UNIVERSITÀ DEGLI STUDI DI BERGAMO

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Classe n. 35/S – Sistemi Informatici

Digital Fountain Erasure Recovery in BitTorrent: integration and security issues

Relatore:
Chiar.mo Prof. Stefano Paraboschi

Correlatore:
Chiar.mo Prof. Andrea Lorenzo Vitali

Tesi di Laurea Specialistica
Michele BOLOGNA
Matricola n. 56108

ANNO ACCADEMICO 2007 / 2008

This thesis has been written, typeset and prepared using $\LaTeX\,2_\varepsilon$.
Printed on December 5, 2008.

Alla mia famiglia

“Would you tell me, please, which way I ought to go from here?”

“That depends a good deal on where you want to get to,” said the Cat.

“I don’t much care where —” said Alice.

“Then it doesn’t matter which way you go,” said the Cat.

“— so long as I get somewhere,” Alice added as an explanation.

“Oh, you’re sure to do that,” said the Cat, “if you only walk enough.”

Lewis Carroll
Alice in Wonderland

Acknowledgments (in Italian)

Ci sono molte persone che mi hanno aiutato durante lo svolgimento di questo lavoro.

Il primo ringraziamento va ai proff. **Stefano Paraboschi** e **Andrea Vitali** per la disponibilità, la competenza, i consigli, la pazienza e l'aiuto tecnico che mi hanno saputo dare. Grazie di avermi dato la maggior parte delle idee che sono poi confluite nella mia tesi.

Un sentito ringraziamento anche a **Andrea Rota** e **Ruben Villa** per l'aiuto e i chiarimenti che mi hanno gentilmente fornito.

Vorrei ringraziare **STMicroelectronics**, ed in particolare il gruppo *Advanced System Technology*, per avermi offerto le infrastrutture, gli spazi e tutto il necessario per svolgere al meglio il mio periodo di tirocinio. Un ringraziamento particolare a **Ruggero Susella** per i preziosi consigli che mi ha fornito.

Grazie anche a tutti i miei **amici** e **amiche**. Sarebbe davvero difficile elencarvi tutti (*you know who you are*), ma posso assicurarvi che ho più di un ricordo per ognuno di voi.

Grazie alla mia fidanzata **Francesca** che mi ha incoraggiato nei momenti più difficili. Grazie per avermi ascoltato e soprattutto sopportato tutte le volte che ero giù di morale.

Ringrazio anche i miei nonni, **Ernesto** e **Clotilde**, per avermi insegnato sin da piccolo quanto fosse importante lo studio; grazie per avermi risollevato il morale tutte le volte che vi siete accorti che ero preoccupato. Il mio pensiero va anche ai miei nonni **Antonio** e **Vincenza**: sarebbero stati sicuramente felici di vedermi laureato.

L'ultimo ringraziamento va senza ombra di dubbio a *coloro che hanno reso tutto questo possibile*: **papà Gianfranco** e **mamma Mariangela**, e mio fratello **Massimo**. Grazie di cuore per aver sempre creduto in me e per avermi permesso di studiare e di laurearmi. Grazie per tutti i sacrifici che avete fatto per me. Grazie per la vostra infinita pazienza (che io cerco di mettere a dura prova ogni giorno), per tutti gli incoraggiamenti e per tutti i consigli che mi date. Sono fiero di voi, e, anche se non ve lo dico spesso, vi voglio bene.

Michele

Contents

List of Figures

List of Tables

Preface

"Do you *really* know what's traveling across the Internet right now?"

Honestly, I knew that most of the Internet traffic was P2P. When I decided to study P2P networks and the BitTorrent protocol, I thought that my study will be useful to the most part of the Internet users. I did not know, however, that P2P networks are so challenging: there are matters of distributed computing, integrity, security, probability and so on.

So I started study the BitTorrent protocol: there are some specifications available, but they are very poor and not so detailed. As a consequence, I had to open up two (or more) BitTorrent clients, install a tracker on my system and watch a small P2P network at my commands. I also knew BitTorrent is the most used P2P program: you can download photos, documents, messages, music, videos and software from all over the world by the time of a click. With BitTorrent, anyone could be the multimedia entertainer: there is no need of typing the URL of your favorite entertainer anymore. You can download everything, legally or not; BitTorrent clients allow you to transfer a large amount of data using all of your bandwidth. However, even with a fast connection, the download process is always a start-and-wait task, especially for big files: is there room for improving?

P2P works on the Internet, which is a reliable channel, but it may experience some losses during a transmission. In fact, when queues and buffers on remote nodes become almost full, the packets traveling on the network are quickly discarded (*congestion* problem). This raises a problem of completeness: a receiver can not complete a download due to the lack of certain pieces of information. We are going to prevent this problem with the addition of redundancy, using erasure recovery codes. In addition, there is still the possibility that the information may be corrupted (intentionally or not). We have a problem of integrity: a receiver must check for integrity and correctness of every received piece of information. To detect integrity we could use hash functions; to correct a received corrupted block, we will use error correcting codes, that are a sort of code that can detect and correct (within certain limitations) any errors which are introduced. Notice that if the receiver can not correct a corrupted block may assume it as a lost block (thus it can use erasure recovery).

We are going to investigate the application of a new type of erasure recovery codes; they are classified as LDPC and are called Digital Fountain codes. This kind of codes guarantee that a receiver can reassemble the information by accumulating enough encoded packets – encoded packets have all the same criticality, thus there are not duplicate packets or most critical packets. Such codes can be used for erasure recovery or to correct errors. We are going to use them for erasure recovery, whereas in the future of this project we will develop correct errors.

First of all, we need to know BitTorrent: the most used P2P application of our days; we are going to investigate its terminology, its way of download and upload files and its weaknesses from a security point of view. Because of the lack of detailed documentation, we have to study the protocol in details and make some experiments. Then, we analyze Digital Fountain codes: the encoding and decoding process, complexity and, mostly, encoding and decoding times, and finally, overhead (given the information total size, the overhead is the additional information that a receiver must collect in order to successfully reassemble the information). Subsequently, we design a new BitTorrent client that uses Digital Fountain codes: BitFountain. We are going to describe its functionality, its way of encode, transmit, receive, decode and validate information sent over the Internet. Integrating Digital Fountain codes won't be so easy: there are some security issues to take in account. We are going to analyze them and propose some solutions to solve those problems. At the end, a survey about future developments will depict the future of this project. This thesis is the most comprehensive and detailed project about BitTorrent and Digital Fountain available today. In addition, we describe how we integrate them and how we solve those security problems we mentioned before.

Chapter 1

P2P and BitTorrent

1.1 Peer to Peer (P2P) networks

The term Peer to Peer (P2P) can be defined as a network in which a significant proportion of the network's functionality is implemented by peers in a decentralized way, rather than being implemented by centralized servers[1]. A peer is a single program that is run on a number of hosts which interconnect to form a P2P network.

P2P networks uses varied connectivity between participants and the cumulative bandwidth of network participants; such networks are typically used for connecting nodes via largely ad hoc connections. A *pure* P2P network does not have the notion of clients or servers but only *equal* peer nodes that *simultaneously* function as both clients and servers to the other nodes on the network. This model of network arrangement differs from the client-server model where communication is usually to and from a central server. See figure 1.1 on the following page for comparison.

An important goal in P2P networks is that all clients provide resources, including bandwidth, storage space, and computing power. Thus, as nodes arrive and load of the system increases, the total capacity of the system also increases. This is not true of a client-server architecture with a fixed set of servers, in which adding more clients could mean slower data transfer for all users[2]. The distributed nature of P2P networks also increases robustness in case of failures by replicating data over multiple peers, and – in pure P2P systems – by enabling peers to find the data without relying on a centralized index server. In the latter case, there is no single point of failure in the system.

[1]This means that the implementation of functionalities is spread across all or most of the peers in the network.

[2]The underneath network is the same in P2P or server-client case. For many reason that we are going to illustrate, P2P exploits and takes bandwidth utilization to the limit. As a consequence, most ISPs throttle the traffic generated by P2P programs [46].

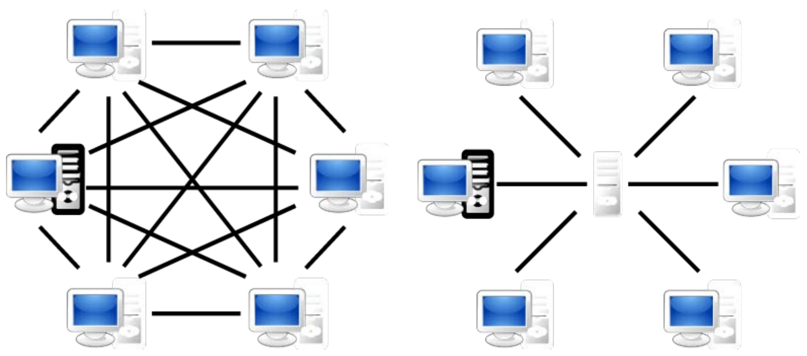

(a) A peer-to-peer based network. (b) A server-client network.

Figure 1.1: A comparison between P2P networks and server-client networks.

1.1.1 Classification

P2P networks can be classified by what they can be used for:

- content delivery.
- file sharing (files containing audio, video, data or anything in digital format).
- transferring real-time data (telephony).
- media streaming (audio and video).
- discussion forums.

Other classification of P2P networks is according to their degree of centralization. In pure P2P networks:

- peers act as equals, merging the roles of clients and server.
- there is no central server managing the network.
- there is no central router[3].

Some examples of pure P2P application layer networks designed for file sharing are Gnutella and Freenet. There also exist countless hybrid P2P systems, characterized by:

- a central server that keeps information on peers and responds to requests for that information.

[3]A decentralized routing algorithm is feasible only under definite hypothesis [9].

- peers are responsible for hosting available resources (as the central server does not have them), for letting the central server know what resources they want to share, and for making its shareable resources available to peers that request it.

- route terminals are used as addresses, which are referenced by a set of indices to obtain an absolute address.

1.1.2 P2P for file sharing

As we just said, P2P can be used for several purposes: file sharing is the most widely used P2P application. P2P file sharing systems consist of programs that are used to create and maintain P2P networks to facilitate the transmission of files between users. They allow users to download files from other users of the P2P network, and often also allow users to designate a set of files from their PC's file system to be shared. Sharing a file makes the file available to other users of the P2P network. There are two key parts of a P2P file sharing system:

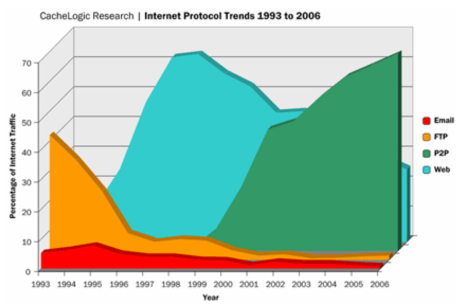
- file distribution system. The file distribution system provides the means to transmit files between peers. It is the protocol used to dictate how peers in the system should behave in order to download and upload files.

- file finding system. The file finding system is the means for users to find the files that are available on the P2P network. P2P file sharing systems typically provide the file finding system by maintaining some form of index of the files.

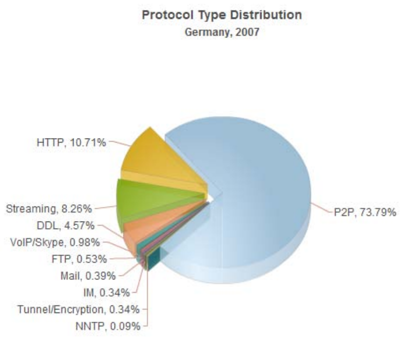
P2P file sharing systems differ in how and where they implement these two parts. Some maintain the file index in a centralized way, and others in a decentralized way. P2P file sharing systems implement the file distribution system in a *decentralized* way. This definition of P2P file sharing systems satisfies the definition of P2P given earlier – the most significant part of network's functionality, the transfer of files, is done directly between the peers in the network, without the use of centralized servers.

File sharing and the law

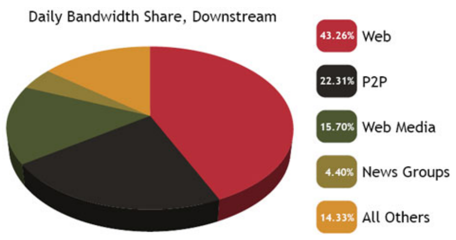
The debate on peer to peer and file sharing is a virtually global phenomenon. Peer to peer technology allows people worldwide to share files and data; however a significant proportion of the data shared is material passed freely between users that is (or should legally be) subject to copyright or other restrictions. Different legal systems, and different technologies, handle this differently. Some of the key background and distinctions are as follows:



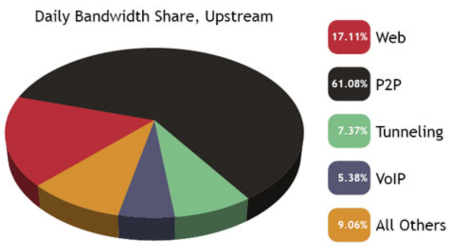
(a) Internet traffic trends from CacheLogic research (2006). We can see the increase of P2P traffic in the recent years.



(b) Ipoque, a German ISP, released a report [43] on P2P traffic usage in 2007. According to this report, P2P is the most used application in the Internet.



(c) Sandvine, a Canadian ISP, released a report [44] on P2P traffic usage in 2008. Regarding downstream bandwidth, P2P is the second most used application.



(d) Sandvine, a Canadian ISP, released a report [44] on P2P traffic usage in 2008. Regarding upstream bandwidth, P2P is the most used application.

Figure 1.2: A comparison of reports on Internet traffic: P2P takes up most of the Internet bandwidth. According to Sandvine's study, P2P technology is going to see explosive growth in the magnitude of 400% in the next five years, equating to 8 petabytes of traffic per month.

- P2P file sharing is used both legitimately (to distribute with permission or non-copyright materials), and illegitimately (in breach of copyright). It is highly popular and effective, with some estimates being that 18–35% of all internet traffic is P2P usage in some form or other.

- P2P systems vary - some rely upon a centralized server, others are decentralized without the need of a site operating the system. Recent systems often have anonymity or obfuscation built in, making it harder to identify senders, recipients and material, and providing a degree of plausible deniability.

- in some file sharing systems, the owner of a sharing system directly distributes files themselves (i.e. Rapidshare[4]). In others, notably BitTorrent, the organizer is not in fact distributing any copyright material. Rather, they act like a cataloger or co-ordinator, indexing files rather than themselves offering any such material. A typical such file might provide a filename, a location it can be downloaded from, and various checksums which can be used to verify the file's integrity when downloaded. It does not, itself, contain any media material, whether legal or otherwise.

1.1.3 Application of P2P network

- Bioinformatics: P2P networks have begun to attract attention from scientists in other disciplines, especially those that deal with large datasets such as bioinformatics. P2P networks can be used to run large programs designed to carry out tests to identify drug candidates. The first such program was begun in 2001 the Centre for Computational Drug Discovery at Oxford University in cooperation with the National Foundation for Cancer Research. There are now several similar programs running under the auspices of the United Devices Cancer Research Project.

- Search engines:

  - The sciencenet karl [40] provides a free and open search engine for scientific knowledge. Universities and research institutes can download the free Java software and contribute with their own peers to the global network.
  - YaCy is a peer-to-peer search engine and web crawler. Users install the software and become a YaCy-peer, volunteering their computer to independently crawl through the web, analyzing and indexing websites into a database shared by all Yacy peers. More than 400 million websites have been indexed by YaCy. There is no central

[4]`http://www.rapidshare.com`.

server; the database is shared and upheld by the YaCy peers. There are a few distinct advantages of a decentralized peer-to-peer search engine: since there is no central server or company who owns the service, the search results cannot be censored, and no incentives to prioritize results based on prospective contracts or advertising dollars.

- Education and Academia: due to the fast distribution and large storage space features, many organizations are trying to apply P2P networks for educational and academic purposes. Stanford University use BitTorrent to give away some of their engineering courses; the university not only gives away videos of lectures, but also syllabi, handouts, homework and exams. In addition to offering torrents, the courses are also available on YouTube, via iTunes and Vyew. With the project Stanford aims to spread knowledge on technology worldwide. Thus far, the online courses have been a great success. Over 200,000 people from all over the world have visited the site already.
- Privacy: Tor is an application that shields its user's identities by sending their traffic through a network of relays set up by volunteers around the world. In other words it prevents somebody watching your Internet connection from learning what sites you visit, and it prevents the sites you visit from learning your physical location. Tor is used by everyday ordinary Internet users who wish to avoid advertiser's behavioral targeting, citizen journalists in countries without safe access to media, law enforcement setting up anonymous tip lines, activists, and whistleblowers. To accomplish this, the Tor network relies on people to volunteer their Internet connection as a relay. These relays send user's content privately to other volunteer relays with the aim of obfuscating the user's location or identity. Those who volunteer their Internet connection as a relay are committing to allowing a certain level of bandwidth usage flow through their pipe (20KB/sec minimum).
- Business:
  - P2P networks have already been used in business areas, but it is still in the beginning stages. Currently, over 200 companies with approximately $400 million USD are investing in P2P network. Besides file sharing, companies are also interested in:
    - distributed computing.
    - content distribution.
    - e-marketplace.
    - distributed search engines.
    - groupware and office automation via P2P networks.

  At the same time, P2P is not fully used as it still faces a lot of security issues.

  - Colonos Workplace is a cross-platform P2P system for collaborative teams to work on projects and transmit their work amongst their work group. This P2P based product fills the gap for small and medium size businesses who don't have the IT budget for the elaborate collaborative software found in larger businesses. Currently out in Beta phase, is its open source integrated VOIP technology, so team members can talk in real time, conduct conference calls and to communicate through its integrated multiplatform, multiprotocol instant messaging features while collaborating on work projects.

- TV: delivery TV content over a P2P network (P2PTV). The European Union has committed €14 million ($22 million) for a four-year project to create an open source, peer-to-peer BitTorrent client called P2P-Next. This client will hopefully become a new standard way for broadcasters to use the Internet as a low-cost distribution platform. Users will have the option of either downloading material or viewing live video streams. The peer-to-peer system will be able to pipe TV programs to set-top boxes and home TV sets. Indeed, the core technical goals of the project are to foster an open standards-based next-generation Internet TV distribution system, employing P2P and social interaction. The European Union's P2P Next project to develop an internet television distribution standard will build on Tribler technology under development at the Delft University of Technology[5].

- Telecommunication: nowadays, people are not just satisfied with "can hear a person from another side of the earth", instead, the demands of clearer voice in real-time are increasing globally. Just like the TV network, there are already cables in place, and it's not very likely for companies to change all the cables. Many of them turn to use the Internet, more specifically P2P networks. Furthermore, many research organizations are trying to apply P2P networks to cellular networks.

  In general, Internet telephony can be implemented without use of a P2P network, but using P2P networks can have advantages. One popular P2P Internet telephony and instant messaging program is Skype. Skype allows users to make voice calls and send instant messages to each other. In a non-P2P system, all of the voice and message packets are routed through a central server. In Skype, the packets are sent directly from one peer to another, or routed through other peers in the Skype network

[5]Tribler is an open source Peer-to-Peer client with various features for watching videos online. It supports standard features such as key word searching for content and segmented downloading.

