the chunks that contain only padding bytes provides the system with a number of chunks that are the same across many files, yet say nothing about the similarity between the contents of that file. Two files may share the same padding bytes yet share nothing else in common.

When the peer downloads sources for the padding bytes, the peer would be flooded with sources and could finish to download those useless bytes in a very short time. However, storing these chunks would make the handprint-based detection much harder, as many files share the same padding bytes, yet that does not mean they share any other chunks in common.

Solving this problem was out of the scope of this thesis. We leave that for future work.

Detecting chunks from handprints

Obtaining old metadata

During the second phase of the insertion algorithm, the insertion service takes all files that had something in common and check their handprint end to decide if those files may have some of the chunks that were not yet matched inside their metadata tree.

If that happens, the insertion service has two choices in order to check for those chunks. Either the service downloads the metadata and checks for itself, or the system request to peers that have the metadata to discover those chunks on the insertion service behalf.

Downloading the metadata may be expensive, but will always produce 100% reliable results. Doing it will also allow the insertion service to refill its Insertion tables with fresh data for that file, in case the data was destroyed (either accidentally or intentionally).

Delegating that task to a peer will reduce the amount of data downloaded. However, peers are unreliable, and many things can go wrong with this approach.

For instance, the peer may refuse to fulfill the request. The peer may also reply that he does not have the metadata needed to reply. In a worse case, the peer may provide wrong information on purpose, such as reporting that the chunks as not present when they are. For leaf nodes, the peer may even say the node exists. Without any way to confirm it, this action would introducing garbage into the system.

For any peer interaction, the insertion service cannot be 100% sure about the received replies.

One way to increase the reliability of this method is to request from multiple peers, and use the most returned value as the right answer. This solution would provide additional work for both the insertion service and for the peers involved.

The best alternative is to use both solutions, but choose carefully when to use each.

For a small number of chunks, delegating to the peers and then validate their findings is probably the best choice. The weight of downloading the entire metadata far outweighs sending a few small requests and the posterior validation.

For a very large number of missing chunks, downloading the metadata would be the best alternative. The overhead caused by sending the many fingerprints to multiple peers, and then receiving a large set of messages containing the results could be greater than the cost of downloading the metadata once again.

Deciding the best way to use each approach is something we leave for future work.

Increasing the speed

The process of detecting chunks using the handprint is much slower than if all fingerprints ever found were stored in the node insertion table.

The handprint detection method needs to either download the data or delegate to some peers. Either interaction takes much more time than simply checking the database.

In order to reduce the impact of the Handprint-based insertion, we would like to reduce the number of fingerprints we have to check outside the insertion service.

By storing the auxiliary metadata from the metadata file (the level hashes and level ranges), we are able to extract some information that can reduce the number of fingerprints to check per file.

The level ranges allow the system to check in which level a chunk could be found.

If a file has no range that matches the chunk's level range, the fingerprint for that chunk is removed from the list of chunks to check.

If a file has that range, but the handprint end for that level range is bigger than the fingerprint for that chunk, that fingerprint is removed from the list of chunks to check.

For files where a specific level has a small handprint end, we can download the entire level from a peer. Without level hashes, we would be forced to download the entire metadata. Downloading only one level reduces greatly the bandwidth spent on metadata downloads.

Fault tolerance

The insertion service has a some fault tolerance regarding the loss of its internal data, such as the loss of the node insertion table.

Let us consider the extreme case, where the entire data stored by the Insertion service was lost.

The insertion service tables would have no entries, yet would still be able to detect some redundancy based on the contents of the lookup service.

Upon receiving the first file, the first phase of the algorithm would only detect duplicate chunks that were present in the lookup service already. However, no other similar chunks would be detected.

When we reached the second phase, the insertion service would look for all files that shared at least one chunk in common with the file. Any file not present in the file insertion table is treated as if it has the smallest hash possible as the handprint end. As the data was lost, all files are considered as possible sources for all remaining chunks.

The insertion service would then download all the metadata files for all files that shared a chunk in common, and insert them into the database, starting to rebuild both the node insertion table and the file insertion table, increasing the chances for duplicate detection.

Additionally, the maintenance service tries to look out for signs of massive data loss, and tries to repopulate the table based on the reverse criteria of its node insertion table cleanup algorithm (see Section 3.4 below).

3.4 Maintenance service

In any real-world system, there are always some resource limitations. This resource limitations may be the capacity of the system, or time constraints.

Due to the size of the data our system deals with, it would be very expensive to keep all data from the metadata. In order to keep the system in working order, we designed the maintenance service.

The maintenance service has two major functions.

The first major function is to make sure the sources reported in the lookup file table still exist.

The other major function is to make sure the insertion tables do not reach limit capacity.

Besides the major functions, the maintenance service also performs smaller jobs, such as checking the system for major faults, such as the complete erasure of the tables of one of the services.

Usefulness of an entry

Before we continue, let us categorize the entries stored in the system in three distinct categories.

If the contents of an entry allows to find sources directly, that information is categorized as *useful*.

If that information is used by the system to detect sources in the future, that information is *potentially useful*.

If the information cannot be used to detect new sources, that information is *useless*.

The main job of the maintenance service is to safeguard the useful information, discard useless information, and be able to detect when useful information becomes useless.

Lookup service maintenance

All information stored by the lookup service is useful when it is stored. However, as time goes by, files are erased from the network they resided in.

If a torrent has 0 peers, it is impossible for any peer to get a chunk from that file. The maintenance service may discover that by contacting trackers and requesting the peer list or even the statistics for that torrent. In that case, that source is now useless. If all sources for a file are useless, we can consider that the file entry in the lookup service for that file is now useless.

If a file no longer has any sources, then any chunk that used that file has now one less source.

If that source was the only source for a chunk, and that chunk has no parents, that chunk is now useless. If that chunk has one source alone, that chunk became potentially useful, and should be moved to the node insertion table.

It is the maintenance service job to detect the loss of a file source, and delete that source. If a file entry has no sources, it is the maintenance service job to delete that entry and any chunk's sources that contain that file ID. The algorithm for finding these cases is presented below.

```
for fileID ∈ fileLookupTable do
    for source ∈ getFileSources(fileID) do
        if not isFileAvailable(fileID,source) then
            removeSource(fileID,source) ;
        end
    end
    if getFileSources(fileID) = ∅ then
        deleteFileEntryFromLookupTable(fileID, fileLookupTable);
        removeFileFromAllChunkSources(fileID) ;
        deleteFileFromInsertTable(fileID) ;
    end
end
```

Algorithm 4: Lookup table maintenance algorithm

The *removeFileFromAllChunkSources* function does two main services. First, it searches the chunk lookup table for any entry that has a source containing the removed file ID.

All chunk entries with less than two sources are removed from the chunk lookup Table.

All chunk entries that had now only one source are sent to the insertion service. The insertion service then decides if they should be kept (are smaller than the handprint end) or should be discarded.

The *deleteFileFromInsertTable* removes from the insert table all references to the deleted file. The maintenance service starts by deleting all chunks that have the deleted file ID as source. Then, the maintenance service deletes the file insert table entry for the deleted file.

Triggering the lookup maintenance

One important question to ask is when to activate this function. Checking all the file's sources all the time would be too expensive. One possibility is keep a timestamp in each source that is used to determine if the maintenance service should check the file. When the timestamp expired, the system would run the check on that source.