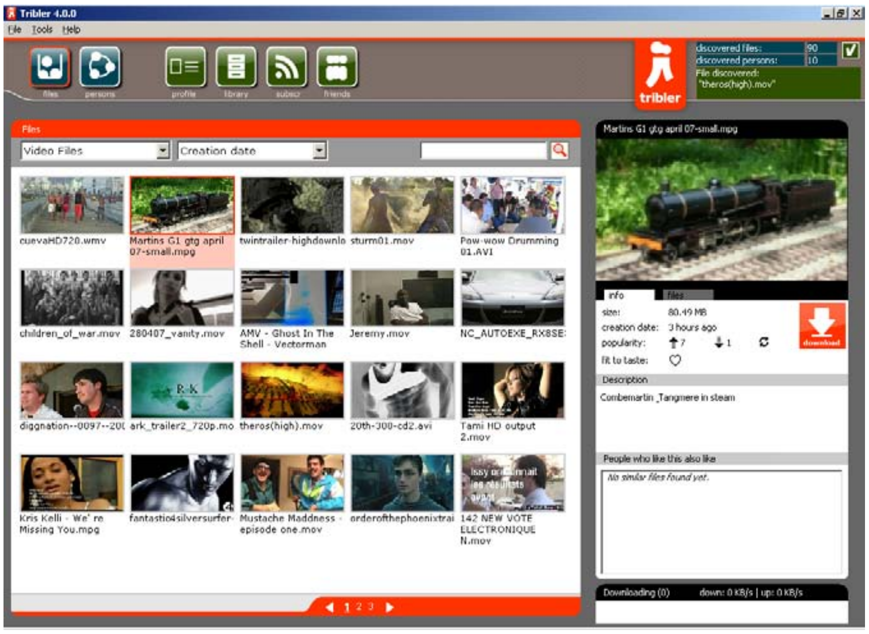

Tribler 4.0.0
File Tools Help
files persons profile library subscr friends
tribler
Files
Video Files
Creation date
Martins G1 gtg april 07-small.mpg
download
Downloading (0) down: 0 KB/s | up: 0 KB/s

Figure 1.3: Tribler, a popular BitTorrent client.

when a direct connection between the two peers is not possible. As a consequence, the network can cheaply scale to millions of users because there is no need for costly centralized infrastructure. Currently, there are around two million users simultaneously using Skype at any given time.

1.2 A technical overview of BitTorrent

BitTorrent is a peer-to-peer file sharing protocol used to distribute large amounts of data. The protocol was designed [4] by Bram Cohen in April 2001 and the first implementation was released on 2 July 2001. Usage of the protocol accounts for significant Internet traffic, as we can see in figure 1.9 on page 27; though the precise amount has proven difficult to measure. Nowadays, there are numerous BitTorrent clients available for a variety of computing platforms. Each client is capable of preparing, requesting, and transmitting any type of computer file over a network using the protocol. A screenshot of Tribler, a popular BitTorrent client, is shown in figure 1.3.

Since there is an official specification which is not very detailed, we have done some experiments with a BitTorrent client (Vuze) and a packet sniffer (Wireshark) to reverse-engineer the protocol, using the (few) details published on the Internet. In the following pages, we describe our results in studying BitTorrent protocol.

1.2.1 Terminology

We will briefly define several terms commonly used in the BitTorrent system.

Torrent file A metadata file containing information of the file you want to download.

Piece A part of a file that has a size of a power of 2. A torrent is split up into pieces that are of the same size except for the last piece which may be less than the piece size. Piece sizes are listed in table 1.1 on the next page.

Sub-Piece/Block A part of a piece that is transferred over the wire at a time. The specification allows 2^{15} (32KB) requests. The reality is near all clients will now use 2^{14} (16KB) requests. Due to clients that enforce that size, it is recommended that implementations make requests of that size. Smaller requests will result in higher overhead due to tracking a greater number of requests.

Peer A peer is any computer running an instance of a client. An entity in the BitTorrent system that uploads/downloads file pieces.

Seed A peer that has a complete copy of the file and is uploading to the network.

Leecher A peer that does not have a full copy of the file and is downloading from the network.

Tracker A central entity that peers communicate with periodically to help them discover one another.

Swarm A group of peers connected to each other to share a torrent.

Bencode An encoding format used to encode torrent information in the .torrent file and tracker responses to peers. Bencoded messages contain nested dictionaries and lists, which contain string and integers.

1.2.2 Components of a BitTorrent system

A BitTorrent system consists of:

- a .torrent file containing information of the original file.
- a BitTorrent tracker that sends out and receives peer information and maintains peer statistics.
- one or more BitTorrent clients, of which one of them is the original seed that has the original copy of the file in mention.
- a web server hosting the .torrent file for users to download.

Figure 1.4 on page 11 represent a basic BitTorrent system.

Table 1.1: Piece size, block size and corresponding number of requests.

Piece size (KB)	Block size (fixed) (KB)	Number of requests
32	16	2
64	16	4
128	16	8
256	16	16
512	16	32
1024	16	64
2048	16	128
4096	16	256

1.2.3 Strengths

BitTorrent introduces a more distributed concept of file sharing. While downloading a file from other peers, a BitTorrent client uploads partially downloaded parts of the file. Note that, assuming zero peer failures, the initial seed peer only needs to transmit each piece of the file once into the swarm, and it is possible for every peer to receive a complete copy of the file, even in a swarm of thousands of peers. To contrast this with conventional web serving, a situation with thousands of requesting clients requires that the complete file is transmitted to every client. This insight demonstrates BitTorrent's scalable nature.

Though both ultimately transfer files over a network, a BitTorrent download differs from a classic full-file HTTP request in several fundamental ways:

- BitTorrent makes many small data requests over different TCP sockets, while web-browsers typically make a single HTTP GET request over a single TCP socket.

- BitTorrent downloads in a random or in a rarest first approach that ensures high availability, while HTTP downloads in a sequential manner.

Taken together, these differences allow BitTorrent to achieve:

- much lower cost (in fact, the upload cost is distributed and shared among its peers instead of being placed on a single host).

- much higher redundancy (pieces are replicated among peers rather than on a single server).

- much greater resistance to abuse or to *flash crowds* than a regular HTTP server.

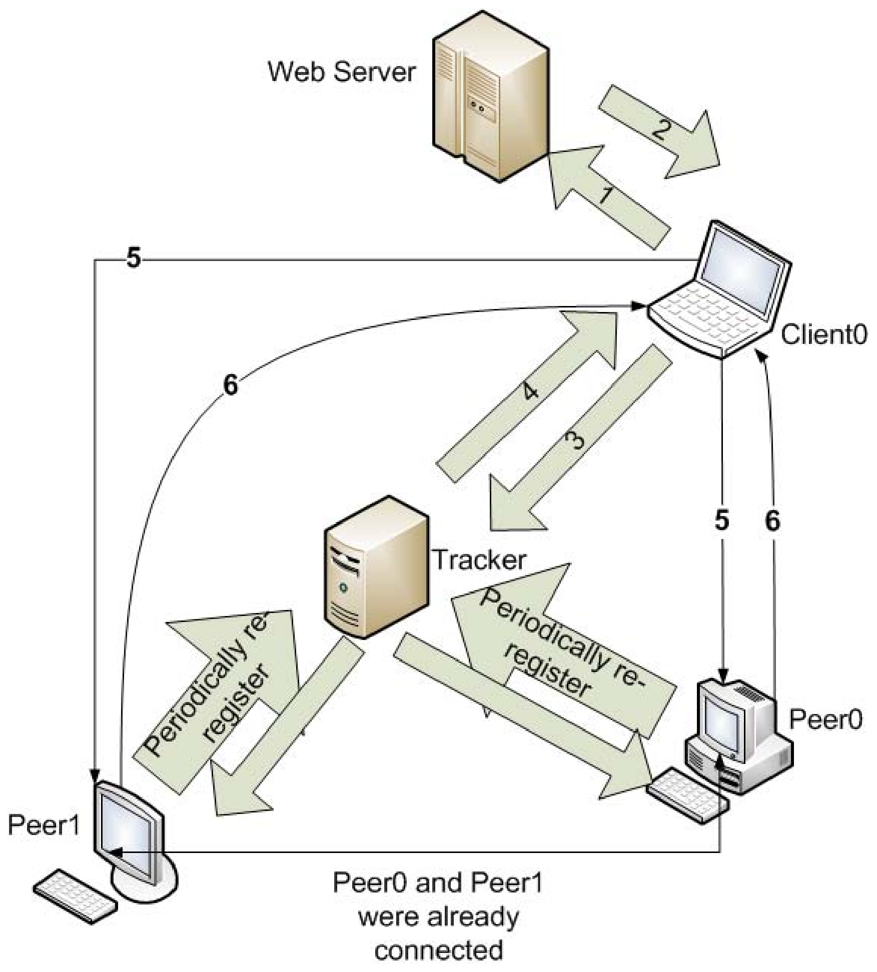


Figure 1.4: Components of a BitTorrent system.

One of the most interesting aspects of BitTorrent protocol is also the *tit-for-tat* mechanism, which specifies that every peer will have a download rate proportional to its upload rate. We will focus on tit-for-tat on paragraph 1.2.7 on page 23.

1.2.4 Weaknesses

Downloads can take time to rise to full speed because it may take time for enough peer connections to be established, and it takes time for a node to receive sufficient data to become an effective uploader. As such, a typical BitTorrent download will gradually rise to very high speeds, and then slowly fall back down toward the end of the download. This contrasts with an HTTP server that, while more vulnerable to overload and abuse, rises to full speed very quickly and maintains this speed throughout.

It is obvious that the fault tolerance of BitTorrent is much higher than HTTP. Even if a node fails, a downloader has other sources to download from. However, if the tracker fails, new downloaders cannot start downloading since they will not be able to look for peers to connect to[6]. Downloaders who have established connections with other peers before the tracker failed can continue to share the file as long as their peers have file pieces that they have yet to obtain and vice versa. However, they cannot look for new peers and a complete download of the file is not guaranteed.

The BitTorrent protocol did not specify any incentives to keep seeders from leaving the swarm. BitTorrent file sharers, compared to users of client/server technology, often have little incentive to become seeders after they finish downloading. With seeders leaving, the remaining downloaders will not be able to obtain a complete file; in fact, a torrent is alive as long as there is at least one seed in the torrent. Some BitTorrent websites have attempted to address this by recording each user's download and upload ratio (*share ratio*), as well as the provision of access to newer torrent files to people with better ratios. Users who have low upload ratios may see slower download speeds until they upload more. This prevents (statistical) leeching, since after a while they become unable to download much faster than 1-10 kB/s on a high-speed connection. Some trackers exempt dial-up users from this policy, because they cannot upload faster than 1-3 kB/s.

It is considered good etiquette to keep share ratio equal or double one's leeching. This provides an opportunity to reciprocate what a peer has downloaded, supporting the torrent and nature of the protocol. While this is usually most easily accomplished with a DSL connection, those using an ADSL (*Asymmetric* Digital Subscriber Line) or dial-up will not be able to conform easily to this rule of etiquette (as they have more downstream bandwidth than upstream). To combat this leeching problem, some seeders deliberately withhold one final piece from the seed, thus leaving a large number of potential seeders once they receive the withheld piece of data. With clients each awaiting that one final piece, the seeder ensures that there will be many more seeds once the final piece is released.

There are cheating clients like BitTyrant [27] and BitThief [18] which claim to be able to download without uploading. Such exploitation negatively affects the cooperative nature of the BitTorrent protocol.

1.2.5 Torrent files

Torrent files are metadata files created prior to sharing a file or files, which we will call an entity. These .torrent files contain information that a Bittorrent client requires to download an entity, but not the entity itself. The metadata is bencoded with the following information:

[6]In this case we are considering a single-tracker system.

announce URL of the tracker.

info A dictionary containing the following keys:

- **name** A suggested name to save the entity. If the entity is a single file, this key represents a file name. If it consists of multiple files, then this key maps to a directory name.
- **piece length** The size of each piece of the entity.
- **pieces** A string containing the concatenation of all SHA1 hashes of each piece of the entity.
- **length** The length of the file in bytes. If this key is present, it means that the entity is a single file. In this case, the key, files, will not be present. If the entity to be downloaded is a directory of multiple files, files will be present instead of length.
- **files** A files list consisting of a list of dictionaries with the following keys.
  - **length** The length of the file in bytes.
  - **path** A list containing one or more string elements that together represent the path and filename. Each element in the list corresponds to either a directory name or (in the case of the final element) the filename. For example, the file `dir1/dir2/file.ext` would consist of three string elements: `dir1`, `dir2`, and `file.ext`.

1.2.6 Tracker

The tracker is the only centralized component of BitTorrent, but it is not involved in the actual distribution of file. It only keeps track of the peers that currently take part in the torrent and it collects statistics on the torrent itself; tracker server knows only if a peer is downloading or if it just seeding. A tracker can coordinate peers for more than one torrent. It organizes sets of peers according to the info hash of the entity they are downloading[7].

The tracker is an HTTP/HTTPS service which responds to HTTP GET requests. The requests include metrics from clients that help the tracker keep overall statistics about the torrent. The response includes a peer list that helps the client participate in the torrent.

Indexing

The BitTorrent protocol provides no way to index torrent files. As a result, a comparatively small number of websites have hosted the large majority

[7]The `info_hash` are the 20 byte SHA1 hash of the bencoded form of the info value from the metainfo file.

of torrents linking to (possibly) copyrighted material, rendering those sites especially vulnerable to lawsuits. Several types of websites support the discovery and distribution of data on the BitTorrent network.

Public tracker sites such as *The Pirate Bay* allow users to search in and download from their collection of .torrent files; they also run BitTorrent trackers for those files[8]. Users can typically also upload .torrent files for content they wish to distribute. Private tracker sites such as Demonoid operate like public ones except that they restrict access to registered users and keep track of the amount of data each user uploads and downloads, in an attempt to reduce leeching.

1.2.7 Clients

To share a file or group of files, a peer first creates a small file called *torrent*. Peers that want to download the file must first obtain a torrent file for it, and connect to the specified tracker, which tells them from which other peers to download the pieces of the file. The peer distributing a data file treats the file as a number of identically-sized pieces (except for the last one). The peer creates a checksum for each piece, using the SHA1 hashing algorithm, and records it in the torrent file. Pieces with sizes greater than 512 kB will reduce the size of a torrent file for a very large payload, but is claimed to reduce the efficiency of the protocol. When another peer later receives a particular piece, the checksum of the piece is compared to the recorded checksum to test that the piece is error-free[9]. Torrent files are typically published on websites or elsewhere, and registered with a tracker. The tracker maintains lists of the clients currently participating in the torrent. Alternatively, in a tracker less system (decentralized) every peer acts as a tracker.

Users browse the web to find a torrent of interest, download it, and open it with a BitTorrent client. The client connects to the tracker(s) specified in the torrent file, from which it receives a list of peers currently transferring pieces of the file(s) specified in the torrent. The client connects to those peers to obtain the various pieces. If the swarm contains only the initial seeder, the client connects directly to it and begins to request pieces. As peers enter the swarm, they begin to trade pieces with one another, instead of downloading directly from the seeder.

[8]In the years to come, *The Pirate Bay* established itself as the largest BitTorrent tracker on the Internet. Over the past 12 months, the number of peers connected to tracker has tripled to 25 million.

[9]Clearly, there's a trade off between the number of hashes to calculate and publish and the discarded bytes due to a piece which does not pass the hash check. In fact, if a piece does not pass hash check, it is discarded and the client re-downloads it.

Downloading in details

The following describes the steps involved for a client to download a file using BitTorrent.

1. a downloader, `client0`, starts off by downloading a .torrent file corresponding to the file it wants to obtain. In figure 1.4 on page 11, this step is represented by 1 and 2. Upon receiving the .torrent file, the BitTorrent client starts. The downloader then selects the path to save his file.

2. upon initialization, a file of that save path is created with length 0. The peer first connects to the tracker to request a set of IP addresses of remote peers that are in the swarm. The tracker returns a *random* list of IPs belonging to peers that are leeching or seeding the torrent. The returned list is a fixed-length subset of the full list the tracker maintains. The request for a set of remote peer IP addresses allows the tracker to add the requesting peer to its list of peer IP address. If the number of peers of a node ever dips below 20, say due to the departure of peers, the node contacts the tracker again to obtain a list of additional peers it could connect to. In figure 1.4 on page 11, 3 and 4 illustrate this step. The tracker protocol uses HTTP. Peers make HTTP GET requests and the tracker sends responses in the returning HTTP response data.

3. after the peer has a partial list of peers in the swarm, it picks a certain amount of them at random (peers in `client0`'s ban list are excluded), and attempts to connect to them[10]. The success of the connections are determined by several factors:

   - `client0` must not already be connected to that peer.
   - that peer must not be a pending connection of `client0`.
   - `client0` and the remote peer have not reached the maximum number of TCP connections allowed by the operating system.

   The number of outgoing connection request ranges from four to around thirty, depending on user configurable settings and peer implementation. The peer also listens on a network port, by default 6881 TCP but also user configurable, to allow remote peers that are also attempting to connect to other remote peers to connect to the peer. The peer should start to receive connections after it gives the tracker its IP address, because remote peers will receive this IP address from the tracker and start to connect to the peer, assuming there are other peers in the swarm. Each peer in the swarm aims to maintain the certain number

[10]See chapter 4 for further details on ban lists.

of connections to remote peers, without consideration of which peer initiated the connection.

The peer wire protocol operates over TCP, and uses in-band signaling for peer communication. Signaling and data transfer are done in the form of a continuous bi-directional stream of fixed-size, length-prefixed protocol messages. A P2P session is equivalent with a TCP session, and there are no protocol entities for tearing down a BitTorrent session beyond the TCP teardown itself.

4. a handshake is done for each successful socket connection. This is represented by 5 in figure 1.4 on page 11. The handshake is done in both directions. If the receiving peer replies with the corresponding information, the BitTorrent session is considered to be opened and the peers start exchanging messages across the TCP streams. In other cases, the TCP connection is closed. The message sent during a handshake has the following format:

   ```
   <protocol_length><protocol><flags><info_hash><peer_id>
   ```

   - `protocol_length`: length of the protocol name, which is "BitTorrent protocol". As a consequence, `protocol_length` is 19.
   - `protocol`: protocol used in the system, which is "BitTorrent protocol".
   - `flags`: 8 flag bytes. Currently the first 7 bytes are not in use (they are reserved for future features) and are set to 0. The last of the 8 bytes is set to 1 if this client supports distributed hash tables (DHT), otherwise it is set to 0[11].
   - `info_hash`: SHA1 hash of the info value of the .torrent metadata file. This value must match that of the receiving client's. This is the same `info_hash` that is transmitted in tracker requests.
   - `peer_id`: 20-byte string used as a unique ID for the client. The recipient will check its own ID to see if the ID sent by the initiating client matches the one it expects. If they do not match, both sides will drop the connection immediately. This is usually the same `peer_id` that is transmitted in tracker requests.

   The initiator of a connection is expected to transmit its handshake immediately. Immediately following the handshake procedure, each peer sends information about the pieces of the resource it possesses. This is done only once, and only by using the first message after the handshake. The information is sent in a `BITFIELD` message, consisting of a stream of bits, with each bit index corresponding to a piece index. Each bit raised corresponds to a piece that the client has.

[11]See section 1.2.11 on page 32 for further details on DHT.

5. the connection is established and peers communicate via an exchange of length-prefixed messages (described afterwards). This is represented by 6 in figure 1.4 on page 11.

   A peer maintains two states for each peer relationship, namely *interested* and *choked*.

   Choked state is whether or not the remote peer has choked this client. When a peer chokes the client, it is a notification that no requests will be answered until the client is unchoked. The client should not attempt to send requests for blocks, and it should consider all pending (unanswered) requests to be discarded by the remote peer. For a peer to be allowed to download, it must have received an unchoke message from the sending peer. Once a peer receives a choke message, it will no longer be allowed to download. This allows the sending peer to keep track of the peers that start downloading when unchoked.

   Interested state is whether or not the remote peer is interested in something this client has to offer. This is a notification that the remote peer will begin requesting blocks when the client unchokes them. Interest should be expressed explicitly, as should lack of interest. That means that a peer wishing to download notifies the sending peer (where the sought data is) by sending an interested message, and as soon as the peer no longer needs any other data, a not interested message is issued. A peer with all data, i.e., a seed, is never interested.

   Connections start out choked and not interested. It is important for the client to keep its peers informed as to whether or not it is interested in them. This state information should be kept up-to-date with each peer even when the client is choked. This will allow peers to know if the client will begin downloading when it is unchoked (and vice-versa). These states are updated using `CHOKE`, `UNCHOKE`, `INTERESTED` ad `UNINTERESTED` messages. Figure 1.5 on the next page summarizes the state transition of a connection with respect to `client0`.

   An unchoked and interested peer will send a `REQUEST` message to request (part of) a piece. A request by an unchoked peer is always granted and the relevant data sent back using a `PIECE` message. Once a piece has been fully received and its hash is confirmed, the peer will broadcast a `HAVE` message to its neighbours. If a neighbour becomes interested after having received the `HAVE` message, it will send an `INTERESTED` message, wait for an `UNCHOKE` message, send a `REQUEST` message and finally receive the `PIECE` message containing the file data. In the described scenario, it thus takes at least 2.5 round-trips for a piece to hop from one peer to the next. If the neighbour was already interested and unchoked

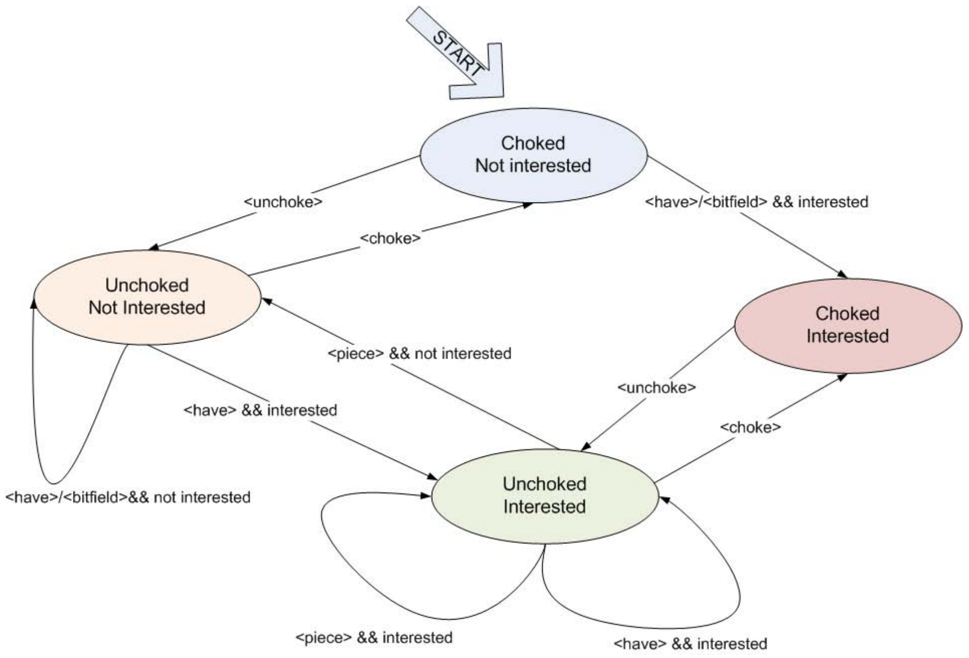


Figure 1.5: State-transition diagram of a BitTorrent connection with respect to `client0`. Messages sent from the other side of the connection, i.e. `peer1`, are represented in the angle brackets, <>. The labels "interested" and "not interested" represent whether `client0` is interested in what `peer1` has to offer.

when it receives the `HAVE` message, the delay is reduced to at least 1.5 round-trips.

Peer wire protocol The peer protocol facilitates the exchange of messages between peers. Messages are in the following format:

```
<message_length><message_type><message>
```

- `message_length`: it is a four byte big-endian value.
- `message_type`: 1 byte value representing the type of message sent.
- `message`: Depending on `message_type`, each message has its own format. Thus, `message_length` will have differing values as well. The following gives the relationship between the `message_type` and `message` (as well as `message_length`). Note that `KEEP-ALIVE`, `CHOKE`, `UNCHOKE`, `INTERESTED` and `NOT_INTERESTED` do not have this field.
  - `KEEP-ALIVE`: `<0000>`
    The keep-alive message is a message with zero bytes, specified

with the length prefix set to zero. There is no message ID and no payload. Peers may close a connection if they receive no messages (keep-alive or any other message) for a certain period of time, so a keep-alive message must be sent by both parties to maintain the connection alive if no command have been sent for a given amount of time. This amount of time is generally two minutes.

– `CHOKE`: `<0001><0>`
  No data will be uploaded until unchoking occurs.
– `UNCHOKE`: `<0001><1>`
  Tell peer that its requests will be answered.
– `INTERESTED`: `<0001><2>`
  Tell peer that we have something to download from it.
– `NOT_INTERESTED`: `<0001><3>`
  Tell peer that we do not want to download from it.
– `HAVE`: `<0005><4><index>`
  As soon as a peer receives a full piece, it can transmit the piece to the remote peers it is connected to. This allows the pieces to be propagated through the swarm. With `HAVE` message, the peer announces that it have successfully downloaded a piece, characterized by its piece index.
– `BITFIELD`: `<0001 + bitfield length><5><bitfield>`
  The bitfield message must be sent by both parties immediately after the handshaking sequence is completed and before any other messages are sent. It has not to be sent if a client has no pieces. The payload is a bitfield representing the pieces that have been successfully downloaded. The high bit in the first byte corresponds to piece index 0. Bits that are cleared indicated a missing piece, and set bits indicate a valid and available piece. Spare bits at the end are set to zero. A bitfield of the wrong length is considered an error. Clients should drop the connection if they receive bitfields that are not of the correct size, or if the bitfield has any of the spare bits set.
– `REQUEST`: `<000d><6><[index][begin][length]>`
  Request for a block, which is a portion of a piece of that particular `index`, at offset `begin` and of that `length`[12]. `index`, `begin` and `length` are 4 bytes each. To allow TCP to increase the throughput, several requests are usually sent back-to-back. Each request should result in the corresponding block to be transmitted. If the block is not received within a certain time (typically one minute), the non-transmitting peer is *snubbed*,

[12]From now on, `length` is assumed to be equal to 16384 (as most of BitTorrent clients do) for all the blocks (except for the last one).

i.e. it is punished by not being allowed to download, even if unchoked. See figure 1.7 on page 22.

- PIECE: `<0009+block len><7><[index][begin][block]>`
  Send a portion of a piece of that `index` at offset `begin`. This corresponds to a `REQUEST` message sent by the recipient earlier. `index` and `begin` are 4 bytes each.
- CANCEL: `<000d><8><[index][begin][length]>`
  Cancel an earlier request of a piece of that particular `index`, at offset `begin` and of that `length`. `index`, `begin` and `length` are 4 bytes each.

Downloading strategies

Clients incorporate mechanisms to optimize their download and upload rates; for example they download pieces in a random order to increase the opportunity to exchange data, which is only possible if two peers have different pieces of the file. Therefore, piece selection is very crucial in the BitTorrent system, because wrong choices can result in not being able to download a complete file or not being able to upload to other peers. The effectiveness of this data exchange depends largely on the policies that clients use to determine to whom to send data.

Random first If the local peer has downloaded strictly less than 4 pieces, it has nothing to upload to other peers, and thus may take a longer time if it uses the rarest piece first policy. The client chooses the next piece to request at random. This is called the *random first policy.* Once it has downloaded at least 4 pieces, it chooses the next piece to download at random in the rarest pieces set. The aim of the random first policy is to permit a peer to download its first pieces faster than with a the rarest first policy, as it is important to have some pieces to reciprocate for the choke algorithm. Indeed, a piece chosen at random is likely to be more replicated than the rarest pieces, thus its download time will be in mean faster.

Strict priority Once a block has been requested, block of the same piece index will have a greater priority than any other blocks. The aim of the strict priority policy is to complete the download of a piece as fast as possible. As new clients can only start uploading when at least one complete piece is obtained, it is important to minimize the number of partially received pieces.

Rarest first BitTorrent applies an algorithm to encourage the spread of the rare resources called *rarest first algorithm.* The local peer maintains the number of copies in its peer set of each content piece by searching for the less

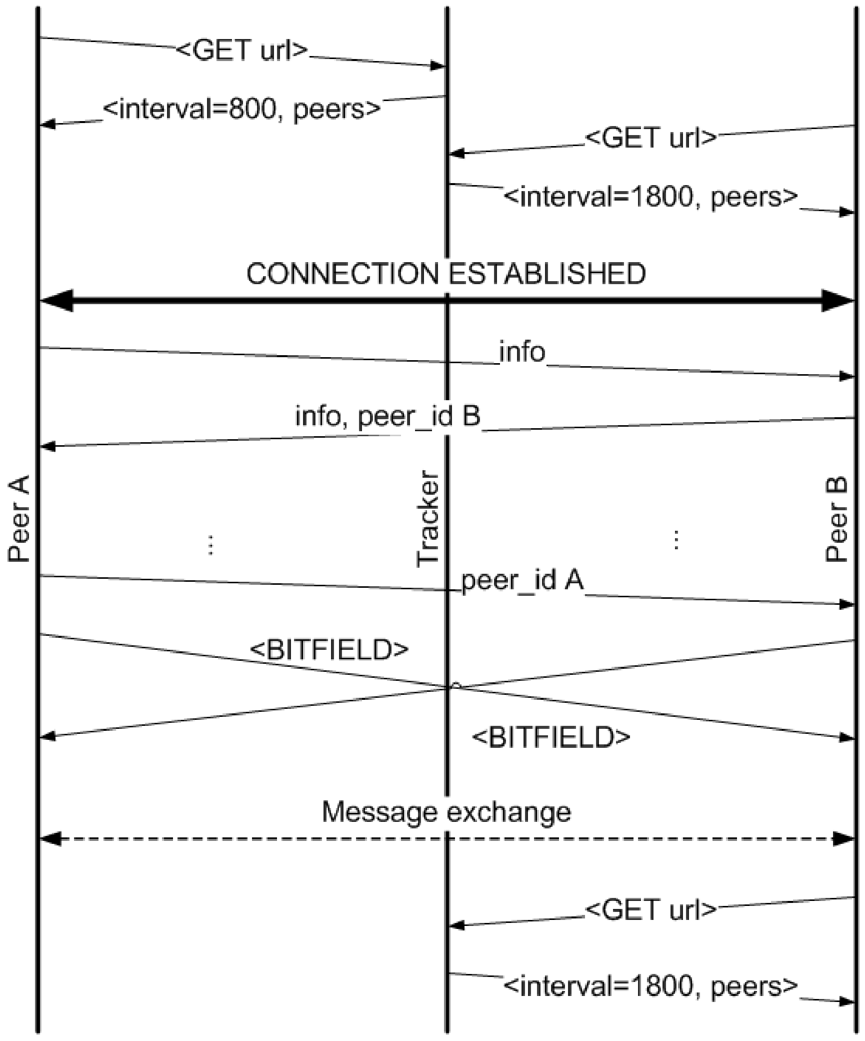


Figure 1.6: Node communications in BitTorrent.

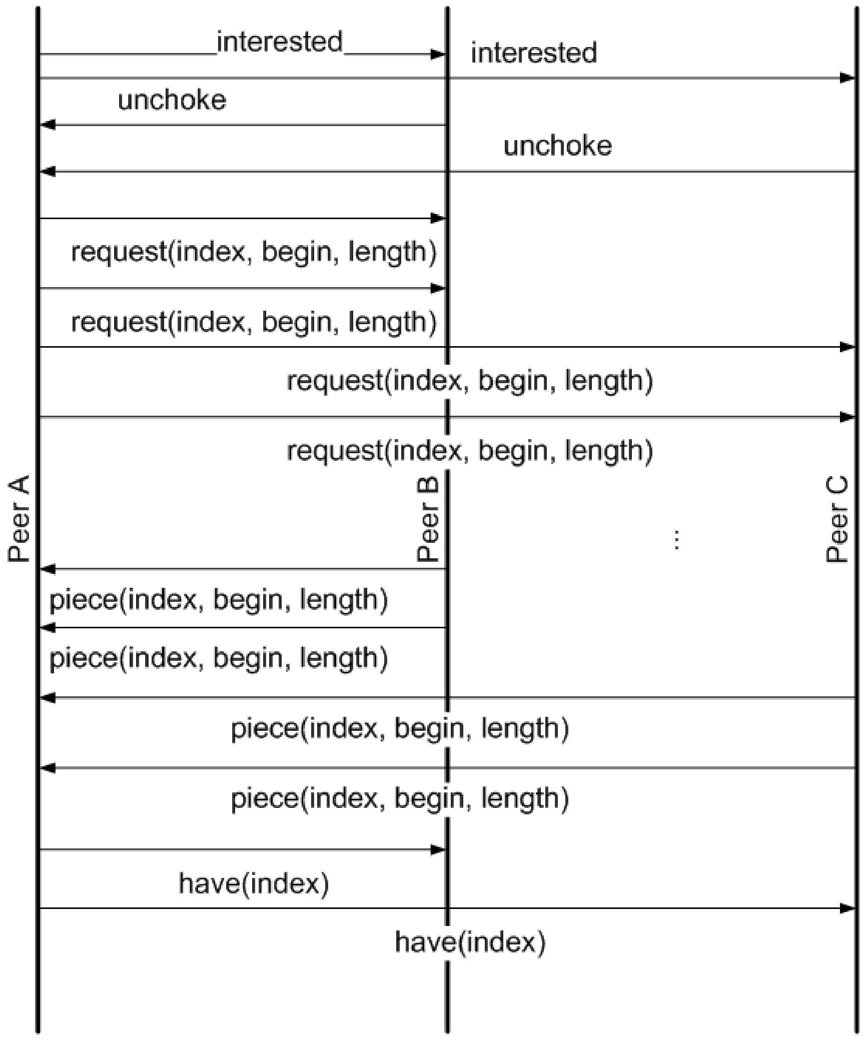

Peer A
Peer B
Peer C
interested
interested
unchoke
unchoke
request(index, begin, length)
request(index, begin, length)
request(index, begin, length)
request(index, begin, length)
⋮
piece(index, begin, length)
piece(index, begin, length)
piece(index, begin, length)
piece(index, begin, length)
have(index)
have(index)

Figure 1.7: BitTorrent protocol exchange.

replicated pieces: they will be the pieces that gain the highest priority[13]. The rarest pieces set is updated each time a copy of a piece is added to or removed from the peer set of the local peer. Note that any rarest first strategy should include randomization among at least several of the least common pieces, as having many clients all attempting to jump on the same least common piece would be counter productive. This ensures that clients will get the less common pieces faster to be able to upload to their peers. In this way there is a higher probability of peers getting complete files.

Endgame mode When the downloader is a few blocks away from obtaining the full copy of the file, there's a tendency for the last few blocks to trickle in slowly. Thus, the client switches in end-game mode. There is no documented thresholds, recommended percentages, or block counts that could be used as a guide. Some clients enter end game when all pieces have been requested. Others wait until the number of blocks left is lower than the number of blocks in transit, and no more than 20. There seems to be agreement that it's a good idea to keep the number of pending blocks low (1 or 2 blocks) to minimize the overhead, and if you randomize the blocks requested, there's a lower chance of downloading duplicates. During this mode, the peer requests all blocks not yet received to all peers in its peer set. Each time a block is received, the client will send `CANCEL` messages to all of its peers so as not to receive duplicate data and waste bandwidth. As a peer has a small buffer of pending requests, all blocks are effectively requested close to the end of the download. Therefore, the end game mode is used at the very end of the download, thus it has little impact on the overall performance. The goal for this design is to enable the downloader to finish downloading the last few blocks faster.

Choking algorithm

Free riders (cheating peers who download but do not upload) are a threat to the P2P system: having lots of these behaviors, the risk is to destroy the whole network. Thus the problem must not be underestimated: BitTorrent uses a tit-for-tat scheme to discourage free riders.

Tit-for-tat mechanism Besides being able to select the appropriate pieces to download at the appropriate times, a client must also find ways to get the most out of his own downstream bandwidth. The main concept is tit-for-tat, which encourages fair trading: between two persons, A and B, if A helps B, B will help A in the future, but no advantage comes if one of them does not cooperate. In BitTorrent, for each peer contacted, the client measures the amount of bytes uploaded and downloaded. The scheme dictates that

[13]The client can determine the less replicated pieces by keeping the initial bitfield from each peer, and updating it with every have message. Then, the client can download the pieces that appear least frequently in these peer bitfields.

each peer only uploads to a subset of remote peers that it is connected to – the remote peers that it is getting the highest current piece transfer rate from, out of all of the remote peers the peer is connected to. With most peers in the swarm following this rule, it is in each peer's best interest to upload to a remote peer that the peer is receiving pieces from. If the peer does not, the peer is likely to stop receiving pieces from the remote peer, because the remote peer is not getting a high enough download rate from the peer. In this way, clients decide for themselves how to maximize their download speeds by selecting peers that have the highest upload rates to them. As a result, to get a good download speed, a client must be able to upload as much as well. Tit-for-tat is achieved using the choking algorithm, described in the following paragraph. When a peer joins the swarm it has not any pieces to reciprocate to other peers: for this reason, the BitTorrent protocol introduces a *optimistic unchoking* strategy to allow newly-joined peers to begin the exchange of data. We will describe optimistic unchoke in the following paragraph.

Tit-for-tat strategy does not consider when there is exceeded capacity of service in the torrent. This situation is a fundamental property of P2P applications, not rare indeed. When a peer has an asymmetrical network connectivity, the upload capacity is lower than the download capacity. In this situation, tit-for-tat forbids who is downloading a file to use its full download capacity; even in case of spare capacity in the P2P session. This is a hard task to accomplish anyway. This is the main reason why tit-for-tat strategy is not reckoned as the best to punish free-riders; other strategies tend to have a more cooperative approach. This kind of problems goes under the name of game theory [38]. Concerning this subject Robert Aumann and Thomas Schelling (Nobel prizes in 2005) have focused on the importance of cooperation [41]. They sustain that in defined circumstances cooperation brings the two contestants to a better pay-off than any other strategy [11]. A promising improvement of tit-for-tat is *give-to-get*, implemented in Tribler client. Basically this algorithm broaden the reputation scheme: if a peer helps another peer, the helper peer gains reputation on all the peers of the swarm (in a tit-for-tat scheme, the helper peer gains reputation only on the helped peer). In addition, the helper gains reputation over time: in the future, it is probable that other peers may help it. This will allow give-to-get to achieve a better and more flexible way to encourage sharing among peers.

Choking and unchoking The choke algorithm was introduced to implement tit-for-tat mechanism, guarantying a reasonable level of upload and download reciprocation. As a consequence, free riders should be penalized. The choke algorithm makes an important distinction between the leecher state and the seed state. In this section, interested always means interested in the local peer, and choked always means choked by the remote peer.

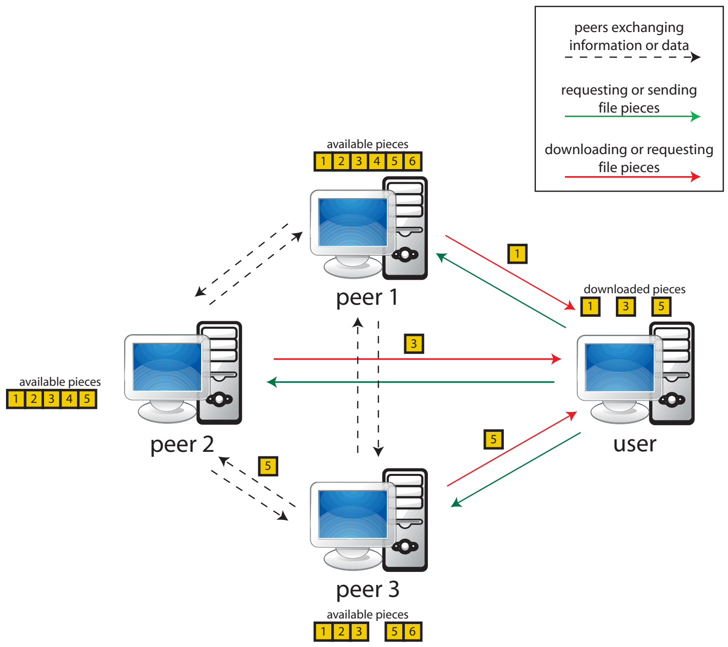


Figure 1.8: BitTorrent during download. In this case we are considering a trackerless system.

Choking is done for several reasons:

- TCP congestion control behaves very poorly when sending over many connections at once.
- give a proportional download rate for all peers and to prevent free riders.
- globally, a better utilization of the client's resources.