Another important question is when to delete the source. A file may be temporarily unavailable for many reasons, such as temporary tracker downtime, or decrease in availability inside the torrent due to

peers leaving. To face this problem, we can keep a counter next to the timestamp, counting the number of times the check failed. After a predetermined number of failures, the source is considered dead, and removed.

Node insertion table cleanup

The second major function of the Maintenance server is to keep the node insertion table smaller than a certain size. In order to accomplish that, we need to delete some nodes. However, we need to carefully select which nodes are going to be erased, in order to keep the insertion service working the same way.

All nodes in the node insertion table are potentially useful. Some nodes may be used to detect a chunk in the future, but most will not. As we have no way of knowing which nodes will be useless and which will become useful, we cannot simply delete the ones we will not need.

Storing all potentially useful nodes is almost impossible for any real world system. As such, we must select the nodes we will not store in a way they may be potentially recovered.

We have already mentioned the solution in Section 3.3. However, let us go over some of the main concepts again.

We need the nodes to detect redundancy. If we delete nodes, we lose our capability to detect redundancy. However, if there is a way for us to compensate for those lost nodes, we can delete a carefully selected few.

In SET, the use of handprints drastically reduces the amount of chunks needed to be stored. Handprints provide similarity detection between files up to a certain percentage, with only a fraction of the data.

However, using handprints introduced some problems, such as not being able to detect the location of specific chunks and needing to download the entire metadata to discover which chunks were in common.

The insertion service can download the metadata every once in a while to find extra redundancy for that file. The insertion service may also have some willing users provide that information at a later date.

This means some nodes can be deleted somewhat safely, the nodes after the end of a handprint .

Using the handprint will increase the message overhead and time taken for our insertion process. Also, some chunks may become impossible to find after deleted, if some of the metadata is lost for all peers. There is also the possibility that some files share only chunks greater than handprint end, and those duplicates would be impossible to detect.

As such, we would like to use this solution as sparingly as possible, and not blindly apply it to every file present in the system.

For this solution to work, we must first select a subset of files to which we will apply this solution.

Considering all of the above, the algorithm for the cleanup function is as follows:

Input: percentage chunks to be deleted, P
for $fileID \in selectedFiles$ **do**
    for *i* $\leftarrow$ *1* **to** `totalNumberOfLevels(`$fileID$`)` **do**
        $deletedCounter \leftarrow$ `deleteNodesFromFile(`$fileID, i$`)` ;
        `updateFileInsertTable(`$fileID$`, getBiggestChunk(`$fileID, i$`),` $deletedCounter$`)`;
    end
end

Algorithm 5: Node insertion table cleanup algorithm

Choosing the subset of files

We will apply this solution to the least recently accessed files. There are several reasons for this choice.

The intuitive reason is that, if a file did not find any duplicates up until now, it is less likely to find some in the future. Either there are no duplicates out there, or they are not that common so far.

Another reason has to do with the circumstances a file gets re-uploaded (with or without slight changes).

It is more likely for a file to be re-uploaded during a short period of time after a file first appears in the network. There are many reasons why, such as competing groups launching the same content for increased reputation, or people applying regional changes to the file (such as adding subtitles or translating a program).

As such, deleting older nodes will have a lesser impact than deleting fresh nodes that were just inserted.

Triggering the cleanup

As we would like to use this function as sparingly as possible, it is important to choose when the function is triggered.

Our approach was to trigger this function whenever the insertion service tables are getting too big. What can be considered too big depends on where the insertion service was deployed, and even how the insertion tables are implemented.

As such, we believe that choosing when to trigger the cleanup will vary greatly with the implementation of the service, and a case for studying in future work.

3.5 Integration with BitTorrent

Up until now, the system definition allows our system to work for any P2P files haring network. However, for a peer to reach the system, we need some client that can connect both to our system and to the P2P file sharing network.

As the objective of this project is to detect and exploit redundancy in the BitTorrent network, we have designed a way for BitTorrent clients to fully take advantage of our system.

This Section is not on how to make the client communicate with our system. The client communicates with our system using TCP/IP messages that were defined in the previous Sections.

This Section is about integrating our system with BitTorrent, through the design of a BitTorrent expansion. This expansion would allow the peers that recognize our system to trade the metadata bits and reduce accesses to the lookup service.

A client using this expansion would announce itself to its peers in the same way with any other expansion, by sending his expansion dictionary.

Expansion messages

Request known duplicates

The peer may send messages to request known duplicated chunks. Other peers in the network may reply with a list of chunks known to be duplicate. The replying peer would send a list of nodes and sources (as torrentID-offset pairs), as well as a timestamp from when the peer had received that message.

This message should be the preferred method for obtaining a starting set of duplicates, instead of contacting the lookup service right away. This should reduce the number of requests to the lookup service, reducing the lookup service's workload.

Request metadata

The peer may also send messages requesting for metadata.

In this message, the user sends a chunk fingerprint or file id to any peer, indicating the maximum depth of the response sub-tree. That peer may reply with a "don't know" message, or reply with the requested node's sub-tree.

The peer may then confirm the received data by comparing the parent node id and childhash to the ones the peer own, and calculate the child hash based on the children sent.

If satisfied with the result, that sub-tree is added to the peer's stored metadata tree. Otherwise, he may request it again, either from the same peer or from a different one.

If the peer suspects the replier sent wrong information on purpose, the peer may block the replier from further interactions.

This method may be used to obtain the metadata tree from other peers instead of having to download it beforehand, much like the metadata expansion in BitTorrent. It is essential to allow for the variable-sized trees mentioned before in this work.

Other similar messages allow for the download of the core metadata (level hashes, level ranges and root of the tree) and entire levels.

Rebuild metadata

If a peer only receives ”don’t know” replies to a certain metadata requests, the peer may assume that sub-tree was lost. On that case, the peer may try and contact a peer that owns the chunk and request that peer to rebuild the metadata.

The rebuild metadata message is sent to the peer that has the chunk, and contains the fingerprint for the chunk that needs new metadata. If the peer accepts the message, the peer recalculates the chunk’s sub-tree using the metadata generation algorithm applied to that chunk.

The peer then replies with the metadata calculated, and keeps the sub-tree stored temporarily in the case some other peer might request it.

Common message replies

All of the above requests may be replied with either a ”Will not reply” or a ”Will not reply this one” message.

The ”Will not reply” message means that peer does not reply to those kinds of requests, and its response will contain a list of messages he will reply to if capable.

The ”Will not reply this one” message means that the peer is not interested in replying to that particular message. The peer is capable of replying to those kind of requests, but is unable to reply right now or in the near future.

The peer may refuse to reply because it is busy, or because it has reached a limit of messages that the peer will reply to of that type. In any case, receiving this message means the client should try to request the information he needs from elsewhere.

Local lookup table

Besides the messages, a BitTorrent client should have a local lookup table, that may be used to reply to other clients.

If a client has multiple torrents downloading at the same time, having this table would allow to detect redundancy between them, and reuse common chunks from one file into another.

Delivering the metadata

In order for our system to work, the metadata must be exchanged in a safe way. Our suggestion is to include the core metadata (or only the core metadata hash) inside the path section of the info dictionary in the .torrent file.