There are several criteria a good choking algorithm should meet:

- it should cap the number of simultaneous uploads for good TCP performance.
- it should avoid choking and unchoking quickly, known as *fibrillation*.
- it should reciprocate to peers who let it download.
- finally, it should try out unused connections once in a while to find out if they might be better than the currently used ones, known as *optimistic unchoking*.

When in leecher state, the choke algorithm is called every ten seconds to avoid fibrillation[14]. Reciprocation and number of uploads capping is

[14]Ten seconds is a long enough period of time for TCP to ramp up new transfers to their full capacity.

managed by unchoking the four peers which have the best upload rate and are interested. This maximizes the client's download rate. These four peers are referred to as downloaders, because they are interested in downloading from the client. Peers which have a better upload rate (as compared to the downloaders) but are not interested get unchoked. If they become interested, the downloader with the worst upload rate gets choked. If a client has a complete file, it uses its upload rate rather than its download rate to decide which peers to unchoke.

Optimistic unchoking Strict policies often result in suboptimal situations, such as when newly joined peers are unable to receive any data because they don't have any pieces yet to trade themselves or when two peers with a good connection between them do not exchange data simply because neither of them wants to take the initiative. To counter these effects, the official BitTorrent client program uses a mechanism called *optimistic unchoking*, where the client reserves a portion of its available bandwidth for sending pieces to random peers (not necessarily known-good partners, so called preferred peers). It allows to evaluate the download capacity of new peers in the peer set, and it allows to bootstrap new peers that do not have any piece to share by giving them their first piece.

At any one time there is a single peer which is unchoked regardless of its upload rate (if interested, it counts as one of the four allowed downloaders). Which peer is optimistically unchoked rotates every 30 seconds[15]. Newly connected peers are three times as likely to start as the current optimistic unchoke as anywhere else in the rotation. This gives them a decent chance of getting a complete piece to upload.

1.2.8 Network Impact

According to *isoHunt*[16] the size of the torrents is currently more than 1.1 Petabytes. CableLabs, the research organization of the North American cable industry, estimates that BitTorrent represents 18% of all broadband traffic. In 2004, CacheLogic put that number at roughly 35% of all traffic on the Internet. The discrepancies in these numbers are caused by differences in the method used to measure P2P traffic on the Internet. Routers that use NAT must maintain tables of source and destination IP addresses and ports. Typical home routers are limited to about 2000 table entries while some more expensive routers have larger table capacities. BitTorrent frequently contacts 300-500 servers per second rapidly filling the NAT tables; in addition, it

[15]Thirty seconds is enough time for the upload to get to full capacity, the download to reciprocate, and the download to get to full capacity.

[16]*isoHunt* is a BitTorrent index with over 1.4 million torrents in its database and 16 million peers from indexed torrents. With 7.4 million unique visitors as of May 2006, *isoHunt* is one of the most popular BitTorrent search engines.

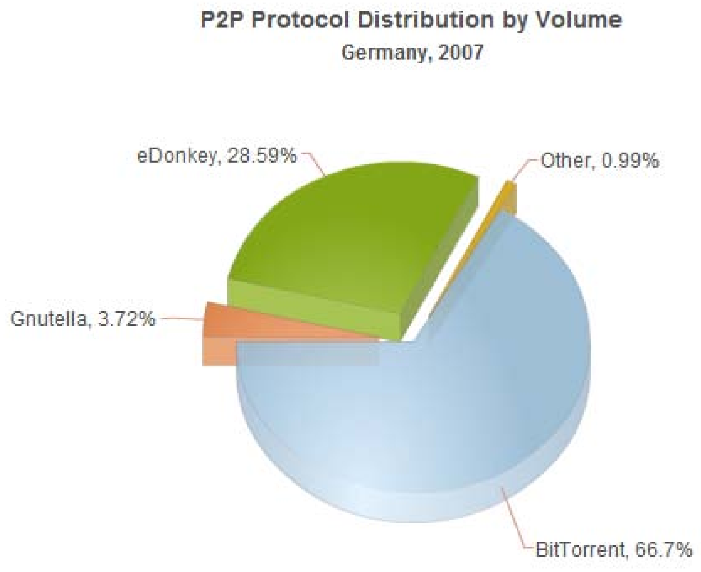


Figure 1.9: Ipoque, a German ISP, released a report [43] on P2P traffic usage in 2007. We can see that BitTorrent is the most used P2P program.

consumes very quickly ISPs bandwidth. For this reason ISPs may decide to throttle the bandwidth of those customers who frequently use P2P programs[17] [46].

1.2.9 BitTorrent adoption

A growing number of individuals and organizations are using BitTorrent to distribute their own or licensed material. Independent adopters report that without using BitTorrent technology and its dramatically reduced demands on networking hardware and bandwidth, they could not afford to distribute their files.

Film, video and music

In general, BitTorrent's non-contiguous download methods have prevented it from supporting progressive downloads or streaming playback. But comments made by Bram Cohen in January 2007 suggest that streaming torrent

[17]Over the past weeks more and more, Comcast users started to notice that their BitTorrent transfers were cut off. Most users report a significant decrease in download speeds, and even worse, they are unable to seed their downloads. A nightmare for people who want to keep up a positive ratio at private trackers and for the speed of BitTorrent transfers in general.

downloads will soon be commonplace and ad-supported streaming appears to be the result of those comments.

- BitTorrent Inc. has amassed a number of licenses from Hollywood studios for distributing popular content at the company's website.
- movies that have entered the public domain are available over BitTorrent. Because they are no longer profitable, there is little business incentive to pay for bandwidth hosting costs. BitTorrent allows these films to reach audiences.
- `Ourmedia.org`, the Open Media Directory, offers free and legal public domain movies through BitTorrent.
- Nine Inch Nails have offered free downloads of their last few albums. For the higher quality (i.e., larger file sized) versions of the tracks, Nine Inch Nails leader, Trent Reznor, has chosen to use BitTorrent to distribute his music. In a text file distributed with the album *Ghosts I*, Reznor states "Now that we're no longer constrained by a record label, we've decided to personally upload *Ghosts I*, the first of the four volumes, to various torrent sites, because we believe BitTorrent is a revolutionary digital distribution method, and we believe in finding ways to utilize new technologies instead of fighting them...". He has also released via BitTorrent the original Garageband files for fans to mix and create new content.
- Sub Pop Records releases tracks and videos via BitTorrent Inc. to distribute its 1000+ albums. The band *Ween* uses BitTorrent to distribute free audio and video recordings of live shows. Furthermore, *Babyshambles* and *The Libertines* (both bands associated with Pete Doherty) have extensively used torrents to distribute hundreds of demos and live videos.
- podcasting software is starting to integrate BitTorrent to help podcasters deal with the download demands of their MP3 radio programs. Specifically, Juice and Miro (formerly known as Democracy Player) support automatic processing of .torrent files from RSS feeds[18]. Similarly, some BitTorrent clients, such as μTorrent, are able to process web feeds and automatically download content found within them.
- music sharing portal *Jamendo* offers a free, legal and unlimited music from over 5500 artists via BitTorrent.
- public domain audiobooks are available at `legaltorrents.com`.

[18]RSS is a family of Web feed formats used to publish frequently updated works – such as blog entries, news headlines, audio, and video – in a standardized format. An RSS document (which is called a *feed*, *web feed*, or *channel*) includes full or summarized text, plus metadata such as publishing dates and authorship.

Broadcasters

- in 2008 CBC became the first public broadcaster in North America to make a full show (*Canada's Next Great Prime Minister*) available for download using BitTorrent.

- the Norwegian Broadcasting Corporation (NRK) have since March 2008 experimented with BitTorrent distribution from this site. Only selected material in which NRK owns all royalties are published. Responses have been very positive, and NRK is planning to offer more content.

Software

- Blizzard Entertainment uses BitTorrent (via a proprietary client called the "*Blizzard Downloader*") to update their *World of Warcraft game.*

- many major open source and free software projects encourage BitTorrent as well as conventional downloads (HTTP, FTP, . . .) of their products to increase availability and reduce load on their own servers, especially when dealing with larger files.

- distributing system patches: apparently, software updates are getting so big these days that simply downloading them from a server is becoming prohibitively time consuming, especially when the same updates need to be applied to many different machines[19].

  A Dutch university has some 6500 desktop PCs in ten locations, which on occasion need to download 3.5GB worth of different types of updates. That's a handsome 22.2TB in total. In a traditional client-server world, that's some modest lifting. In fact, INHOLLAND University's IT department used to have almost two dozen servers distributed over the university's locations to serve up these downloads. The school was able to retire 20 of them after adopting BitTorrent to distribute updates. The peer-to-peer protocol allows PCs to download most of the updates from each other – the remaining servers are mostly needed to send out the first few copies and then coordinate the up – and downloading. One of the advantages of the BitTorrent protocol is that it uses bandwidth where it can find it: faster links are automatically used more. *Using this technology, updating all 6,500 PCs can be done in less than four hours. Previously, this took four days.*

  The university's management team was reluctant to adopt the peer-to-peer technology, but they quickly changed their minds after seeing a demonstration. Students and staff who think they can use the modified

[19]Microsoft is currently developing a Windows Update mechanism [8] based on BitTorrent.

BitTorrent client for other purposes will be disappointed to learn that the system is completely locked down.

We identify three main scenarios where BitTorrent could be used:

- download of popular files: if peers cooperate among them, this is the only case where BitTorrent works perfectly. Nevertheless, peers do not necessarily cooperate: most of the people have an asymmetric bandwidth (i.e. ADSL), so the upstream capacity is less than downstream. In addition, people close the BitTorrent client whenever the download process is finished – this will decrease the seeders in the swarm, leading to a slower download process for other users. To gain efficiency in downloading, files must be popular (many peers are interested in them). However, as *long tail* theory explained, popular files are just too few, and they are dominated by less popular files that most *different* users want. In other words, long tail contains many users which have different needs, whilst the popular peak contains users that almost want the same files, which are, in this way, shared and downloaded by much more people. Such files become popular and gain success on BitTorrent swarms.

- progressive download or streaming (Video-on-demand): the order of the received pieces matters. At the moment, there is not a download strategy to prioritize more urgent pieces (those near to playout time). In addition, we can encounter start-up delay because the receiver waits for the buffer to be filled to avoid interruptions. As a consequence, TV zapping is not possible and the interactivity is much more reduced.

- live streaming for movies, and music: the order of the received pieces matters and we require minimum latency on start-up. In addition, the distribution must be very fast: in this way, there would not be a delay between the transmission event and its reception. On the minimum delay, we can use some results [17] to determine the best scenario, where upload must be done in sequential mode using a fastest peers first strategy (which involves a "keep uploading" policy rather than a "upload for given time interval"). We may also consider that live events have not a defined duration. . . how could we handle the download and validation process? There is a study [25] that describe a solution of this problem: briefly, they propose a rotating sliding window over a fixed set of pieces in order to project an infinite video stream onto a fixed number of pieces.

1.2.10 Legal issues

There has been much controversy over the use of BitTorrent trackers. BitTorrent metafiles themselves do not store copyrighted data, hence BitTorrent itself is not illegal – it is the use of it to copy copyrighted material that contravenes laws in some locations.

There are two major differences between BitTorrent and many other peer-to-peer file-trading systems, which advocates suggest make it less useful to those sharing copyrighted material without authorization. First, BitTorrent itself does not offer a search facility to find files by name. A user must find the initial torrent file by other means, such as a web search. Second, BitTorrent makes no attempt to conceal the host ultimately responsible for facilitating the sharing: a person who wishes to make a file available must run a tracker on a specific host or hosts and distribute the tracker address(es) in the .torrent file. Because it is possible to operate a tracker on a server that is located in a jurisdiction where the copyright holder cannot take legal action, the protocol does offer some vulnerability that other protocols lack. It is far easier to request that the server's ISP shut down the site than it is to find and identify every user sharing a file on a peer-to-peer network. However, with the use of a distributed hash table (DHT), trackers are no longer required, though often used for client software that does not support DHT to connect to the stream.

1.2.11 Technologies built on BitTorrent

The BitTorrent protocol is still under development and therefore may still acquire new features and other enhancements such as improved efficiency. In the following paragraphs we are going to describe only a few of those enhancements.

Multitracker

Multitracker is an extension to the BitTorrent metadata format proposed by John Hoffman and implemented by several indexing websites. It allows the use of multiple trackers per file, so if one tracker fails, others can continue supporting file transfer. It is implemented in several clients, such as Vuze, BitComet, BitTornado, KTorrent and μTorrent. Trackers are placed in groups, or tiers, with a tracker randomly chosen from the top tier and tried, moving to the next tier if all the trackers in the top tier fail.

Torrents with multiple trackers can decrease the time it takes to download a file, but also has a few consequences:

- users have to contact more trackers, leading to more overhead-traffic.
- torrents from private trackers suddenly become downloadable by non-members, as they can connect to a seed via an open tracker.

Distributed trackers

In June 2005, BitTorrent, Inc. released version 4.2.0 of the Mainline BitTorrent client. This release supported trackerless torrents, featuring a DHT implementation which allowed the client to use torrents that do not have a working BitTorrent tracker. Current versions of the official BitTorrent client, μTorrent, BitComet, and BitSpirit all share a compatible DHT implementation that is based on Kademlia. Vuze uses its own incompatible DHT system called the distributed database, but a plugin is available which allows use of the mainline DHT.

Most BitTorrent clients also use *peer exchange* (PEX) to gather peers in addition to trackers and DHT. Peer exchange checks with known peers to see if they know of any other peers. With the 3.0.5.0 release of Azureus, now known as Vuze, all major BitTorrent clients now have compatible peer exchange.

RSS feeds

A technique called *broadcatching* combines RSS with the BitTorrent protocol to create a content delivery system, further simplifying and automating content distribution. The RSS feed will track the content, while BitTorrent ensures content integrity with cryptographic hashing of all data, so feed subscribers will receive uncorrupted content.

One of the first software clients (free and open source) for broadcatching is Miro. Other free software clients such as PenguinTV and KatchTV are also now supporting broadcatching.

Encryption

Since BitTorrent makes up a large proportion of total traffic, some ISPs have chosen to throttle (slow down) BitTorrent transfers to ensure network capacity remains available for other uses. For this reason methods have been developed to disguise BitTorrent traffic in an attempt to thwart these efforts.

Protocol header encrypt (PHE) and Message stream encryption/Protocol encryption (MSE/PE) are features of some BitTorrent clients that attempt to make BitTorrent hard to detect and throttle. At the moment Vuze, Bitcomet, KTorrent, Transmission, Deluge, μTorrent, MooPolice, Halite, rTorrent and the latest official BitTorrent client (v6) support MSE/PE encryption.

In general, although encryption can make it difficult to determine what is being shared, BitTorrent is vulnerable to traffic analysis. Thus even with MSE/PE, it may be possible for an ISP to recognize BitTorrent and also to determine that a system is no longer downloading, only uploading, information and terminate its connection by injecting TCP RST (reset flag) packets.

Decentralized keyword search

Even with distributed trackers, a third party is still required to find a specific torrent. This is usually done in the form of a direct hyperlink from the website of the content owner or through indexing websites like The Pirate Bay or Torrentz.

In May 2007 Cornell University published a paper proposing a new approach to searching a peer-to-peer network for inexact strings [35] which could replace the functionality of a central indexing site. A year later, the same team implemented the system as a plugin for Vuze called Cubit and published a follow-up paper reporting its success [34].

Chapter 2

Digital Fountain Codes

2.1 Noisy Channel Features

From “A Mathematical Theory of Communication”, the famous article written by Claude Shannon in 1948, *Information Theory* evolved by error correction codes. They are essential to put into communication always new and more powerful electronic devices. Noise is present in every type of communication, it is caused by imperfections in the channel where the message travels. Talking with other people is an example of a communication system. The voice we produce may encounter other sounds, or modifications: it may be because of the wind. In the worst conditions, there may be a total misunderstood between speaker and listener.

In figure 2.1 on the next page, we can see that the message is added with noise, passing through the channel and it modifies itself. This system is not reliable, because it does not have any correction of the distorted message. According to the type of signal, we may distinguish three different communication systems:

- discrete: digital devices.
- continuous: analog devices such as radio and analog television.
- mixed: PCM acronym of Pulse Code Modulation, the conversion of signals from analog to digital.

We are going to use bits so our reference model is the discrete one, moreover our system must be without memory:

- output at time i depends only on input at time i.
- input and output alphabet is finite, where alphabet is the set of symbols used to communicate.

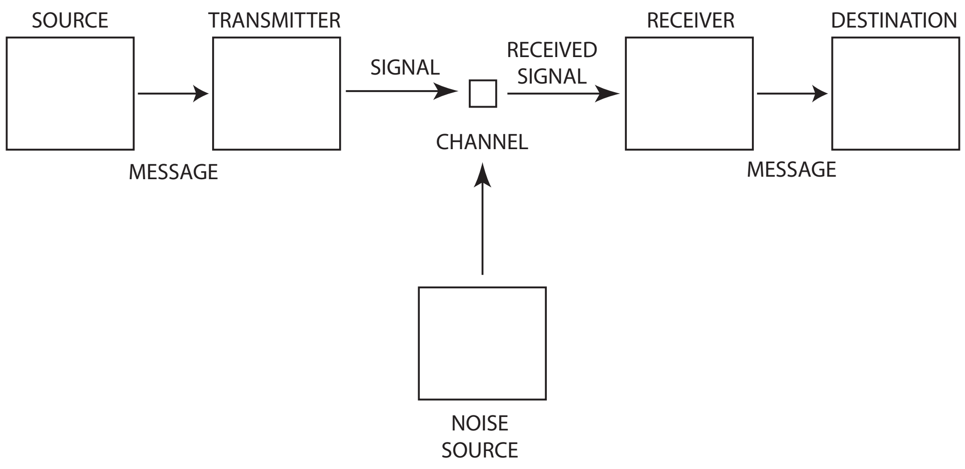


Figure 2.1: Theoretical model of a communication system.

In a channel without noise, a source can send a precise amount of information. This is the *capacity of a noiseless channel* [36]:

$$C = \lim_{T\to\infty} \frac{N(T)}{T}$$

$N(T)$ is the number of allowed signals of duration T. We may even calculate its *entropy*:

$$H = -K \sum_{i=1}^{N} p_i \cdot \ln p_i$$

where p_i is the set of probabilities of each transmitted symbol. In a binary channel we obtain:

$$H = -(p \cdot \log_2 p + q \cdot \log_2 q)$$

p is the probability to transmit one symbol and $q = 1 - p$ is the probability to transmit the other symbol.

Now we have all the elements to define the *capacity of a channel with noise* having an input x and an output y:

$$C = \max(H(x) - H_y(x))$$

$H_y(x)$ is the entropy of the input when the output is given, for convenience it is called *equivocation*. It is a conditional probability and measures the average ambiguity of the received signal. It is usually referred as the error rate. Let us give the definition of *rate* for any transmission:

$$R = H(y) - H_y(x)$$

According to the Shannon theorem [36], it is impossible to send data at a rate over the capacity of the channel: this is the limit that the source has, to guarantee the integrity of information it sends.

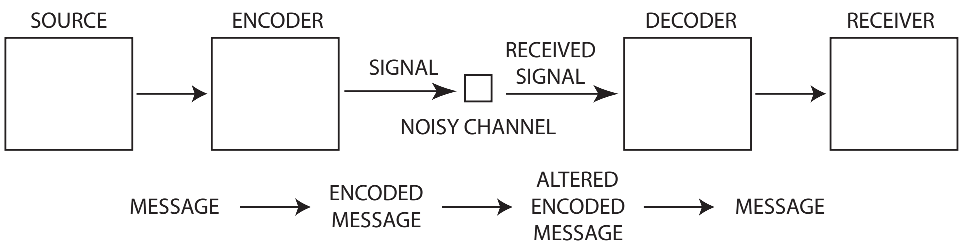


Figure 2.2: Theoretical model of a communication system with encoder and decoder. If the alteration of the message is kept within estimated limits, the decoder can recover the original message correctly.

To improve the reliability of communications, we may follow both a physical and a computational approach. The first has the aim to reduce the noise in the devices. The second tries to correct the noisy messages that arrive at destination, according to the possible error nature. The two methods are often combined in projects of electronic engineering. The difficulty is to adopt the right compromise between these solutions in order to satisfy the budget costs.

Since we are at the beginning of the project, we focus on *the computational approach*. This leads us to create a new diagram model of a general communication system (figure 2.2).

2.2 Channel Models

Whatever the source transmits, data may encounter two destinies:

1. to arrive modified: the receiver may misunderstood the message.

2. not to arrive at all: the receiver does not get the message at all.

With a binary alphabet, these situations are explained through two models respectively: Binary Symmetric Channel (BSC) and Binary Erasure Channel (BEC).

What may happen in a BSC is represented in figure 2.3 on the next page. The output could get a different message. Here the risk is to send 0 and receive 1 and vice versa. f is called either failure probability or flip probability. Figure 2.4 on the following page shows the diagram of a BEC. Error correcting codes (ECC) are used when dealing with BSC channels. This time data will be correct, if they arrive at destination; if they got lost, the receiver does not have any message. e is called erasure probability. An example of BEC is the Internet. Erasure recovery codes are used with BEC channels.

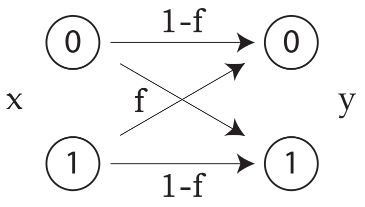


Figure 2.3: A BSC with its probabilities of failure (f) and success ($1 - f$) in sending one bit from a source x to a receiver y.

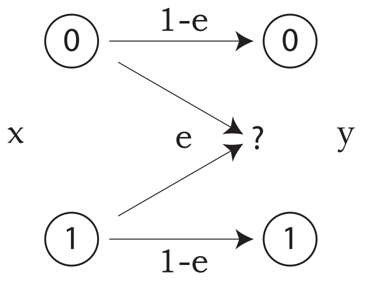


Figure 2.4: A BEC with its probabilities of erasure (e) and success ($1 - e$) in sending one bit from a source x to a receiver y.

It is difficult to find real channels belonging to one type or the other. Usually they are the superimposition of BEC and BSC. These models are very high level simplifications of actual physics means of communication. But they help avoiding too complex computations in the study of reality. Since we are going to investigate data transmission mainly over the Internet, we assume that our exchange of information takes place in a BEC.

2.3 ARQ: Automatic Repeat and reQuest

ARQ is based on the retransmission of missing data packets, which is triggered by explicit or implicit (based on timeouts) ACK/NACKs from the receiver(s). Figure 2.5 on the next page is an example of transmission with ARQ. As you notice, multiple retransmissions can be necessary to insure delivery of all data packets to a receiver, but on average the number of retransmissions for each packet is $\frac{1}{(1-p)}$, where p is the packet loss rate (PLR). For example, if (PLR) is equal to 10%:

- receiver asks for 100 packets the first time. Only 90 arrives at destination.
- receiver asks for these 10 packets that it is missing. Only 9 arrives at destination.

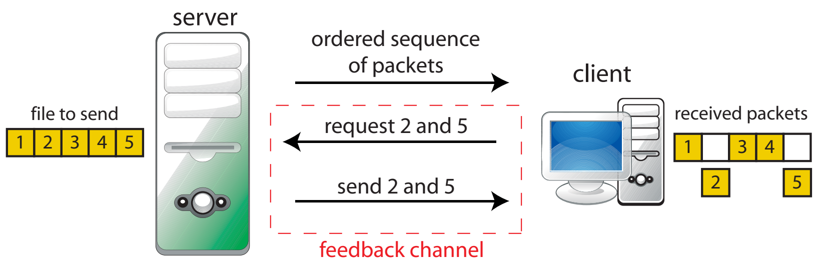


Figure 2.5: In ARQ based transmissions, the client sends requests for the missing packets. The server transmits the requested packets. This goes on until the client has got the whole file.

- receiver asks for the single packet that it is missing and it arrives at destination.

In this case there are 11 retransmissions, for a total of 3 request-answer rounds.

ARQ is generally used in unicast protocols, because it is very effective, simple to implement, and it does not require any processing of the data stream. The are a few drawbacks in ARQ-based protocols:

- the need for a feedback channel (whose use can be minimized though).

- the time (a full round-trip time (RTT) as a minimum) required to recover missing packets. This is an important limitation when propagation delays are very large (e.g. in the case of fast satellite links or deep space communications), or when there are real-time applications (e.g. full duplex voice/video conferencing and control system).

ARQ scales badly to multicast protocols and large groups, because the chance of uncorrelated losses grows with the size of the group, and that might require the retransmission of the majority of packets, even in presence of moderate PLR. This phenomenon is tolerable only in a few cases, e.g. when the group is small, or receivers have similar features and loss patterns. In all other cases ARQ performs poorly, since the aggregate PLR can become very close to 1. The transmitter needs to know precisely from receivers which blocks to retransmit. If the request does not arrive within a certain time (*timeout*), the server may deliver many times the same lost packet. This is known as *feedback implosion problem* [21]. Therefore, introducing redundancy in sent packets may be better, so the receiver can recover the errors on its own. This may lower the capacity of the channel, but it avoids delivering different amounts of missing data. FEC based protocols make this possible [31].

2.4 FEC: Forward Error Correction

FEC operates on a different principle [21] [22], namely it anticipates some amount of losses, and obviates by sending redundant data which allow the receiver to reconstruct up to a certain number of missing packets. The communication process thus includes an encoding phase at the sender, where redundant packets are constructed from the source data, and a decoding phase at the receiver, where source data are extracted, if possible, from the available packets (see figure 2.6 on the facing page). Because of the redundant data being transmitted, the PLR after decoding - i.e. the probability that some data packet cannot be reconstructed - can be made much lower than the raw PLR on the communication channel. FEC by itself does not guarantee reliable delivery (unless the number of redundant packets goes to infinity), but, by choosing a suitable encoding, the residual PLR at the receiver can be made arbitrarily small (obviously at a cost in the total transmission time), and those residual cases can be dealt with by using an ARQ-based protocol, if necessary at all. FEC can be computationally expensive, since the entire data stream must be processed by the encoder, so that each transmitted packet carries information on a (possibly large) number of source data packets. Decoding can be expensive as well, depending on the encoding technique being used and the actual amount of losses experienced. In reliable multicast protocols, though, the advantages of FEC may overcome the encoding/decoding overheads. In fact, to some extent, the presence of the decoding process decouples receivers from each other. Independent losses do not combine so badly anymore, because the same redundant data allows different receivers to reconstruct distinct missing packets. The effect of independent losses becomes less important also because of the presence of fewer data units (groups of K packets as opposed to single packets) and of a much lower PLR after decoding. These combined events allow FEC-based multicast protocols to scale to much larger groups than plain ARQ-based protocols.

2.5 Erasure codes

In network protocols, FEC can be implemented using erasure codes, which are a subset of error control codes, largely used in the telecommunication field. Basically, an (N, K) block erasure code takes K source packets and produces N encoded packets in such a way that any subset of K encoded packets (and their identity) allows the reconstruction of the source packets. Block codes are rigorously based on finite field arithmetic and abstract algebra. They can be used either to detect or to correct errors. They have the property to be without memory: there is no correlation between the symbols of a code-word and the next. A block code is defined as:

$$C(N, K)$$

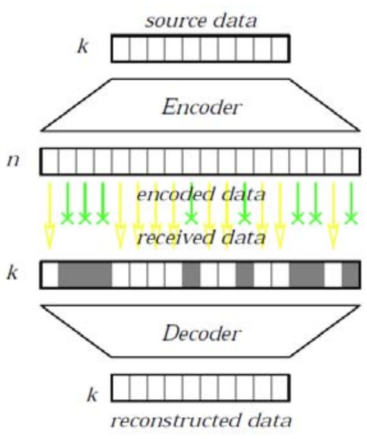


Figure 2.6: A graphical representation of the encoding/decoding process.

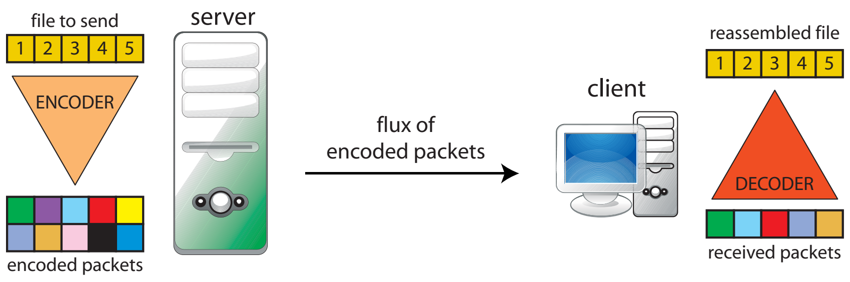


Figure 2.7: In FEC based transmissions the client receive a continue flux of packets, without reckoning if some of them got lost. The server continues to provide encoded packets for a certain amount of time.

where:

- C is the matrix of the code-words.
- N is the length of binary encoded data that are actually sent.
- K is the length of the pure binary information to transmit.

The most used are *linear block codes*: the code vectors are linear combinations of information bits. In this way the encoding and decoding complexity is significantly reduced. Linear block block codes can be represented in matrix form as:

$$\underline{y} = G\underline{x} \text{ (modulo-2)}$$

where $\underline{\text{x}}$ is the source data (a vector of size N), $\underline{\text{y}}$ is the encoded data (a vector of size N) and G is an $N \times K$ matrix called the *encoding matrix*. G must be structured in such a way that any subset of K of its rows has full rank, so that the original data can be reconstructed by using any K components from $\underline{y}$. The code words $\underline{y}$ are the set of vectors satisfying $\underline{y} \cdot H^T = 0$ mod 2 operation, where H is the parity check matrix.

In some cases, erasure codes are trivial to build. For generic N and K, erasure codes can be constructed and studied basing on the properties of linear algebra. For example, when the first $K \times K$ block of G forms an identity matrix, the code is called *systematic.* The encoded packets of a systematic code include the source packets in clear (systematic codes are much cheaper to decode when only a few erasures are expected).

$$G = \begin{bmatrix} 1 & 0 & \dots & 0 & p_{0,0} & \cdots & p_{0,N-K-1} \\ 0 & 1 & \dots & 0 & p_{1,0} & \cdots & p_{1,N-K-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & p_{K-1,0} & \cdots & p_{K-1,N-K-1} \end{bmatrix}$$

For example, if we want a $(K + 1, K)$ systematic code, G is the identity matrix followed by a row vector of all ones, and the redundant data is just the sum of the source data elements.

2.5.1 Tanner graphs

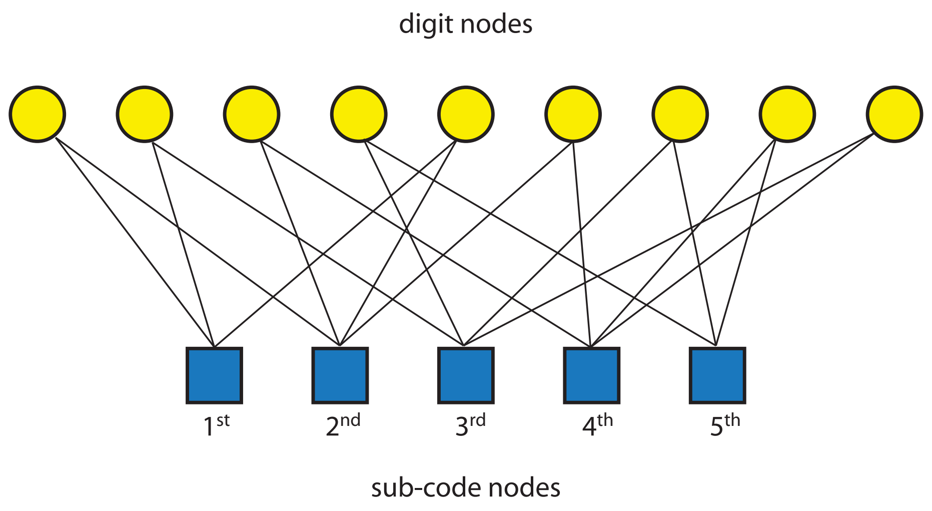
Robert M. Tanner [37] proposed another representation to create, by recursive techniques, larger error correcting codes from smaller ones. Tanner graphs are bipartite graphs, they are partitioned into sub-code nodes and digit nodes. These denote respectively the rows and the columns of the parity-check matrix H. If a nonzero entry exists in the intersection of the corresponding row and column, an edge connects a sub-code node to a digit node. Figure 2.8 on the next page is an example to explain the correlation between the matrix and the graph.

In mathematics the density of a graph can be calculated as:

$$D = \frac{\mid E \mid}{\mid V \mid}$$

where $\mid E \mid$ denotes the number of edges and $\mid V \mid$ is the number of vertices. This definition only works if $\mid V \mid > 0$; we define D to be 0 if the graph has no vertices. A graph with a low number of edges is defined as *sparse.* So a code that can be represented with a sparse Tanner graph is a sparse graph code. Dealing with linear codes, the terminology is slightly different.

- digit nodes become transmitted bits: *bit nodes.*
- sub-code nodes are the constraints to satisfy: *parity check nodes.*



$$H = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Figure 2.8: The matrix represents the connections between sub-code nodes (rows) and digit nodes (columns). Rows are exactly ordered as the sub-code nodes in the graph.

If the length of a code vector is N and the rate is $R = K/N$, the number of constraints is of order $M = N - K$. Any linear code can be described by a graph. In a sparse graph code each constraint involves a small number of variables only, thus the number of edges in the graph scales almost linearly with N.

2.6 LDPC erasure codes

Low-density parity-check (LDPC) [23] codes are *erasure codes* and they are used in *forward error correction*; LDPC are a subset of linear block codes. The name comes from a feature of their parity-check matrix: it contains only a few 1's in comparison to the amount of 0's. They are also known as Gallager codes in honour to Robert G. Gallager, the first to conceive them in early 1960's. The main advantage of LDPC codes is that they provide a transfer rate very close to the capacity of the channel. Moreover, they have linear time complex algorithms for decoding and they are suited for implementations that make heavy use of parallelism [16].

A regular Gallager code has a parity check matrix H, in which every row has the same number j of 1's and every column has the same number k of 1's. The connections in the graph are made randomly. We may calculate the code rate:

$$R = \frac{K}{N} = 1 - \frac{j}{k} \leq 1 \tag{2.1}$$

In an irregular Gallager code the matrix H is always sparse, but it has not the same number of 1's in its columns or rows. Digital Fountain codes are an example of irregular Gallager code.

They are the result of a long process of studying on LDPC codes by D. McKay, M. Luby, M. Mitzenmacher, A. Shokrollahi, J. Byers and D. Spielman. They developed and patented different types of LDPC codes [24] that are based on the scheme of Digital Fountain: Tornado codes, LT codes and Raptor codes. These patents belong to Digital Fountain Inc., founded by Luby after developing his Luby Transform codes [19]. Regular and irregular LDPC codes have been applied to Forward Error Correction (FEC) methods, defined by Internet Engineering Task Force (IETF). Nowadays two kinds of solutions exist to use FEC in transferring large data blocks: a free of patents solution proposed by INRIA [29] about a low density generator matrix (LDPC/LDGM) [30] and Raptor codes [33] patented by Digital Fountain Inc.

2.7 Digital Fountain Codes

When we think about a fountain of water, we image a spring where water comes out of a hole in the ground, falling around in form of drops. Now, if you are thirsty, you put a glass under the drops and when you have enough you can drink, irrespective of the particular drops that fill the glass (figure 2.9 on the facing page). Digital Fountain rely upon the same principle: a file is subdivided into parts, then encoded into packets which are delivered from a source to a destination; when the receiver has got enough symbols to rebuild the whole file, it stops the communication. In this idealized solution, each receiver can reconstruct an exact copy of the original file from the received encoding packets, independent of which servers generated the encoding packets, independent of losses, and independent of the intervals of time the receiver was joined to the sessions. No matter if some parts got lost, or in which order they are received, the task is accomplished either way. Ideally, the amount of processing required by the servers to generate encoding packets and by the receivers to reconstruct the file from received encoding packets is minimal.

More technically, Fountain codes are rateless because a potentially limitless sequence of encoding symbols can be generated from a given set of source symbols such that the original source symbols can be recovered from any subset of the encoding symbols of size equal to or only slightly larger than the number of source symbols. Regardless of the statistics of the erasure events

Figure 2.9: The principle of a Digital Fountain is the same as a fountain of water.

on the channel, we can send as many encoded packets as are needed in order for the decoder to recover the source data. Fountain codes are known for having efficient encoding and decoding algorithms and that allow the recovery of the original K source symbols from any K' of the encoding symbols with high probability, where K' is just slightly larger than K[1].

Different schemes and rules exist to encode and send data over the Internet using Digital Fountain [22] [20]. Figure 2.7 on page 41 illustrates a general model of Digital Fountain based transmission. The source generates encoded packets using the chunks of the object to send; then it delivers them to destination. The receiver decodes the received data in order to reassemble the source object. In this kind of communication *any encoded symbol is useful* and there are not duplicates; the receiver does not worry about the lost packets. It does not even reckon if any loss happens. Here, no more requests for missing parts are sent to the source and there is no need to wait till lost packets are resent. The receiver may use any packet that arrives, to reconstruct the whole object.

We need to know what the hurdles are implementing an FEC scheme like Digital Fountain in a communication system. For this reason, we will test the performances of various types of sending patterns, always based upon the XOR between packets. We have two parameters to evaluate the efficiency of an error correction code:

- overhead: $K' - K$, where K is the length of file in packets and K' is the total number of packets sent to reassemble it.

[1]Since $K' = K + H$, where H is the *overhead*, Digital Fountain codes are *near optimal codes*. We say that a Fountain code is *optimal* if the original K source symbols can be recovered from any K encoding symbols.

- computational complexity: the total number of XOR operations among packets.

2.7.1 Why use erasure coding?

The traditional scheme for transferring data across an erasure channel depends on continuous two-way communication:

- the sender encodes and sends a packet of information.
- the receiver attempts to decode the received packet. If it can be decoded, the receiver sends an acknowledgment back to the transmitter. Otherwise, the receiver asks the transmitter to send the packet again.
- this two-way process continues until all the packets in the message have been transferred successfully.

Certain networks, such as ones used for cellular wireless broadcasting, do not have a feedback channel. Applications on these networks still require reliability. Fountain codes in general, and LT codes in particular, get around this problem by adopting an essentially one-way communication protocol:

- the sender encodes and sends packet after packet of information.
- the receiver evaluates each packet as it is received. If there is an error, the erroneous packet is discarded. Otherwise the packet is saved as a piece of the message.
- eventually the receiver has enough valid packets to reconstruct the entire message. When the entire message has been received successfully the receiver signals that transmission is complete.

Globally, a Digital Fountain allows any number of heterogeneous clients to acquire bulk data with optimal efficiency at times of their choosing: in fact, receivers may join the stream at any time, then listen until they receive all distinct packets comprising the transmission. Moreover, no feedback channels are needed to ensure reliable delivery, even in the face of high loss rates.

During this thesis, we will focus on three Digital Fountain implementations:

- Random Fountain codes.
- LT codes.
- Raptor codes.

We are going to describe Random Digital Fountain and Raptor codes, since LT codes are included in Raptor.

2.7.2 Encoding and decoding

In the following paragraphs we describe Digital Fountain codes encoding and decoding methods. They are common to Random Digital Fountain and LT codes (and thus also to Raptor codes).

The encoding process

In a Fountain code [23] we want to encode a file of size K packets $s_1, s_2, \ldots, s_K$. At every iteration, labeled with n, the encoder generates K random bits $\{G_{kn}\}$. The transmitted packet t_n is the bitwise sum, modulo 2, of the source packets where G_{kn} is 1.

$$t_n = \sum_{k=1}^{K} s_k G_{kn} \tag{2.2}$$

This sum is the result of executing a succession of XOR's among packets. We can think of every set of K random bits as a new column that is added to the generator matrix G. This encoding operation defines a graph connecting encoded pieces to source pieces. If the mean degree is significantly smaller than K then the graph is sparse. To produce t_n we proceed in the following manner:

1. choose a number between 1 and K using a random number generator[2]. This is called *degree* and indicated as d_n. The degree is fixed and it is chosen following a particular distribution.
2. pick up d_n pieces of the file among K. This is achieved by using random numbers that are invariably taken from a uniform distribution on $(0, N]$, where N is the number of pieces into which the message has been divided. t_n is the exclusive-or operation between the selected source pieces.