Chapter 4

Implementation

We were unable to implement the entire project as described above due to time constraints. We fully implemented the metadata, lookup and insertion services. Only one function of the maintenance service was implemented, the node insertion table cleanup. We did not implement the BitTorrent extension.

The entire project was coded in Java. All data structures specific to the project were encoded in a binary format that would minimize their size. The structures include the metadata, the file and chunk sources and the messages exchanged between services and between service and client.

All service tables were stored in a MySQL database, with most values stored in the tables being Binary or Blob in format.

Most numeric values in this project were stored as formatted unsigned numbers, as defined in Section 3.1.1 The only numeric values stored in a different manner were the timestamps. Timestamps were 64 bit integers that represented time in the UNIX time format.

4.1 Metadata

The metadata implementation followed the definition presented at Section 3.1, taking into consideration the findings of Section 3.1.1.

The metadata is composed of an array of level hashes, a set of level ranges and a root node containing the file ID. The root node (and its children) compose the metadata tree.

Metadata tree nodes are encoded in a specific binary format.

All nodes starts with a formatted unsigned number. The lower two bits of that number represent the type of the node. The remainder of the bits represent the number of children that node currently has in its children node list, the child counter.

All nodes have a fingerprint field, containing the 20 byte SHA-1 hash of the chunk represented, and a size field, containing the size of the chunk represented. Some nodes have an offset field, containing the location where a node starts inside its parent. If that field is absent, the offset is 0.

Type 0 nodes are leaf nodes. They have no children or child hash. Type 0 nodes have an offset field if the child counter is 0.

Type 1 nodes contains all four fields: fingerprint, child hash, offset and size.

Type 2 nodes have no offset, but have fingerprint, child hash and size.

Nodes of type 3 are reserved for further extensions.

Type 1 and Type 2 nodes may have children in that tree. The children node list starts immediately after the child hash, and has a number of nodes equal to the child counter. Having a child counter of 0 on a node with a child hash different from the fingerprint means the sub-tree for that node was truncated.

The metadata is generated by an Optimized Hash Tree Builder class, that follows the algorithm described in the Section 3.1 to create the tree.

The algorithm reads the file to a buffer that will reach, at most, the maximum size for level 1. Every byte read is buffered in a byte array. When a level 1 node is created, the entire buffer is cleared. This optimization makes the algorithm faster, at the cost of some memory.

Even so, the metadata generation algorithm using this optimization never exceeded the 250-500MB memory range.

4.2 Services

This project defines a generic service that is used as a base for all the services, implemented in a ServiceThread class. This generic service contains two message queues, one for incoming messages and another for outgoing messages.

The service runs on four interconnected threads. The first thread runs the main service, the second thread receives incoming connections, the third thread turns incoming connections into messages and the fourth thread sends the outgoing messages. All messages are sent and received through TCP sockets. The service tries to group messages for the same address together, and sends as many messages as possible using the same TCP connection.

Messages received are processed by *handlers*, objects called by the ServiceThread. The various services' implementation vary only by their message handlers.

The ServiceThread system is based on the flow of messages. Each message received triggers a handler.

Each handler understands a set of messages. All messages exchanged by the system are called *service messages*. All service messages start with the same header information, and then append specific content based on the type of the service message.

The first byte indicates the message type. If a message has a reply, the reply type is usually 0x80+type of the message.

Next, the header contains an ID field that is used to identify the sender of the message, or an established session.

Finally, there is a reply port field, that tells which port the sender is listening to in order to receive a reply or a new request.

4.3 Lookup service

The lookup service is implemented as a handler that is capable of understanding ”lookup”, find and update” and ”insert” messages. ”Delete” messages were not implemented because they were not needed, because the maintenance service source cleanup was not implemented.

All the implemented messages have two variants, one for files and another for chunks. The chunk and file ”lookup” messages also have two variants, one that only allows for a single identifie to be looked up, and one that allows for a list of identifiers.

The lookup service contains two MySQL tables, one file lookup table and one chunk lookup table.

For implementation simplicity, both the tables were implemented with the same fields. Both contain an id field, a sources field and a parents field. The id field is a 20 byte binary, while both sources and parents are LongBlobs containing an array of DataSources (20 bytes id, formatted number offset). The table is indexed by the primary key, id.

The parents field, that stores the node’s parents, is not used on the file table.

The lookup service implements the chunk lookup algorithm as it was specified in Section 3.2.

4.4 Insertion service

The Insertion service contains a MySQL file insert table and 16 MySQL node insert tables.

4.4.1 File insert table

The file insert table contains the fileID as a 20 byte array, the UNIX timestamp of last access as a long, the number of levels of that file as a short and a small sub-table containing the number of fingerprints of that level present in the node insert table and the handprint end for that node.

This sub-table was stored as two LongBlobs, each containing a column of the sub-table. The number of entries in the sub-table is equal to the number of levels.

While the number of levels cannot be greater than 256, as explained in the Section 3.2, we still stored it a a shot due to Java not recognizing unsigned bytes.

The file insert table has the id as its key, and an index at the time, that orders the files in a least recently used first order. This index is fundamental to find the files that are going to be cleaned by the maintenance service in a timely manner.

4.4.2 Node insert table

The node insert table contained an chunk id field with 20 bytes, a source id field with 20 bytes, an offset field as a varbinary and a level number field as a byte. The source id field contains the file ID of where the chunk was found, and the offset field contains the offset where that chunk starts as a formatted unsigned number.

The table has the chunk ID as the key, and has one index with the file ID and one index to sort id by biggest first.

The node insert table was divided into 16 different tables because the indexes grew too large, and it became very slow to navigate such a large table.

It was necessary to find a way to divide the fingerprints into 16 different categories. The naive approach would be to divide the chunks equally, with each table keeping fingerprints with the same top half byte (ie: 0x00-0x0f = first partition, 0x10-0x1f = Second partition...). However, due to the cleanup system, the top partitions would be nearly impossible to clean up. Dividing by any other byte or even by the lower half of the first byte would solve that problem.

If we divide using the top half byte of the first byte, but spread the distribution unequally (ie: first partition is 0x00 to 0x03, yet last partition is 0xd8 to 0xff), the table that fills up faster will also be the one to be cleaned up more frequently.

This division also tells us where the most important and least important fingerprints are. If we have to deploy the 16 tables in different computers with different capabilities, we could use this division to choose where to store the tables, with the top tables being placed on the most reliable computers and the bottom tables being stored in less reliable computers.

The method that creates and distributes the hashes was implemented in a way that allowed for dynamic expansion of the number of tables.

As an alternative to erasing entries, the insertion service could instead decide to split the existing tables into smaller ones. The higher partitions should be split whenever needed, while lower partitions should be cleaned. This division system can only be used if the system still has space available.

4.4.3 Sessions

Because the ServiceThread treats every message individually, and the insertion algorithm requires various messages to be exchanged during its course, it would be impossible to implement the insertion

process as a single message handler. Because of that, the insertion service uses a concept of session, which represents the insertion algorithm for a single file insertion.

The Session class stores all the variables declared in the insertion algorithm in Section 3.3, as well as the source for the file being inserted and the client address. Sessions contain a state, that represents which is the step the algorithm is currently on. The state is used to filter which message types are expected, and how to deal with those messages.