The information about the selected pieces must be sent together with data, in order to help the encoder to recover the file (figure 2.10 on the following page); in one approach, each symbol is accompanied with an identifier which can be used as a seed to a pseudo-random number generator to generate this information, with the same procedure being followed by both sender and receiver.

The decoding process

Now the file must be reassembled without errors. Let us assume that the receiver knows the part of the G matrix associated to the packets it has. This may happen in two ways:

[2]The choice of a distribution for the random number generator is a key point: it will affect overhead and number of XOR operations, as we discuss in the following paragraphs.

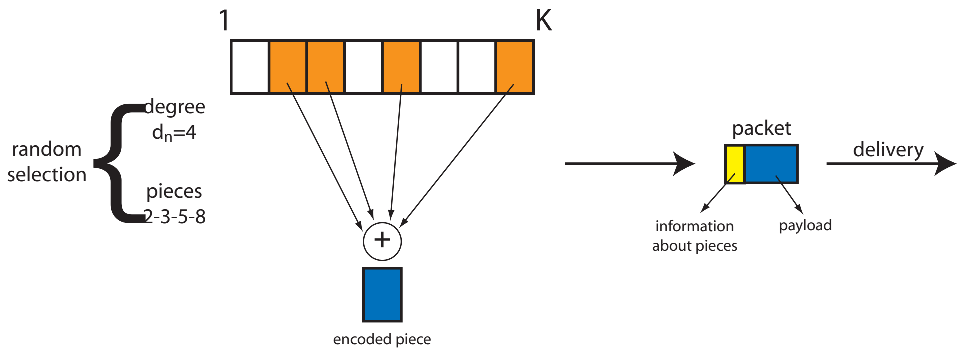


Figure 2.10: Example of packet generation in a Digital Fountain.

- the sender synchronizes with the receiver the same random numbers generator. So it knows what vectors of G it receives.
- transmit data together with a random key (usually of 32 bits) k_n. It is used to determine the K bits $\{G_{kn}\}_{k=1}^{K}$ by a pseudo-random process.

The received matrix, called G from now on, has a size of $K \times N$. To reassemble the file there must be $N = K$ packets at least. If the matrix $K \times K$ is invertible (modulo 2), G can be inverted and we obtain:

$$s_k = \sum_{n=1}^{N} t_n G_{kn}^{-1} \tag{2.3}$$

To begin the file decoding, we must create a graph using the information stored in the received packets: we associate the XOR'ed pieces (s_k) to source nodes and the symbols received (t_n) to check nodes. This is like a Tanner graph where there are K source nodes and N check nodes. The inversion process for a $K \times K$ matrix can be done in various ways:

- Message passing. Let's consider an example.
  - look for a check node t_n connected to only one source packet (the missing of such a situation causes the failure of this task).
  - set $s_k = t_n$ and add t_n to a list L.
  - add all the check nodes connected to s_k to the list L.
  - execute for all the t_n present in L the operation $t_n = t_n + s_k$ (XOR).
  - remove all the links to s_k.
  - repeat from the beginning until all the source packets has been determined.

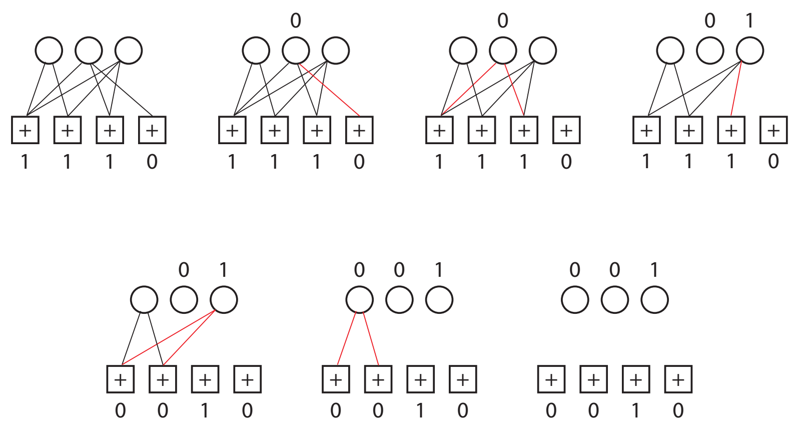


Figure 2.11: Message passing algorithm for $K = 3$ and $N = 4$. Notice the progressive decoding as soon as new source packets are discovered.

Figure 2.11 shows the decoding process in case each packet of the file is just 1 bit.

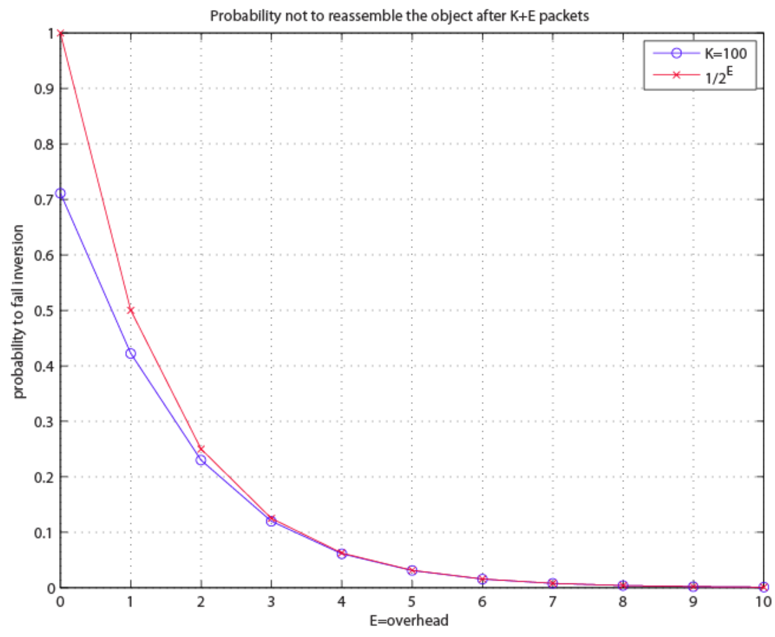
- Gaussian elimination, which has a complexity of $O(K^3)$. Let's consider the main idea: suppose A is a $n \times n$ matrix and you need to calculate its inverse. The $n \times n$ identity matrix is augmented to the right of A, forming a $n \times 2n$ matrix (the block matrix $B = [A, I]$). Through application of elementary row operations and the Gaussian elimination algorithm [45], the left block of B can be reduced to the identity matrix I, which leaves A^{-1} in the right block of B. If the algorithm is unable to reduce A to triangular form, then A is not invertible.

Even with no losses in the channel, it may happen to need a number of packets slightly greater than K. As the file pieces to add in the XOR operation are randomly chosen, the packets received may sometimes be redundant. So the decoding process requires some more data to get the file completely recovered. This exceeding number of information is called *overhead*. The result is equation (2.4) when $E = 0$ (overhead), while figure 2.12 on the following page shows the probability of not inversion $(1 - P_i)$ with E variable.

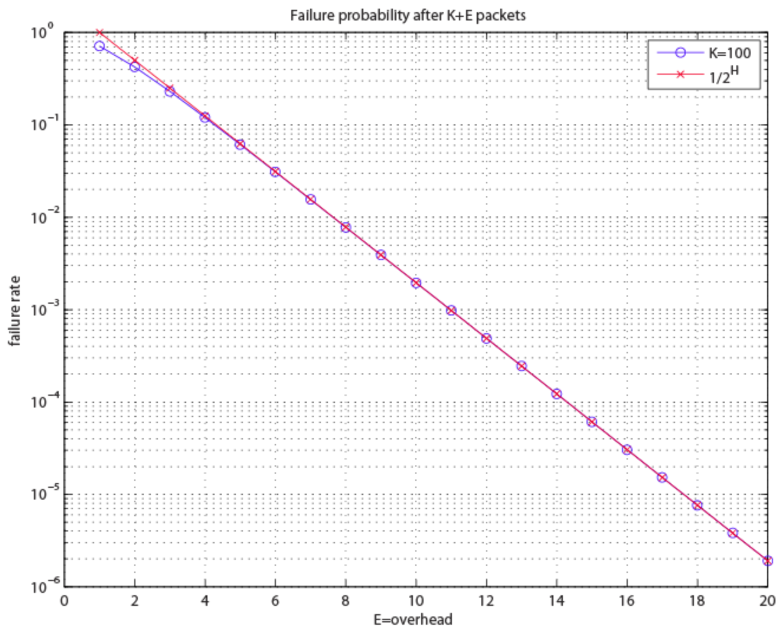
$$P_i = \prod_{j=0}^{K-(1+E)} 1 - 2^{-(K-j)} \tag{2.4}$$

2.7.3 Designing the degree distribution

Digital Fountain are based upon the XOR operation between randomly chosen pieces. In this way we cannot foresee what parts are picked up at any time,



(a) $1 - P_i$. After $K + 10$ packets we are about sure to recover the file correctly.



(b) semilogy$(1 - P_i)$. Here we can see better the failure rate.

Figure 2.12: Failure probabilities on Fountain codes.

the result is a redundancy in the sent data. This situation in what we want: since we cannot know the loss percentage for any channel at any time, we use random redundancy to supply the erasures. This is the principle of our FEC transmission. But a too high redundancy may raise the amount of XOR operations, while if it is too low we may wait for a long time to recover the file and the overhead goes high. We identify two scenarios according to the degree (d_n) selection. In these example, d_n identifies the amount of pieces (packets) the encoder picks up from the file to send. The parts are encoded together using XOR operation to form a single packet. So the degree represents also the number of 1's each vector has in the decoding matrix.

- defined random: $d_n = d_{\text{fixed}}$ with binomial variation. This will tune the random number generator to choose d_n around a certain percentage of K. If we have $K = 100$, selecting $d_n = 50$, the symbols generated would contain the XOR'ed pieces of about 50% of the object. The distribution of the ones in matrix varies like a binomial so the figure would be a narrow bell with a peak at 50%. Lowering d_n we get more single packet than before.

- defined function: d_n defined by a distribution function. This function drives the random generator to choose among a specified set of degrees, created by a rough mix of the features of both the ideal soliton and the robust soliton[3]. This scenario reduces the binary operations during the decoding process, as it contains a specified amount of low degrees.

To achieve a low number of XOR's we need to keep d_n low, but this is not enough to guarantee the success in reassembling the file.

2.7.4 Random Digital Fountain codes

In Random Digital Fountain codes, data bits to be XOR'ed are chosen randomly. The degree distribution is fixed and is defined random: every data bits is selected with 50% probability. If there are N data bits, the number of selected bits (degree) has a binomial distribution.

Targets and performances

Random Digital fountain codes improve the way to transmit data over a channel with erasures eliminating the requests for resending missing pieces. In the above description, we saw that they require XOR operations to have the file available and there is also the overhead. These are the two parameters to examine and if they both are not kept small the communication will not be efficient.

[3]Ideal soliton distribution and robust soliton distribution are two distribution functions used in LT codes [19].

Overhead As the decoder picks up parts from the file randomly, we cannot control what packets we have already transmitted: here the problem of redundancy arises. For this reason, we must wait until we receive the right data to recover the file completely; this could happen in few more packets, but nobody guarantees. Now, we suppose that the number of received packets is K', which is $K + H$, where H is the overhead. To estimate H, we introduce the probability of a failed decoding (F), due to an insufficient number of packets collected:

$$P(F) \leq 2^{-H} \tag{2.5}$$

Now, given K, it is very easy to estimate K' and thus the corresponding overhead $H = K' - K$. For example, if $K = 1000$ we estimate that K' must be at least 1020, with a corresponding overhead of 2%. Notice that overhead depends only on the probability of failed decoding desired and it is independent from K.

Complexity Another important feature is bound to the creation of a Digital Fountain code: to recover the whole file from the packets arrived we have to execute XOR operations. The computation is strongly related to the number of 1's present in the G matrix: the more the 1's are, the higher is the amount of calculations. As a consequence both the encoding and the decoding processes may influence this parameter. Here we understand the importance of using distributions that provides the creation of packets having low degrees. An ideal decoding algorithm should try to decode packets within a reasonable time and with as less number of arrivals as possible. But this would push high the number of XOR operations and the result would be slow motion or freezing frames due to a congestion of the hardware. To prevent this scenario we must find the right balance among overhead and computation, because any application could run at the same time and it may be greedy of resources like a video player.

The decoding matrix must have at least K rows in order to guarantee that the code is decodable. It has been demonstrated [23] that $\log(K)$ is the minimum degree that every row must have in order to achieve an overhead equal to 0 (notice that this is not guaranteed – it occurs with a defined probability). That being said, the minimum number of 1's in the decoding matrix must be greater than $K \log(K)$. Since the number of XOR depends on the number of 1's in every row, we could try to lessen the number of XOR's by keeping the matrix sparse.

However, in a pure Digital Fountain code, it is impossible to keep both the overhead and the XOR operations very low. Some codes tried to solve some of these problems reducing the efficiency of the random generation: they lower the degree, at the cost of a growth of the overhead. Examples are Tornado Codes and LT Codes [24].

2.7.5 LT Fountain codes

In LT Fountain codes, the number of selected bits (degree) is random but follows a precise law known as *soliton distribution.* Encoding is done in two steps:

1. the degree is chosen (robust soliton distribution). With solition, the degree is upper bounded to 40 (but this is very rare): in fact, most rows have only one or two 1's.

   There is only one parameter that can be used to optimize a straight LT code: the degree distribution function. Luby himself discussed [19] the ideal soliton distribution defined by:

$$\begin{aligned} P(d=1) &= \frac{1}{n} \\ P(d=k) &= \frac{1}{k(k-1)} \qquad (k=2,3,\ldots,n) \end{aligned} \tag{2.6}$$

   This degree distribution theoretically minimizes the expected number of redundant code words that will be sent before the decoding process can be completed. Unfortunately the ideal soliton distribution does not work well in practice and a modified distribution, the *robust soliton distribution* [13], is usually substituted for it. The effect of the modification is, generally, to produce more packets of degree 1 and fewer packets of degree greater than 1.

2. data bits are chosen (uniform distribution).

Decoding can be done with Gaussian elimination, however LT codes are designed to be decoded by message passing: at every iteration there is at least one degree one parity bit, and it is very improbable for the decoder to get stuck. It may happen that some data bit is never selected to generate parity bits: in this way the decoder is unable to decode a file with a reasonable overhead (2%) because it is still missing a fraction of 5%-10% of the file. In these cases the overhead of LT codes can be very high (100% in some situations). Raptor codes applies a LDPC pre-code in order to to recover those unselected data bits.

2.7.6 Raptor Fountain codes

Raptor codes are one of the first known classes of fountain codes with linear time encoding and decoding. They were invented by Amin Shokrollahi in 2001 and were first published in 2004 as an extended abstract.

Raptor codes encode a given message consisting of a number of symbols, K, into a potentially limitless sequence of encoding symbols such that knowledge of any K or more encoding symbols allows the message to be recovered with some non-zero probability. The probability that the message can be recovered increases with the number of symbols received above K becoming very close to 1, once the number of received symbols is only very slightly larger than K (there is a probability of 99.99% that the overhead is less than 2%). A symbol can be any size, from a single bit to hundreds or thousands of bytes.

Raptor codes may be systematic or non-systematic. In the systematic case, the symbols of the original message are included within the set of encoding symbols. An example of a systematic Raptor code is the code defined by the 3rd Generation Partnership Project for use in mobile cellular wireless broadcast and multicast and also used by DVB-H standards for IP datacast to handheld devices.

The encoding process

Raptor codes are formed by the concatenation of two codes:

1. a fixed rate LDPC code, usually with a fairly high rate, is applied as a pre-code (outer code). This pre-code may itself be a concatenation of multiple codes, for example in the code standardized by 3GPP a high density parity check code derived from the binary Gray sequence is concatenated with a simple regular low density parity check code. Another possibility would be a concatenation of a Hamming code with a low density parity check code.

2. the inner code takes the result of the pre-coding operation and generates a sequence of encoding symbols. The inner code is a form of LT code. Each encoding symbol is the XOR of a randomly chosen set of symbols from the pre-code output. The number of symbols which are XOR'ed together to form an output symbol is chosen randomly for each output symbol according to a specific probability distribution. As we already said, the distribution as well as the mechanism for generating random numbers for sampling this distribution and for choosing the symbols to be XOR'ed must be known to both sender and receiver.

In the case of systematic Raptor codes, the input to the pre-coding stage is obtained by first applying the inverse of the encoding operation that generates the first K output symbols to the source data. Thus, applying the normal encoding operation to the resulting symbols causes the original source symbols to be regenerated as the first K output symbols of the code. It is necessary to ensure that the random processes which generate the first K output symbols generate an operation which is invertible.

In the case of non-systematic Raptor codes, the source data to be encoded is used as the input to the pre-coding stage.

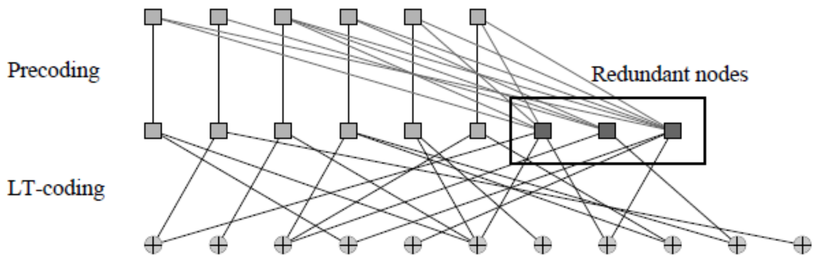


Figure 2.13: Systematic Raptor code: we apply a pre-code (outer code) and a LT encoding (inner code).

The decoding process

The relationships between symbols defined by both the inner and outer codes are considered as a single combined set of simultaneous equations which can be solved by the usual means as in Digital Fountain, typically by Gaussian elimination. Decoding succeeds if this operation recovers a sufficient number of symbols, such that the outer code can recover the remaining symbols using the decoding algorithm appropriate for that code.

Targets and performances

Raptor codes currently give the best approximation to a Digital Fountain. A virtually limitless supply of packets can be generated on the fly after some small initial preprocessing, with each packet taking only constant time to produce. Decoding can be accomplished after receiving just a few percent more than the minimum of K encoding packets (with high probability), and requires space and time linear in the size of the original message. Moreover, very efficient implementations are possible.

Overhead For any overhead $H \geq 0$ and sufficiently large K, the message M can be decoded after receiving only $K + H$ packets with high probability. Just notice that in case of systematic Raptor and if the erasure probability is equal to zero, the required number of packets to decode is exactly equal to K (first K output elements of the coding system coincide with the original K elements).

Complexity Raptor codes reduces significantly the complexity in comparison with Random Digital Fountain codes. Because of weakened LT code, the number of selected bits (degree) is bounded (e.g. max degree is 40 – although a packet with a degree 40 is very rare). As a result, Raptor require $O(1)$ time to generate an encoding symbol and $O(K)$ time to decode a message of length K.

Chapter 3

BitFountain

3.1 Data exchange and connections

In the Internet, the fluxes of data are regulated by mainly two protocols: TCP and UDP. Let's briefly describe them.

3.1.1 TCP (Transmission Control Protocol)

TCP is a connection-oriented protocol, which means that upon communication it requires handshaking to set up end-to-end connection. A connection can be made from client to server, and from then on any data can be sent along that connection. TCP has the following features:

- reliable - TCP manages message acknowledgment, retransmission and timeout. Many attempts to reliably deliver the message are made. If it gets lost along the way, the server will re-request the lost part. In TCP, there's either no missing data, or, in case of multiple timeouts, the connection is dropped.

- ordered - if two messages are sent along a connection, one after the other, the first message will reach the receiving application first. When data packets arrive in the wrong order, the TCP layer holds the later data until the earlier data can be rearranged and delivered to the application.