When a user wishes to insert a file, he must first start a session. The insertion service generates an ID for that session and gives it to the client. Messages containing that ID in their ID field belong to that session, and are handled if the session allows it.

4.4.4 Source recovery service

Because there were no peers from which the insertion service could fetch the metadata, we created this dummy service that acts as a peer that provides metadata information.

This service acts as a cooperating peer, and does not refuse requests.

The service receives requests from the insertion service that contain a file ID and the fingerprints to check.

The source recovery service starts by fetching the metadata tree for the file ID the insertion service requested from the disk.

Then, for every node from the tree, the service compares it with every fingerprint requested. In order to simplify this interaction, both the fingerprints and the nodes of a level are ordered.

The service then sends the reply containing the nodes that matched the fingerprints request.

4.5 Maintenance service

Only one function of the maintenance service was implemented. That function was the node insertion table cleanup.

This function was not implemented as a standalone service, but instead was implemented in the insertion service code.

After each file insertion, we send a call to the simulated maintenance service. The cleanup code is run in the same thread that sent the simulated message that triggers a cleanup check.

The node cleanup algorithm is only triggered if a cleanup check decides the tables are getting too big in size.

In order to check for the need to cleanup, the maintenance service would need to count the rows of each node insertion table. Our database ”count” query was too slow. In order to speed up the process, we implemented another table, a statistics table, that kept an estimate count of the number of entries in each node insertion table.

We feel that this implementation stays true to the maintenance service designed. Nothing stated in the architecture forces the maintenance service to be a different process or thread. In fact, it would be much safer to run the maintenance service locally to the insertion service.

The cleanup algorithm was implemented exactly as specified in Section 3.4.

Chapter 5

Evaluation

With this evaluation, there are several questions we would like to answer. How many additional sources can we find for each of our given files? What is the expected time for inserting a file into the service? What is the overhead produced by the services? Can we, at least, provide as many sources as SET?

Terminology

Before we begin describing this evaluation, we must define some terms.

We define the popularity of a torrent as the total number of seeders, with a torrent with many seeders being more popular than one with few seeders.A torrent with a larger number of seeders would be a torrent that many people that finished the download decided it was worth to keep sharing, a good definition of popularity.

The number of Whole file sources is the number of distinct torrents from which a copy of a file can be downloaded. To simplify our evaluation, this number is used as the total number of copies of a file that exist in the network.

The number of chunk sources is the number of files from where one can obtain a given chunk. These sources may be from inside other, different files, or from inside the file itself (internal redundancy). A chunk is worth a percentage of the file it is in, which is given by dividing the size the chunk by the total size of the file. This is called the *chunk percentage*.

The percentage of chunk sources is given by this equation

$$p = \frac{\sum size_{C_i} * fileSource_{C_i}}{fileSize} * 100 \tag{5.1}$$

for every chunk C_i that that composes the file. The size of a chunk C_i is given by $size_{C_i}$, $fileSource_{C_i}$ is the total number of files where C_i was found.

The total availability of the file adds to the above equation $100 * wholeFileSources$, and represents the percentage of the file available in the network. For any file, the number of sources is at least 1, so the availability is always bigger than 100.

5.1 Experimental methodology

Used data set

In order to evaluate this system, we first downloaded the top 10 most popular torrents from the torrent indexing site ”piratebay.se” audio section, which included music in mp3 and m4a formats, and videos in the mpeg format. For each of those top 10 popular torrents, we searched for other torrents that shared the same artist and name, and downloaded a maximum of 10 from that list. Those other downloads, by comparison, usually had one tenth to one hundredth of the original torrent seeders, being much less popular.

We chose these datasets in order to fairly compare our results with SET, which used a similar set of data for their tests [42].

The total number of torrents downloaded and analyzed was 62, with a total of 1154 files, of which 956 were distinct, and 198 files had at least one copy. Among these 198 files, one file had 7 copies of itself, 32 files had 3 copies, 4 files had 2 copies and 90 files had only one other copy. This means that one of the files had 8 whole file sources, 32 files had 4 whole file sources, 4 files had 3 whole file sources and 90 files had 2 whole file sources.

Experimental procedure

We generate the metadata for each downloaded file using our multi-level metadata generation algorithm. We then proceeded to insert them into our system, using a test client that activated the insertion service. During the insertion process, we collected several measures, namely the time it takes to insert a file and the associated network overhead.

After all files were inserted into the system, we used a test client that measured the number of sources found per file. The results are presented in the above described percentage of sources found, and distinct sources found. We also present the time it took to find all sources, assuming a greedy client that wants to find all sources for all chunks.

Finally, we compare the results obtained with the number of sources found using SET’s algorithm.

Experiment setup

To simplify the testing process, all tests were performed on a single machine, with an Intel i7 3610 CPU at 2.3GHz. The computer ran on Windows 7 64-bit operating system, using Java Runtime Environment 7u16 with 2GB maximum heap size. The database used to store both Insertion and Lookup tables was MySQL server 5.6 community edition, held inside one Samsung 830 SSD, in order to minimize the impact of simple table retrievals on the testing. All the tests were run only once, due to time constraints.

This environment allows us to evaluate some specific metrics, such as the message overhead and time taken in a relatively optimistic scenario, where messages always arrive at their destination almost

instantly. This evaluation shows the performance of the system, ignoring the outside variables such as bandwidth latency. The reason this environment was chosen was due to limited time and hardware availability. In future work, it would be interesting to test this system working in a multiple computer environment, possibly already taking into account the network environment.

5.2 Insertion evaluation

The first step of insertion testing was generating the metadata. For each downloaded torrent, we created one set of metadata per file, with at most 9 levels, starting at 1024 bytes and ending at 64MB. The lower threshold (1KB-4KB) is due to being the lowest range that reaches our acceptable metadata size, as explained in the Section 3.2. The highest threshold (64MB+) was chosen because we believed it to be a realistic top level that BitTorrent users could choose for very large files.

The average size of the metadata was 2.89% of the total file size with an standard deviation of 1.14%, for any file larger than the minimum chunk size. No file over 2KB has more than 7% of its size in metadata. For smaller files, that ratio increases due to the auxiliary metadata (level hashes, level range definition). This confirms the mathematical analysis presented in Section 3.1.1. The time it takes to build the metadata file grows linearly with the size of the file analyzed, with an average of 0.43ms/KB and a standard deviation of little over 0.01, for any file over the size of 50KB.[1]

After the metadata was generated, we started inserting it into the system. We grouped up the files by the torrent they belonged to, and inserted one torrent at a time.

We measured the time and message overhead of the insertion process. Insertion time was measured for each torrent and for each file that composed it. Because our file selection contains files of vastly different sizes, insertion times and message overhead would also vary greatly with each files, making it difficult to examine the data. As such, we will usually present ratios of time and sizes, dividing those values by the number of nodes of a given file.

As inserting a torrent means to simply insert each of its files in order, the results were simply the sum of the results of the composing files data. We will, therefore, ignore the whole torrents insertion data, and focus on the files only.

We can divide the files inserted into two main categories: Duplicates and originals. Originals are files that either have no whole-file duplicate or that were the first of the duplicates to be inserted. Duplicates are all files where at least one copy was already present in the file lookup table.

5.2.1 Duplicate insertion

The insertion of duplicate files shows us something we already expected from the algorithm. Duplicate files are looked up and inserted very fast, and have almost the same message overhead across multiple

[1] We did not include files under 50 KB due to their ratio being vastly bigger due to rounding errors in the time taken.

files. This happens because each duplicate insertion only takes one lookup and exchanges only one node, the file node. This table presents the value obtained for the times as milliseconds and message overhead as bytes, not as ratios. The values are nearly constant for all duplicates inserted, a total of 198 duplicate files.

	Average	σ	Max	Min
Time (ms)	6.87	2.34	20	4
Message Overhead (Bytes)	206.19	1.93	209	199

Table 5.1: Statistics collected for Duplicate insertion

5.2.2 Original file insertion

The insertion of original files involves a different part of the algorithm. The number of nodes now influences heavily the time and message overhead for the file insertion. By ordering the files by the number of nodes, we can see the trend as the number of nodes grow for the algorithm to stabilize the ratios of message overhead and time.

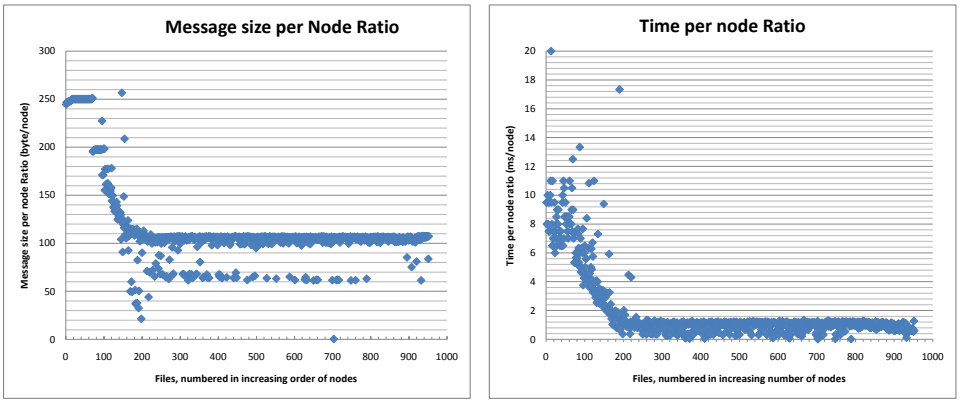

Figure 5.1: Time and Message size ratios for all original files

The above figure show a stabilization of the ratios as we reach files with over 50 nodes. With that in mind, we separate the files into two groups, with the first group (first 168 files) having less than 50 nodes, and with the remaining files (785 files) having at least 50 nodes. The results are as follows:

	Average	σ	Min	Max
Time (ms/node, $<$ 50 nodes)	6.33	3.08	1.69	23
Message Overhead (B/node, $<$ 50 nodes)	195.92	52.84	90.88	256.6
Time (ms/node, $>=$ 50 nodes)	0.99	0.68	0.01	1.31
Message Overhead (B/node, $>=$ 50 nodes)	101.09	13.45	0.30	115.2

Table 5.2: Statistics collected for unique file insertion. Divided between files with 50 nodes

If we first look at the time, we see that the ratio between nodes and time at the lower group varies greatly with a maximum registered of 23 ms per node being registered on a case with 2 nodes. As the

number of nodes increases, the time spent per node decreases to nearly 1ms per node.

This means that, for a very large file in the order of 1GB (around 870.000 nodes), the expected time for insertion is about 14 minutes if no duplicate node exists. For our largest file, a 2.5 GB file with 2 million nodes in its corresponding hash tree, the insertion took nearly 20 minutes. Even though we consider 20 minutes to insert a file is an acceptable time for an uploading peer, we designed the system to be quick and easy to use for the usual peer, not the uploader.

Looking back at the message overhead graphic, it is noticeable that after 50 nodes, some files have a much lower ratio than the average. The files that break the average are the ones that have found some duplicates in higher levels of the tree already present at the Chunk Lookup table.

When chunks are found in that manner, the algorithm does not need to request that entire sub-tree, and that is evident in the reduced message overhead, with the average being 72.51 bytes per node, a reduction in nearly 30 bytes per node.

This may appear small, but the difference between a 107 byte per node average (of our largest file) and 83.7 bytes (from our second largest file, with 2.3GB) means nearly 40MB of saved overhead. This proves that our level system is useful in reducing the network overhead from transferring metadata, as long as some higher-level nodes are detected.

5.2.3 Cleanup of the Node Insertion Service

The cleanup function of the Maintenance service only kicked in when the last torrent was inserted, due to the amount of data involved. We tested this last torrent separately with and without the cleanup code. Strangely enough, the time ratio was roughly the same, with approximately 107 message overhead ratio and 0.8 time ratio in both cases.

Because the data extracted was not significant enough to be presented, we will not discuss these results. On the other hand, the time to execute the cleanup was measured to be between approximately 44.7 minutes in the first run, and approximately 33.2 minutes on the second run. This also required further testing before a conclusion could be reached, but it clearly indicates the cleanup function is very expensive.

5.3 Lookup testing

After finishing the insertion, we used another test client to lookup every file, in order to find out how many sources were found.

The client would open the metadata file and, for each level, would request from the Lookup Service all chunk sources found. When the client finishes looking up the entire file, the client would then gather several statistics, such as the number of files sources found, the number of chunks found in common between files, the internal redundancy found in each file, the percentage of the file available in the network (by adding up all existing sources) and the time and message overhead for the system usage.

We will begin by checking the number of file sources found against the number of sources found using the SET method.

5.3.1 Number of file sources found

Being the closest related work available for testing, we implemented a simple version of SET for comparison with our work. We measured the number of sources found by both systems, as well as the number of fingerprints required to be stored to keep the system functioning. We used the second to last level of our metadata structure (the 4KB-16KB range level) as the metadata for SET. We used the first 30 ordered hashes from that level as the handprint. This causes SET to detect files up to 5% similar.

We implemented a client that first inserted the handprints of every file into a table. Then, for each file, the client looked up the handprint from that same table. We then compared the number of file sources found against the results obtained with our algorithm. Finally, we compared the space taken by useful data from SET memory versus our own from the Lookup Table.

We extracted the number of files that have at least one chunk in common from our source results. 196 files were not detected as having a single chunk in common by either SET or our system, and as such were not considered for this test. The remaining 760 files were analyzed, and the results are presented in the following figure:

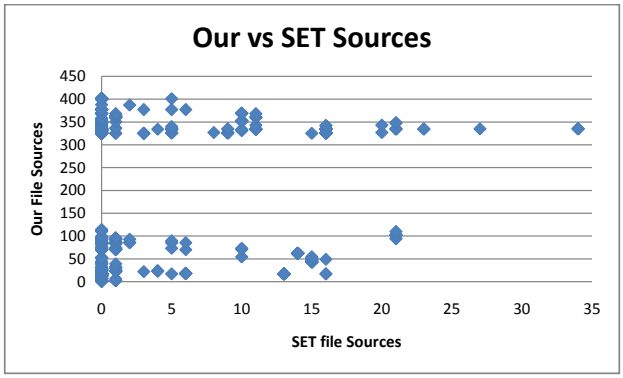

Figure 5.2: Files detected using SET vs. using our approach. The SET file sources scale had to be adjusted in order for SET results to show on the graphic

As we can see from the results, there are a large number of sources for files we have detected that SET has not.