- heavyweight thus slow - TCP requires three packets just to set up a socket, before any actual data can be sent. It handles connections, reliability and congestion control. It is a large transport protocol designed on top of IP.

- limited by Windows OS. With the release of Service Pack 2 for Windows XP, Microsoft has introduced a feature that limits concurrent TCP connection attempts. Prior to SP2, there was no such limit. This feature is introduced in order to reduce the threat of computer worms spreading too fast without control. With the limit, only a maximum of 10 connection attempts per second are allowed. This may have affects applications, servers and P2P programs that attempt to open many outbound connections at the same time. However, there is no limit in other operating systems, like GNU/Linux or MacOSX.

- streaming - Data is read as a stream with nothing distinguishing where one packet ends and another begins. Packets may be split or merged into bigger or smaller data streams arbitrarily.

3.1.2 UDP (User Data Protocol)

UDP is a simpler message-based connectionless protocol. In connectionless protocols, there is no effort made to setup a dedicated end-to-end connection. Communication is achieved by transmitting information in one direction, from source to destination without checking to see if the destination is still there, or if it is prepared to receive the information. With UDP messages (packets) cross the network in independent units. UDP has the following features:

- unreliable - When a message is sent, it cannot be known if it will reach its destination; it could get lost along the way. There is no concept of acknowledgment, retransmission and timeout.
- not ordered - If two messages are sent to the same recipient, the order in which they arrive cannot be predicted
- lightweight thus fast - There is no ordering of messages, no tracking connections, etc. It is a small transport layer designed on top of IP. Avoiding the overhead of checking whether every packet actually arrived will lead to a lightweight implementation and thus makes UDP *faster* and *more efficient* for applications that do not need guaranteed delivery.
- datagrams - Packets are sent individually and are guaranteed to be whole if they arrive. Packets have definite bounds and no split or merge into data streams may exist.

UDP and NAT

NAT traversal through UDP hole punching is a method for establishing bidirectional UDP connections between Internet hosts in private networks using NAT. It does not work with all types of NATs as their behavior is not standardized. The same technique is sometimes extended to TCP connections, albeit with much less success. The basic idea is to have each host behind the NAT contact a third well-known server (usually a STUN server) in the public address space and then, once the NAT devices have established UDP state information, to switch to direct communication hoping that the NAT devices will keep the states despite the fact that packets are coming from a different host. UDP hole punching work with a Symmetric NAT (also known as bi-directional NAT, which tend to be found inside large corporate networks) if and only if we use *port prediction*[1]. With this technique, we predict the state of NAT when the endpoint tries to connect to us, and viceversa [42].

The technique is widely used in P2P software (although not in BitTorrent) and VoIP telephony. It is one of the methods used in Skype to bypass firewalls and NAT devices. It can also be used to establish VPNs (using, e.g., OpenVPN).

3.1.3 Packet splitting

In TCP and UDP, data is not transmitted as a unique block. If it was so and something went wrong, we would lose the whole information and the sender would have to retransmit all from the beginning. For this reason, the object to send is divided into pieces called *packets*. Before starting any explanation we must define the meaning of the word *packet*. For us this term may have two meanings according to the context. When we are talking about data transmitted through the Internet, packet is an object composed by various parts. One of them is the actual data, technically called payload. But, we could use the word packet to refer to the actual parts in which the file has been divided. Splitting files in packets goes to the advantage of communications over the Internet. If some packets got lost the receiver would ask only for a small amount of data. This saves both time and use of bandwidth. TCP can identify what are the missing packets through ARQ: a method used to detect errors

3.1.4 Digital Fountain blocks

Digital Fountain approach is based on a redundant encoding of data which makes the protocol tolerant to packet losses and client heterogeneity: since each received packet conveys information on a (potentially large) number of source data packets, reception of specific packets is not necessary; rather, it is the total number of received packets which determines the successful completion of the data transfer. This approach has large implications on the structure of the communication protocol: the need of feedback from the receivers is drastically reduced, with significant advantages on the complexity

[1]With Symmetric NAT, the IP address of the well known server is different from that of the endpoint, and therefore the NAT mapping the well known server sees is different from the mapping that the endpoint would use to send packets through to the client.

and scalability of the protocol and on its applications. Summarizing, using Digital Fountain blocks we can identify two main advantages:

- there is not need to retransmit of a specific block.
- when a block is transmitted it is useful for many peers rather than one single peer.

We are going to discuss them in the following paragraphs.

3.2 ARQ vs. FEC: a simple example

Suppose a scenario (see figure 3.1 on the next page) where we are interested in a file split in 3 pieces (x, y, z). The P2P network is composed by 6 peers, named $A - F$, and we are J. We will make these requests:

- we request block x to A and B.
- we request block y to C and D.
- we request block z to E and F.

Suppose that only 50% of the contacted peers will answer to our requests, therefore:

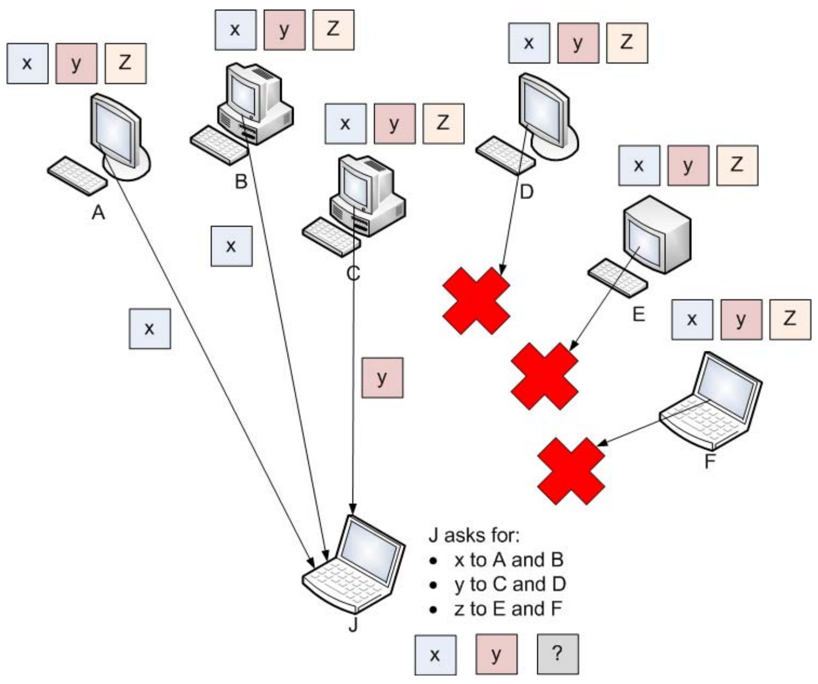
- A and B will answer, so we have 2 copies of the block x (the first copy is necessary, the second one is useless).
- C will answer, D does not. We have one copy of block y.
- E and F will not answer, we have no copies of block z. One more request to get z block.

This solution exposes one main drawback (typically of ARQ paradigm): we have an overhead due to request of a retransmission, while we have wasted bandwidth to get 2 copies of block x.

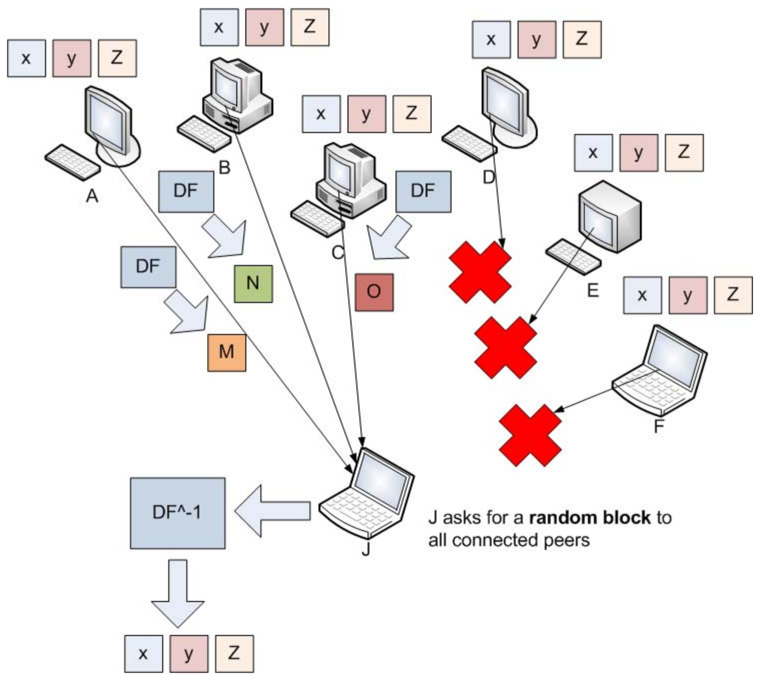
Now let us see how the encoded symbols are built in a Digital Fountain scheme. Each symbol must contain the result of the exclusive-or operation between the actual pieces of the object. Following a Random Digital Fountain scheme, P2P clients provides to deliver the XOR of the pieces that are picked up from the file randomly. Obviously, a client must decode the symbols received to regain the actual object. This means that with FEC some problems related to computation may arise. With an ARQ scheme this did not happen because the packets contained only source pieces of the file.

So, let's review the example applying a Digital Fountain FEC mechanism:

- A will answer with M, which is $x \oplus y$.



(a) Using ARQ. One more request is needed to retrieve z.



(b) Using FEC. We can decode the original file without additional requests.

Figure 3.1: A comparison between ARQ and FEC in a P2P file transfer. 50% of the peers will answer to the requests made by J.

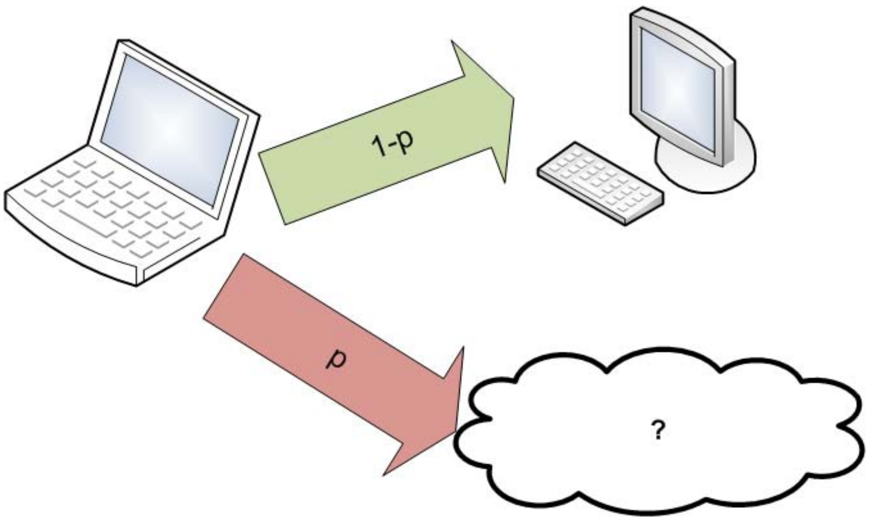


Figure 3.2: We are dealing with BEC channels when we transfer information on the Internet.

- B will answer with N, which is $x \oplus y \oplus z$.
- C will answer with O, which is $y \oplus z$.

Just note that also in this scenario only 50% of the peers have answered to our request, but we can decode with elementary operations the 3 pieces and thus decode the original file:

- $x = N \oplus O$.
- $y = M \oplus N \oplus O$.
- $z = M \oplus N$.

As we notice, there is no need of retransmission.

3.3 FEC: another example

Imagine that we have a transmitter and a receiver. The receiver wants to retrieve a file, divided in K blocks. Also, the receiver does not want (or can not) acknowledge the transmitter for every received packet. The channel is also noisy: the receiver can receive correctly only a fraction (with $1 - p$ probability, where p is the packet loss rate) of the transmitted packets (BEC, as we said in chapter 2). Figure 3.2 depicts the situation.

In one simple approach, transmitter keeps transmitting every packet with a Round Robin technique, which transmits every packet, in order, forever (see figure 3.3 on the next page). The receiver listen to channel until it gets all

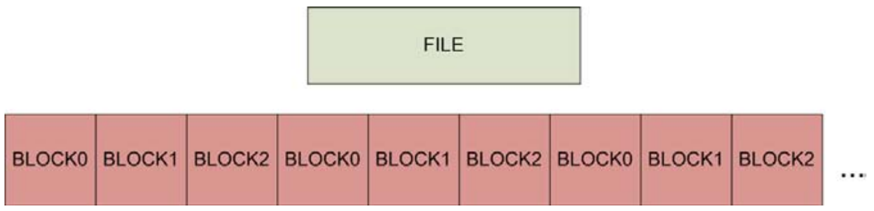


Figure 3.3: Round Robin transmission: it transmits every packet sequentially in a continuous loop.

the blocks of that file, and then disconnect. While listening, it counts every packet that it receives. How many packets does the receiver must collect to retrieve the original file?

We denote the total number of received block with X. We already talked about overhead in chapter 2, however let's emphasize its role in this situation. As we said, we call overhead the number of additional received packets (no matter if duplicated or not) that we have used to reassemble correctly the file. We introduce the overhead percentage and we denote it with E. We calculate the overhead percentage with:

$$E = 100\frac{X}{K} - 100$$

in this way:

- if $X = K$ there is not overhead because we have used the minimum number of blocks to reassemble the file.
- otherwise, as X can be only greater than K, we have an overhead and E expresses it in percentage.

Now, we want to estimate X when using a Round Robin technique. Let's make some considerations:

- for every given block the probability of not being received is p, whereas the probability of being received is equal to $1 - p$.
- in the first round we receive $K(1 - p)$ blocks (lost: Kp blocks).
- in the second round we receive $Kp(1 - p)$ of the previous lost blocks, whereas Kp^2 are lost again.
- in the third round we receive $Kp^2(1 - p)$ of the previous lost blocks, whereas Kp^3 are lost again.
- we repeat every round until $Kp \cdot p \cdot p \dots$ becomes less than 1.

Now, we estimate average number of rounds R in the following way:

$$\begin{aligned} Kp^R &< 1 \\ p^R &< \frac{1}{K} \\ R &< \log_p\left(\frac{1}{K}\right) \end{aligned} \tag{3.1}$$

or, in a more general form:

$$R < \frac{\log\left(\frac{1}{K}\right)}{\log(p)} \tag{3.2}$$

In the Round Robin case we transmit K packets for each round, thus $X = RK$, where R is the number of request-answer rounds and K is the number of total packets that constitute the original piece of information. That being said, Round Robin technique could lead to an high overhead, like $E = 500\%$ when $p = 99\%$. However, with erasure coding, we could achieve an overhead percentage of 2% in the same conditions. We will illustrate this result in figure 3.10 on page 87. Keep reading for further details.

3.4 Design

Looking for an average rate of erasures in the Internet, we would find a value around 5% [5], but there are also peaks at 12%. So we must accept the idea that the Internet is a very complicated system, variable in unpredictable ways. In one second the packet loss can change dramatically because of a lot of factors: cross-traffic, link quality, routing updates etc. There are lots of books dealing with this problem, so this aspect must not be undervalued. An erasure code code may pass over the problem of a continue request for missing packets. As an example of erasure code, Digital Fountain can reassemble the source objects only by received packets, without minding the losses. But the fewer packets arrive, the more the connection must stay alive in order to complete the data transfer.

The problem of losses during a transmission is independent from any method of erasure and P2P can reduce it. In this kind of network when a person requests a file, packets may come from different sources. Other people who have parts of the same object may help the download. Even if some sources are slow, the faster ones continue to send packets and the download does not stop. So there is a reduction of losses at cost of increasing network complexity to manage all the connections.

We design a new BitTorrent client, called **BitFountain** (**Bit**Torrent + Digital **Fountain**) which will be the first P2P to integrate the power of BitTorrent and the adaptive characteristics of Digital Fountain (this kind of codes do not have a fixed recovery capacity so they can adjust it using the loss probability estimated). Since we wanted an initial implementation which had to be compliant with BitTorrent protocol, we must integrate Digital Fountain scheme with little modifications to BitTorrent protocol. As a consequence, we started developing BitFountain client by modifying a BitTorrent client. The development will not be easy: we are going to deal with network programming, distributed computing, threading, and so on.

In the following pages, we are going to describe its characteristics in detail. Before explaining how we developed this client, we will illustrate the steps we took in designing this client.

3.4.1 Using UDP instead of TCP

BitFountain takes advantage over normal BitTorrent client in using UDP sockets:

- UDP sockets meets the characteristics of erasure channels (no ordering, losses, unreliable, ...).
- we can open more connections to other peers at the same time.
- we can ask every (unchoked) peer for a random block: since in Digital Fountain any block has the same importance as the others, there will be no duplicate blocks: *every received block is useful*[2].
- as a consequence of the previous statement, lost requests are unimportant and discarded. For example, if we need N blocks and we expect that only a fraction $\frac{1}{Z}$ of our block requests will be answered, we request for NZ random blocks to NZ peers. In this way we receive N random blocks and we reassemble the original information.
- technically speaking, we put the client in end game mode from the beginning, asking a random block to all unchoked peers.
- globally, we can increase the connectivity of the P2P network graph using less resources than normal BitTorrent.

3.4.2 Digital Fountain applied to BitTorrent data levels

Since there are two BitTorrent data levels, pieces and blocks (see figure 3.4 on the next page), we studied how we could integrate Digital Fountain at each level.

[2]In fact, a random block is useful if it is linear independent from other blocks already received. So, as the received number of blocks increases, the probability that a new received block is independent from the others decreases. However, the probability that a block is dependent from the already received blocks is negligible when the number of the input blocks K is elevated (because the total number of random blocks that can be generated is very high).

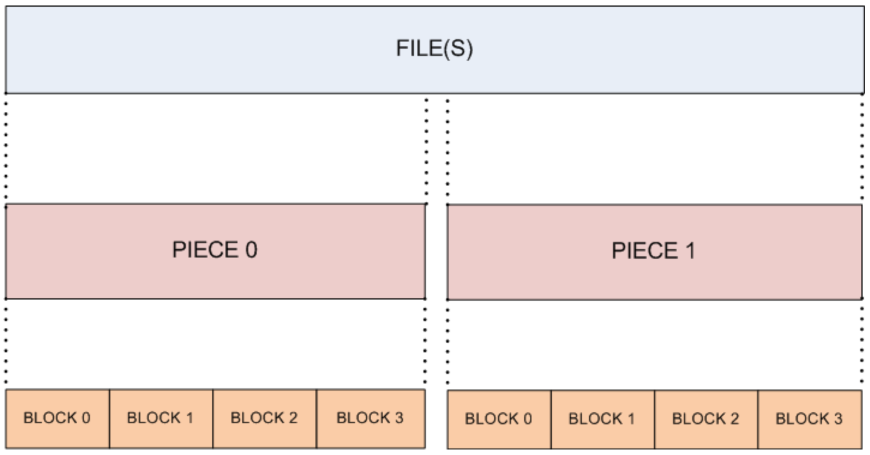


Figure 3.4: Data levels in BitTorrent. We could apply Digital Fountain to piece or block, or both. At the moment, we are not considering the tail piece.

DF applied to pieces

The main idea is to apply Digital Fountain encoding method to BitTorrent pieces: the input file is already divided in pieces (so Digital Fountain are not required to split it); for every request, we apply Digital Fountain encoding method using chosen pieces as input (Digital Fountain will randomly select which pieces to XOR). In this way we obtain a random piece, which has the same size as BitTorrent pieces. Then, the random piece is sliced accordingly to the `length` specified in the request, obtaining a random block, which is sent to the requester. See figure 3.5 on the facing page for further details.

During the design of this solution, we realized that a major problem will arise: this approach is applicable if we have all the pieces. Indeed, we can just wait to have only the pieces selected to generate the random piece. We have discarded this approach because the BitTorrent protocol prescribes that a piece must be made available to other peers as soon as it is completed: this will not occur with random pieces, because we are not able of advertising the completed receiving of a random piece; let's explain the reason. In order to exchange random peers, every peer constructs and sends different random pieces. As we said, a random piece could be created only if the peer has the required input pieces; otherwise, the peer can only retransmit the random piece that it has previously received – this is slightly useful for the other peers.