SET only detected 294 similar files out of the 760 files with shared similarity. SET failed to detect 50 files within the range of similarity it was supposed to detect, mostly because the nodes in common were detected at a lower range (1KB-4KB range). There was no case SET detected we were unable to detect, which was expected as determined before in the Section 3.3. There was no case where SET detected the same amount of sources our approach did.

From the graph, we can clearly distinguish between two groups of results. One set of files shared between 400 to 300 file with some similarity. The second group shares between 120 files to one file with

some similarity. The reason for this file division is due to the files in the 400-300 range sharing a group of very common chunks with each other (padding chunks).

5.3.2 Found sources

We began by calculating the maximum and minimum similarity between files. To do this, we calculated the percentage of sources found for each chunk when considering only one other file as the extra source. We ignored whole-file copies for this evaluation. The results are presented in the figure below:

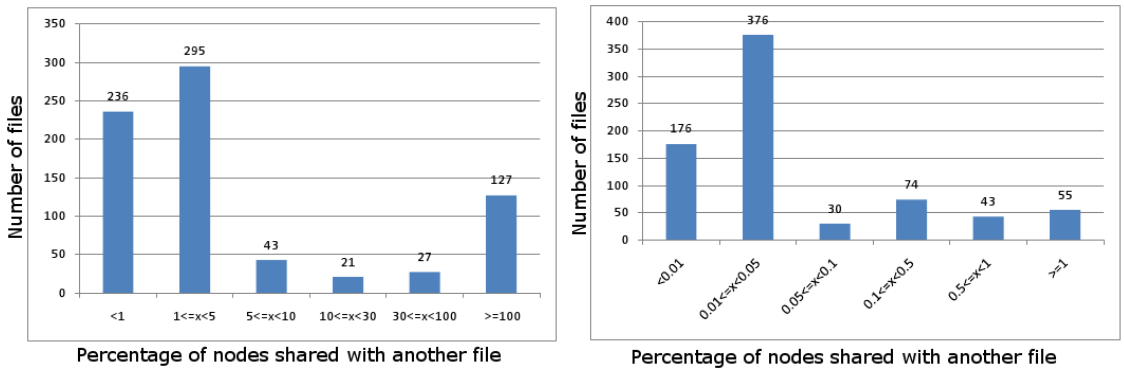

Figure 5.3: Maximum and minimum similarity between any pair of files. The x axis represents a percentage interval, while the y axis represents the total number of files

As we can see in the left-hand graphic, most files that had duplicate chunks never shared more than 5% of their contents with another file, totaling 531 of the 753 files. Out of the remaining files, most files (127) have more than 100% of themselves in common with another file[2]. What this means is that there is a file out there that contains this file, with some internally redundant chunks also shared by these files.

On the right-hand graphic, we see that most files shared less than 0.05% of their content with another file. While this is very little, it still represents an improvement over no extra sources, and may allow the user to download 500KB data on a 1GB file form another source. We can also see that only 55 files had at least 1% or more share content between them. This chart clearly shows there is much similarity to be exploited on the very small similarity range. These small pieces add up, and the result is an increased availability as presented in the next graphic.

This above graphic tells us what is the total availability of a file in our system. This availability is the percentage of the file that can be downloaded from any source that our system has detected. The majority of files (491 in total) fall in the range between 100 and 150% availability, with 80 files fitting in the 130 to 150% range. There are 269 files with over 150% availability, with 218 files having more than double the availability from different files using our system.

All the above results support the claims that there is much redundancy between files in P2P networks, and more specifically, our belief that much of that redundancy happens below similarity levels considered

[2]No files were encountered with exactly 100% in common

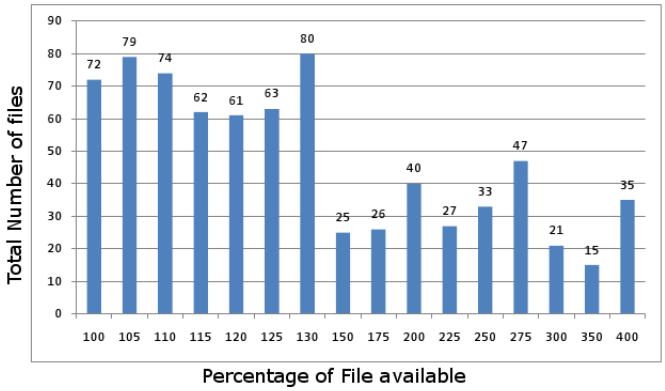

Figure 5.4: Total Availability of a file, not counting extra whole file sources

relevant by most other work done in the field. The result also show that when adding all those small, irrelevant, chunks together, it is possible to increase the total availability of a file by a significant amounts.

5.3.3 Time and Message overhead

Another important aspect of our system is the time it takes to lookup a chunk, and the overhead consumed by that action. Our lookup test is an extreme case, where the user looks up the entire tree. This is not the typical, expected case, but is the worst case scenario. This makes this case relevant.

Once again both functions show a stabilization of the time and message overhead ratios as we reach files with over 50 nodes. As such, we will examine the less than 50 nodes case separately from the greater than 50 nodes case.

	Average	σ	Min	Max
Time (ms/node, $<$ 50 nodes)	3.71	2.33	0.40	9
Message Overhead (B/node, $<$ 50 nodes)	144.48	268.44	49.73	1935.36
Time (ms/node, $>=$ 50 nodes)	0.17	0.14	0.004	1.34
Message Overhead(B/node, $>=$ 50 nodes)	65.64	52.53	4.49	934.60

Table 5.3: Statistics collected for whole file lookup. Divided between files with 50 nodes

The full tree lookup takes, on the average case for larger trees, 17% the time of the insertion, with a maximum registered of 1.34 on the top of the tree. The message overhead, however, is much more spread out, although also smaller. The reason for these wildly varying values is due to the number of sources found, which can be much greater than the number of nodes in a good run.

The following are the graphics that show the system performance in the worst case.

Another important note is that the time it takes to lookup a file is always increased if more sources are found. It took little less than 8 minutes (455908 ms) to lookup the entire tree for the largest file, that has a 1.04% extra sources. However, it took almost double that (under 12 minutes, 701847ms) to lookup the second largest file, that had 28.43% extra sources.

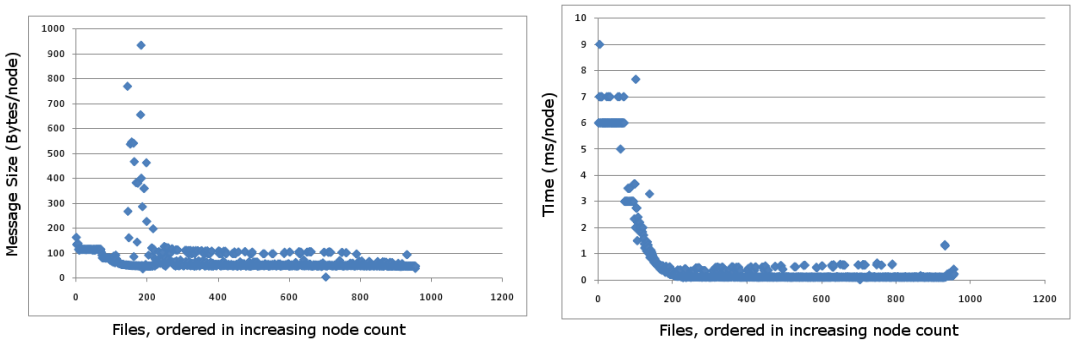

Figure 5.5: Time and Message size ratios for all original files

Overall both this and the Insertion Service tests show that our system is more adequate for files with more than 50 nodes, that is, files that are over 60KB in size. These tests also show that the system is capable of accurately identifying a large quantity of sources for many common chunks, even across many files that have little in common.

5.3.4 Space occupied

The total size of the node lookup table is 122MB in disk, with only 28MB actually containing table information. Considering the working set has 39 GB, this means the ratio between data processed and space occupied by the nodes was 0.07% at least, and 0.3% at most.

The SET table was setup in memory, and took about 5MB in Java's virtual memory, with an estimate 1MB of actual data content.

The massive difference in space occupied comes from some important differences. First, SET only stores a very small amount of the total number fingerprints, requiring the user to download the metadata for the files that share one fingerprint in common. Our table stores all found duplicate chunks, providing the sources immediately to the client, with no more overhead required. Second, SET only stored handprints for a single level. Our system stored found chunks for all levels.

Every node stored in our table was useful, containing unique sources. SET table had 20070 empty nodes out of 23765 total. These empty nodes are only potentially useful.

When balancing both solutions, ours offers greater accuracy and ease of use in exchange for disk space. SET is fast and small, but requires the user to download large metadata files and find common chunks on his own, and is not capable of identifying as many sources as we do.

Chapter 6

Conclusions

When we began this project, we started with a simple question: ”Would it be possible to detect similar data across BitTorrent and exploit that to find additional sources?” We knew that whichever solution we brought up had to be capable of dealing with the massive amount of information in the BitTorrent network, as well as be simple to use and accurate.

As we searched for the response to that question, we came across SET, which provided a possible answer. However, SET came with a number of shortcomings, such as imposing a heavy overhead onto the client, requiring the client to download the metadata for all similar files found, which broke our expectations in terms of accuracy and the amount of work imposed on the client.

We wished to find a solution that allowed the user to find as many sources as there are for any piece of data, while minimizing the overhead for the peer that looked for that information. That system had to be implementable on the scale of BitTorrent.

In the end, we proposed a novel approach composed of a group of services: the insertion service, that found and gathered the redundant data; the lookup service, that provided the users the information about the sources and was responsible for keeping it stored; the maintenance service, responsible for keeping the other two services in working order.

We also created our own type of metadata, a multi-level fingerprint tree that brings together the benefits of small chunk redundancy detection, yet can be distributed in a much smaller size to the users.

We implemented and evaluated a version of our three services, as well as the metadata structure.

We have shown that it was capable of detecting similarity between files even when the similarity was as low as 0.01%. Our system detected at least as many file sources as SET, finding between 10 to 100 times more file sources than SET. We also shown that many files share very little in common, but adding the common chunks together created a much larger availability for the file. In over 1/3 of the tested files, that availability increased by 50%.

Our lookup service spent 28MB in useful data reporting on 39GB of unique files, reporting nearly 498000 different nodes. It could reply to the client in an average time of 0.1ms per node, spending an

average of 145 bytes of message overhead per node accessed.

The insertion service can work in any extra space given, working better when more space is available. The time taken to insert per node is close to 1ms, and spends an average of 101 bytes overhead per node.

All the above results obtained meet our requirements of a low overhead on the user, and implementing the system in a way that could scale to the entirety of BitTorrent.

6.1 Future Work

There is still much work to be done. There are still many parts of the system of which the behavior must be further studied. There is also some work left in order to deploy the system in a real-world environment.

The first and most important study to be made would be to implement the remaining of the project and test its impact on the actual BitTorrent network. It would be needed to evaluate not only the impact of the system in the peer, but in the torrent health as well.

In order to deploy the system in the real world, there are several areas that must be looked upon.

Improvements to the Insertion and Cleanup times

It would be important to study new ways to reduce the insertion and cleanup times, that may reach times in the order of minutes in our tests.

In the insertion case, most of the time was spent in accessing the MySQL table and ordering the nodes. Finding more efficient storage methods may reduce the total insertion time further to much smaller values.

On the maintenance service node cleanup case, the reason the implementation took so long was because the first time it was triggered it was due to the insertion of a file that had much more nodes than any previously entered file. In order to make room for the new file, the cleanup algorithm had to cycle through the entire table several times. Finding new heuristics for choosing the nodes to delete could speed up the process significantly.

Trust

One big set of problems come from the lack of trust in the peers using the system. This includes users of both services, lookup and insert alike.

The first problem comes from the authenticity of inserted data. We can only verify the metadata integrity A peer may submit a piece of metadata created from a random array with the size of the file at

hand, and submit the tree for that random array with the file ID on top, and we would not know the better. This problem affects the whole project, which can only be deployed as-is if we trust the inserting peer.

As it was out of the scope of this work, we did not develop any way for verifying the authenticity of the data, either before, during or after their insertion.

In order to solve the trust issue at time of insertion, one possible solution would be to trust only the system.

Either the insertion service or the maintenance service could be responsible for searching the web for new metadata to be inserted. The service would verify the metadata authenticity by downloading some random chunks and verifying them against the metadata. The metadata would then only be inserted if they are satisfied that data is true.

Another possible solution would be to introduce trust-based mechanisms, and allow for the data stored in the Lookup service to have associated degrees of trust in each source.

Both these solutions would allow to insert files even if the person who produced the file was not the one to create the metadata.

In any case, it would be good to develop a way to verify the data after insertion. Even simple hardware faults can corrupt the data, and there is no way to correct it. We suggest using the Maintenace service to sometimes try and download a chunk, and see if it is true, and if not question the source it came from for its validity.

Security

Related to the trust problem, there are many parts of this system that are not secure.

We mentioned some security measures, such as authentication for the insert, update and delete messages for all the services, but never specified how it was done. Implementing the authentication process would be required to deploy this solution.

Besides the authentication, it would also be interesting to study the possibility of securing the messages through encryption and adding anonymity to the system.

Fault Tolerance

The data in the Lookup service is considered permanent and true. This makes it important to secure the data inside the lookup tables.

While we touch the subject, there is much work to be done in this field as well. The size of the data inside the lookup tables, as well as the rate that data should enter and leave the tables, makes it hard to make the system fault-secure.

The information in the insertion table can be safely lost without crippling the system. However, that does not mean the information can be stored in a faulty environment. If an entry is wrongfully altered, it

will provide wrong information that will be taken as true. Adding some simple checksum to the entry and discarding it if corrupted should be enough.

Storage

This system was created with the storage problem in mind, but its implementation and testing were done in a single-server database, which is an expensive solution in this case. From the beginning of conception we talked about how this system would benefit from being placed in a DHT, but that implementation is not trivial. All of the above mentioned problems would be amplified if a DHT was used for storage. The move would be most beneficial, but needs to be studied carefully.

On the lookup case, it is important to study the impact the parent chain method of storage causes on table lookups. It should be measured if the space it saves is significant considering the increase in lookup time.

Metadata generation

The metadata generation needs further study, for many reasons.

First, the parameters used for the metadata generation (size of smallest chunk, number of levels and range increase) should be evaluated for its performance impact. The analysis we provide only show the impact of those parameters in the resulting metadata file size.

One example of a parameter study would be to see their relation to the number of duplicates found. Another example study would look at the parameter influence in the system time performance.

These studies should lead to a standard level range chain, that would be well suited to all files. This level range chain should maximize the rate of deduplication achieved while minimizing the space occupied and time to lookup.

Another problem with choosing the level ranges is that identical files with completely different level ranges may result into entirely different common chunks, yet only one of those files would be inserted. The benefits of inserting both files, and allowing the user to access both, needs to be studied as well.

The way chunks are grouped together to form their parents is another section of the metadata generation that needs to be explored.

The approach we used, blindly joining the chunks of smaller size, might not be the best in order to maximize redundancy detection.

The way chunks are grouped may also affect the way levels are formed. This has an impact on the fingerprints generated in those levels, and that affects the level hash and the children hash of every node in the level above.

If we create two trees for the same file, with slightly different level ranges, the resulting trees would be almost entirely different. Studying alternative ways of joining chunks together to form parents may allow for a greater duplicate detection on the middle and higher levels.

Finally, allowing for the metadata file to create simple replacements for parts of the file could be beneficial. If the generation algorithm detected a very large pattern of repeated data, such as a section of padding, the algorithm could mark it as a special type of chunk, that would not be transferred.

This detection would remove the user's need for transferring these useless portions of a file. It would also reduce the reporting of identical sources across files that are largely different.

This form of padding detection could lead to other mechanisms that detect common patterns, or even file-type specific information.

Bibliography

[1] http://www.isoHunt.com/.

[2] http://www.bittorrent.com/help/manual/appendixa0403#0ther.Preserve_file_order.

[3] The gnutella protocol specification v0. 4.

[4] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys (CSUR)*, 36(4):335–371, 2004.

[5] K. Bauer, D. McCoy, D. Grunwald, and D. Sicker. Bitblender: Light-weight anonymity for bittorrent. In *Proceedings of the workshop on Applications of private and anonymous communications*, page 1. ACM, 2008.

[6] Brian Berliner et al. Cvs ii: Parallelizing software development.

[7] J. Black. Compare-by-hash: A reasoned analysis. In *Proc. 2006 USENIX Annual Technical Conference (Boston, MA*, pages 85–90, 2006.

[8] D.R. Bobbarjung, S. Jagannathan, and C. Dubnicki. Improving duplicate elimination in storage systems. *ACM Transactions on Storage (TOS)*, 2(4):424–448, 2006.

[9] Scott Chacon. *Pro Git*. Apress, Berkely, CA, USA, 1st edition, 2009.

[10] X. Chen and XW Chu. Understanding private trackers in bittorrent systems. Technical report, Technical Report, http://www. comp. hkbu. edu. hk/˜ xwchen/private _tracker/pt_tech_2010. pdf, 2010.

[11] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

[12] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[13] B. Cohen. The bittorrent protocol specification, 2008.

[14] S.A. Crosby and D.S. Wallach. An analysis of bittorrent's two kademlia-based dhts. Technical report, Technical Report TR07-04, Rice University, 2007.

[15] G. Dán and N. Carlsson. Dynamic swarm management for improved bittorrent performance. *IPTPS'09*, 2009.

[16] I. Drago, M. Mellia, M.M. Munafò, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: Understanding personal cloud storage services. 2012.

[17] K. Eshghi and H.K. Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30, 2005.

[18] Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes. Jumbo store: providing efficient incremental upload and versioning for a utility rendering service. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, FAST '07, pages 22–22, Berkeley, CA, USA, 2007. USENIX Association.

[19] António Ferreira. Monitoring bittorrent swarms. Master's thesis, Instituto Superior Técnico, 2011.

[20] Antonio Homem Ferreira, Ricardo Lopes Pereira, and Fernando M Silva. Content redundancy in bittorrent. In *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, pages 1–7. IEEE, 2012.

[21] D. Harrison. Bep 0004 assigned numbers, 2008.

[22] D. Harrison. Bep 0023 tracker returns compact peer lists, 2008.

[23] G. Hazel and A. Norberg. Bep 0009 extension for peers to send metadata files, 2008.

[24] V. Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 13–18, 2003.

[25] N. Jain, M. Dahlin, and R. Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4, pages 21–21, 2005.

[26] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 640–651, New York, NY, USA, 2003. ACM.

[27] T Klingberg and R Manfredi. Gnutella 0.6 rfc draft.

[28] A. Legout, G. Urvoy-Keller, and P. Michiardi. Rarest first and choke algorithms are enough. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 203–216. ACM, 2006.

[29] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys (CSUR)*, 22(4):321–374, 1990.

[30] Jian Liang, Rakesh Kumar, and Keith W Ross. The kazaa overlay: A measurement study. In *Proceedings of the 19th ieee annual computer communications workshop*, pages 2–9. IEEE, 2004.

[31] A. Loewenstern. Bep 0005 dht protocol, 2008.

[32] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM, 2002.

[33] S. Manuel. Classification and generation of disturbance vectors for collision attacks against sha-1. *Designs, Codes and Cryptography*, 59(1):247–263, 2011.

[34] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Peer-to-Peer Systems*, pages 53–65, 2002.

[35] D.S. Menasche, A.A.A. Rocha, B. Li, D. Towsley, and A. Venkataramani. Content availability and bundling in swarming systems. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 121–132. ACM, 2009.

[36] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference*

on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology, CRYPTO '87, pages 369–378, London, UK, UK, 1988. Springer-Verlag.

[37] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 174–187, New York, NY, USA, 2001. ACM.

[38] A. Norberg. Bep 0029 utorrent transport protocol, 2009.

[39] A. Norberg, L. Strigeus, and G. Hazel. Bep 0010 extension protocol, 2008.

[40] Spek. O. Bep 0015 udp tracker protocol for bittorrent, 2008.

[41] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in bittorrent. In *Proc. of NSDI*, volume 7, 2007.

[42] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. 4th USENIX NSDI*, Cambridge, MA, April 2007.

[43] M.O. Rabin. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[44] Stefan Saroiu, Krishna P Gummadi, and Steven D Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia systems*, 9(2):170–184, 2003.

[45] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[46] Tran Doan Thanh, Subaji Mohan, Eunmi Choi, SangBum Kim, and Pilsung Kim. A taxonomy and survey on distributed file systems. In *Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management - Volume 01*, NCM '08, pages 144–149, Washington, DC, USA, 2008. IEEE Computer Society.

[47] A. Tridgell and P. Mackerras. The rsync algorithm, 1996.

[48] Sandvine Incorporated ULC. Global internet phenomena report: 2h 2012, November 2012.

[49] Li Xiong and Ling Liu. Building trust in decentralized peer-to-peer electronic communities. In *Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.

[50] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Fast*, volume 8, pages 269–282, 2008.

[51] T. Zink and M. Waldvogel. Bittorrent traffic obfuscation: A chase towards semantic traffic identification. 2012.