The "missing piece" problem A major problem for BitTorrent swarms is that when they get old and don't have a permanent seed, they tend to become non-recoverable. The problem is that if even a single piece winds up

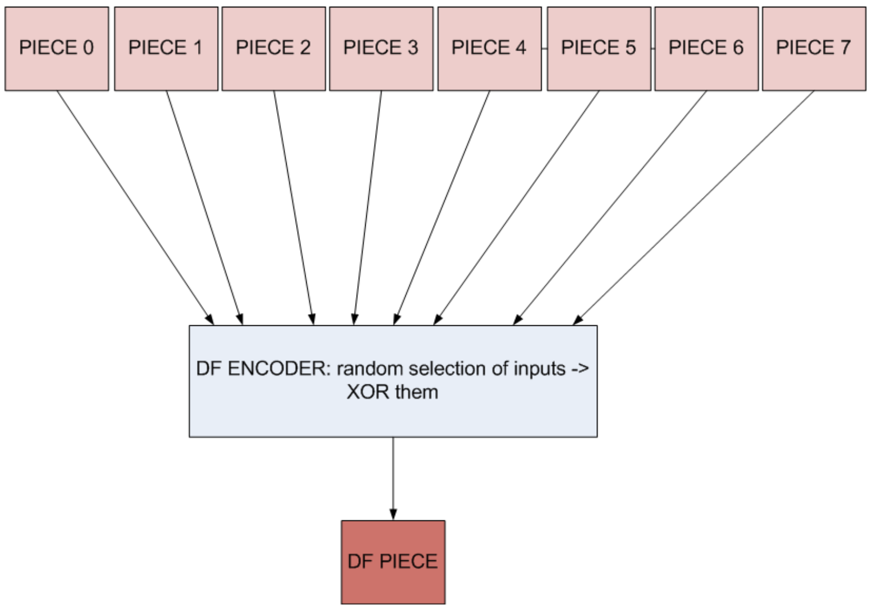


Figure 3.5: Digital Fountain applied to BitTorrent pieces.

not being on any current peers, then nobody can get a complete copy of the whole file. Rarest first does a fairly good job of making pieces be as evenly distributed as possible, and there's an obvious theoretical limit in that if the total amount of data all peers currently have is less than the complete size of the file then there's no way to do a complete recovery.

Let's consider the case of three peers downloading a file, and they're all halfway done when one of them disappears. The remaining two have enough bits between them, but is it possible for them to be guaranteed the ability to reconstruct the file? With this approach we are eliminating the "missing piece" problem: since all pieces have the same importance as the others, there won't be any critical piece which prevents the completion of the download. BitTorrent developers, as Bram Cohen, are investigating a similar solution to this problem using parity check XOR pieces [47].

DF applied to blocks

We have followed this approach during the development of BitFountain client. This approach requires a minimum change in BitTorrent client, making BitFountain inter operable with normal BitTorrent clients. We have followed this approach during the development of this thesis.

Encoding

The Digital Fountain encoding is applied to blocks within a piece; in other words, when a client receive a request for a block inside a piece (which the

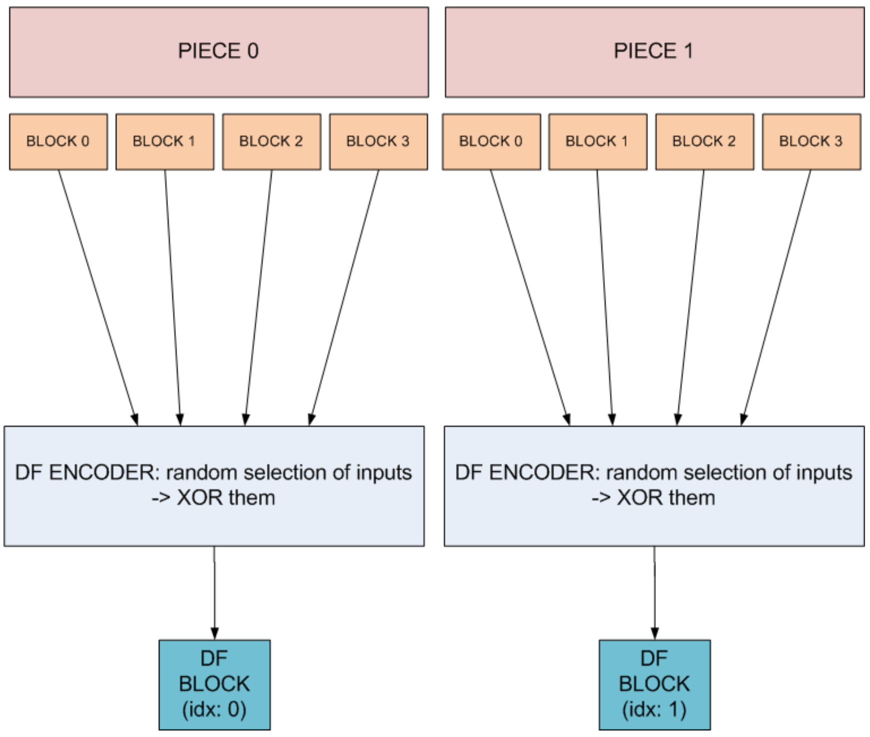


Figure 3.6: Digital Fountain applied to BitTorrent blocks.

client has for sure, this is a consequence of BitTorrent protocol), we select random blocks within that piece, and then XOR them, obtaining a random block. The encoding process is described in figure 3.6. The output block will be sent in answer to a peer. During the encoding, the quotient between `offset` and `length` will be used as a seed for the random generator number inside the Digital Fountain encoder[3]. The quotient will generate a sort of *sequence number*, as we will see in section 3.7 on page 88. In this way, encoder and decoder both know which source packet (blocks) are used to produce that kind of random block.

Decoding

A client must request other peers for a block belonging to a specified piece index, but it does not care about `length` and `offset`. In fact, a client must request random blocks until it can decode the original piece. It does not matter *which* block a client receives: it is only important that a client receives *enough* random blocks to decode. Any Fountain capable client must maintain a decoding area for every piece. When a random block comes in, the client

[3]`offset` and `length` are specified in the request coming from the remote peer, as we already said in chapter 1.

tries to decode the piece using random blocks contained in the decoding area for that piece:

- if the decoding is possible (we have collected enough random blocks), we reassemble a BitTorrent piece. The client must hash check it to prove its integrity.
- otherwise the client must enqueue a new request for a random block belonging to that particular piece.

Just notice that the client will always decode a piece: in case of corrupted random blocks, the piece will not pass the hash check (see chapter 4 for security issues). We will discuss further implementation details in section 3.7.

DF applied to pieces and block

This is a combination of the two previous approaches and leads to two different implementations:

- we generate a random piece by randomly XOR'ing all the pieces. Then, we generate a random block by randomly XOR'ing blocks within the same random piece.
- we create random blocks by randomly XOR'ing blocks from randomly chosen pieces. Then, we generate a random block by randomly XOR'ing random blocks generated on the previous step.

Because of the problem described in section 3.4.2 on page 66, we haven't focused on this approach.

3.4.3 A more efficient approach

If a client is Fountain capable and it is in end game mode during the whole download, it asks to every unchoked peer a block: *all* the peers which answer sending a random block will help the client to speed up the download of the file, leading to a more efficiently and rapidly download process. Now, we metaphorically compare this approach with normal BitTorrent approach: what is more likely to succeed?

- ask 100 $ to one person (it is likely you do not get anything).
- ask 1$ to 100 people (it is likely you do get something).

You are certainly sure that the second scenario is more likely to succeed. So, again, which is more likely to succeed?

- ask 100 blocks to one peer (it is likely you do not get anything). BitTorrent asks for all the block in a piece to specified peers. Weak peers slow down the download, as you have to wait for them.

- ask 1 random block to 100 peers (it is likely you do get something). BitFountain develops a light BitTorrent protocol (suitable for real time video streaming) that asks random blocks to several peers in parallel, an easy one-to-many approach (as we are using UDP). Every request is very cheap, other peers may quick reply and then close the connection. In the BitTorrent situation, this will leads to useless duplicates (hence the `CANCEL` message in the end game mode). Using Digital Fountain leads to a more efficient approach, because:
  - there is no need to request for a specific block.
  - every received block is a useful contribution (there are not duplicates).
  - Fountain capable clients can work in endgame mode, maximizing the probability of receiving random blocks. There is not the need to send `CANCEL` messages to other peers when a block is received (in fact, there are not duplicates). `CANCEL` messages should be sent only when a piece is reassembled successfully.
  - every client does not worry about lost requests or lost blocks.
  - it is just like asking for an upload speed sample to every peer: weak peers are unimportant, as long as fast peers send enough blocks.
  - we do not have to modify the BitTorrent protocol heavily (in case of Digital Fountain applied to BitTorrent blocks within the same piece).

In this way, Digital Fountain enables fast parallel P2P.

3.5 BitFountain: an initial prototype

3.5.1 In search for a BitTorrent client

As we already said, we must start by editing a BitTorrent client. Therefore, we have searched for a well-written, easy-maintenable BitTorrent client. We have taken in account several clients, inspecting the source code and decide pro and cons for every client analyzed.

CloudStorm library

A previous intern student at STMicroelectronics, Andrea Rota, developed a Digital Fountain library called CloudStorm [32], written in Java. Since we want to reuse its work, we initially searched for a Java-based BitTorrent client.

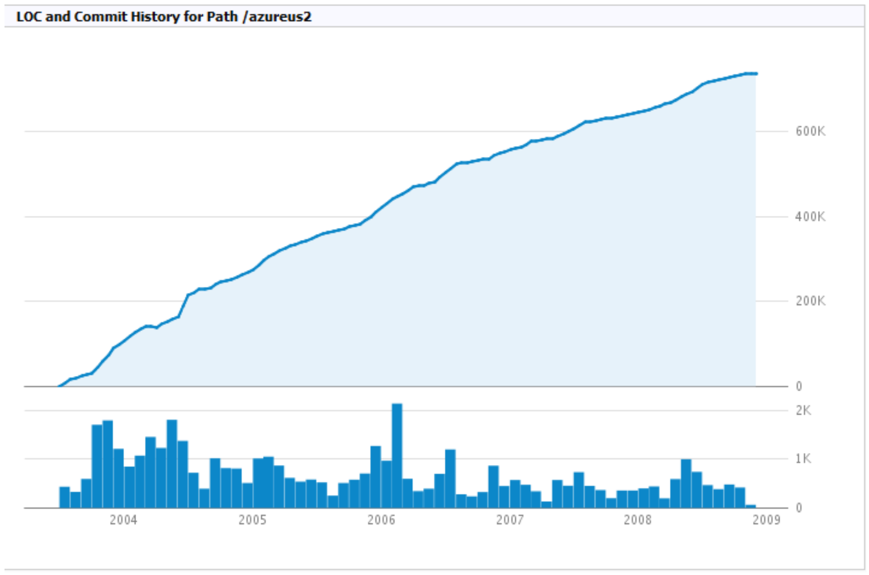


Figure 3.7: Azureus lines of code during time. Note that they have surpassed 600Klines of code.

Vuze (formerly Azureus) Vuze is a free and highly popular BitTorrent client written in Java. Its initial release was published in June 2003. In addition to BitTorrent-ting, Vuze allows users to view, publish and share original DVD and HD quality video content. Content is presented through channels and categories containing TV shows, music videos, movies, video games and others.

There is not an official documentation for Vuze core (the developers said that the code "is auto-explicative"). Therefore, we have studied its code for a couple of months. The code is really a mess: in addition of lack of documentation, the code is really disorganized: bad package splitting, multiple class for one task, user-implementations of synchronization schemes as semaphores, and so on. Vuze's lines of code are very, very numerous (600K, see figure 3.7). Developers keep adding features that we, frankly, don't require (such as HD videos); because of these problems, we abandoned the idea of modifying Azureus to suit our needs.

BitLet BitLet, abbreviation for BitTorrent Applet, is a BitTorrent program that enables the use of this file sharing protocol inside any Java-enabled web browser, without the need of an external dedicated client program. The software intends to make the use of BitTorrent very simple. To make a download, the user first needs to obtain a valid torrent metafile URL. The user then copies the URL to BitLet's main page and clicks the download link, initiating the downloading process.All the sharing is done by the user's computer, who is actively uploading and downloading as long as the program window is opened.

Bitlet is an interesting project: compliant to BitTorrent protocol, written in Java and very simple. Unfortunately, it's a closed source project. We have tried to contact its developer, an Italian programmer, asking him to inspect the source code for this thesis, but he refused to send us the source code. So, we dropped the idea of modifying BitLet.

HPBTC and jBittorrentAPI HPBTC[4] and jBittorrentAPI[5] are two promising open source BitTorrent clients. Because of their initial status, they are very simple, and thus they were a good candidates for our modifications. However, they are relatively old (their last release was in 2004) and they don't provide some core functions that we need (i.e. end game mode). At this moment, we had finished to inspect all Java-based BitTorrent clients: tired and hopeless, we switch to Mainline BitTorrent client.

Mainline BitTorrent is a peer-to-peer program developed by Bram Cohen and BitTorrent, Inc. used for uploading and downloading files via the BitTorrent protocol. BitTorrent was the first client written for the protocol.

BitTorrent is often nicknamed *Mainline* by developers denoting its official origins. Prior to version 6.0, BitTorrent was written in Python, and was free software. The source code for versions 4.x and 5.x are released under the BitTorrent Open Source License, a modified version of the Jabber Open Source License. Versions up to and including 3.4.2 were distributed under the MIT license. Since version 6.0, the BitTorrent client is a rebranded version of μTorrent. As a result, it is no longer open source, and this version of the program is currently only available for Windows (although, like μTorrent, the FAQ suggests to use it in Wine).

The client enables a range of features, including multiple parallel downloads. BitTorrent has several statistical, tabular and graphical views that allow a person to see what events are happening in the background. A host of views offer information on the number of peers and seeds which are present, from how much data is being downloaded and to how much data is being uploaded. It has an automatic recovery system which checks all data that has been been handled after an improper shutdown, such as a power failure. It also intermediates peering between itself, source trackers and other clients, thereby yielding great distribution efficiencies. The client also enables users to create and share torrent files.

[4]`http://hpbtc.sourceforge.net`

[5]`http://bitext.sourceforge.net`

3.5.2 Mainline client for developers

After a small overview, we decided to carry on the project using BitTorrent Mainline client[6]. It is certainly compliant with the BitTorrent protocol and it have all of the features that we need. We have chosen release 4.4.0 for our modifications. Again, there is not an official documentation, so we must inspect and study the code for ourselves, trying to understand how the code works. During the following paragraphs, we will give a description of the structure and workings of the BitTorrent client.

Network connection handling A BitTorrent client needs to initiate connections to the tracker and its peers. The communication between BitTorrent clients and tracker is through HTTP and the duration of connection is relatively short. When a BitTorrent client finishes downloading and before it is closed by an end-user, it serves as a seeder and uses upload rate to decide which peers to upload more instead.

Class `RawServer` (defined in `RawServer.py`) is one of the most important classes. It multiplexes IOs (for both server and client sockets) and is responsible for clearing timed-out sockets. It avoids using multiple threads or multiple processes and thus is much more efficient. Class `SingleSocket` is a simple wrapper class around the Python socket library and is also defined in `RawServer.py`. It is used to handle data in the buffer and send data through the socket. In BitTorrent client, other socket-related classes depend on the services provided by both `RawServer` and `SingleSocket` to focus on data transmissions and receptions without being bogged down to network connection handling details.

With the services provided by `RawServer`, several classes implement the BitTorrent protocol itself, handling peer-to-peer and peer-to-tracker information exchange and rate limiting. These classes include:

Connection Defined in `Connecter.py` and is used to handle BitTorrent protocol handshake between peers. Its member function `data_came_in` is called each time the socket receives data from other peers. Function `_read_message` is used to analyze each field in the received messages. A `Connection` object is created for each peer and tracker.

Rerequester Defined in `Rerequester.py` and is used to handle BitTorrent protocol handshake between peer and tracker, e.g., request for peer list, updating its own status and handling peer list.

[6]STMicroelectronics is involved in the development of P2P-Next (we have talked about that in chapter 1). Tribler is the main client of P2P-Next; Tribler is based on ABC, a Python client which is based on BitTornado, which is, in turn, based on Mainline BitTorrent. Because of this hierarchy, our work could be used in the development of Tribler, and thus in P2P-Next.

Upload Defined in `Uploader.py` and is responsible for uploading blocks to peers.

SingleDownload Defined in `Downloader.py` and is responsible for downloading from peers. A `SingleDownload` object is also created for each connecting peer. It maintains a list of all the active requests it sends out, keeps track of download rate from peers and checks if remote peer has any pieces local client is interested in. Its member functions respond to different messages received from peers and then act accordingly.

Choker Defined in `Choker.py` and is used to decide which peers to choke or unchoke. The tit-for-tat algorithm is implemented in this class.

Piece selection Though this is related to data transmissions, we list it separately because it is the part upon which we make our changes. There are several related classes here:

Bitfield Defined in `bitfield.py`. This is a helper class that provides easy access to bit strings that identify which pieces a peer already has and which pieces it needs to get.

PiecePicker Defined in `PiecePicker.py`. This implements the piece selection algorithms, e.g., the rarest first algorithm and the random first algorithm. It also keeps record of which blocks in a specific piece the client has and needs to request. In fact, we can modify the class such that a BitTorrent client only requests the pieces that we need, for example, the first half of the pieces for a file.

Storage handling The `Storage` class (defined in `Storage.py`) provides lower-level functions to open, read and write files and pieces. `StorageWrapper` class (defined in `StorageWrapper.py`) provides higher-level functions to read and write BitTorrent pieces. For example, it is possible to read or write a piece with specified index through `StorageWrapper`. Besides, `StorageWrapper` also ensures that pieces are assembled in order when it writes the file even though these pieces can be received out of order.

As we have said, Storage is a simple wrapper around Python's file functions for reading/writing at specified places. This is important because a file to download is divided into several pieces and each piece is further divided into slices as one unit of BitTorrent transmissions. These pieces/slices can be received out of order, so it is important to provide some higher level functions to read from or write to specific places. This is what Storage class provides: it contains a read and write function. With the read function, you can specify the position and the amount of data (in bytes) to read. With the write function, you can specify the position to write to with the data (string). Here a string can contain any characters and does not need to be `'\0'` terminated.

The `StorageWrapper` class may seem intimidating on first reading. It is. To initiate an object of this class, we have to define a lot of dummy functions to avoid some compilation and/or run-time errors.

`new_request(piece#)` function The `new_request` is called by other classes (`Downloader`) to request new piece given by index. Actually it returns the slice of the piece that is missing. For example, when we first call this function manually, it should return offset 0 and the length of a slice. When we call it the second time, it should return offset and length that both equal the length of a slice. Each time we call this function we get different results. The reason is that when we first call `new_request` for a piece, that piece is not requested yet. So the function will automatically populate a list which contains pairs of offset and length. The list is actually the specification for the list of slices for this piece. It is saved in `inactive_requests[index]` while `inactive_requests` is a list of lists. Then the function will remove and return the first element of list `inactive_requests[index]`. Later whenever it is called, it returns the slice with the minimum offset.

The `piece_came_in(piece#, offset, slicedata)` function Whenever we get a slice from one of our peers (offset and the actual data), then `piece_came_in` function will save the slice and do some book-keeping. Suppose one piece contains two slices and we just get one slice, then if we call `do_I_have(piece#)`, then it should return `False` because we haven't got all the slices yet. Once we get all the slices for a certain piece, then `do_I_have` should return `True`.

Downloading We describe the steps taken when downloading a file, and give a brief description of what each Python file does. All modules and classes mentioned are defined in the BitTorrent directory, or its BT1 subdirectory.

1. the user calls e.g. `btdownloadheadless.py` in the root directory to start the download.

2. the run method reads the config file and parsers command-line parameters.

3. creates a 20-byte BitTorrent peer ID by calling `createPeerID` for this download session.

4. creates a `RawServer` object which creates a `SocketHandler` object and schedules a task that checks for timeouts on the `SocketHandler`'s connections. A task is a method that the `RawServer` will execute once at a given time. Most tasks reschedule themselves, making themselves periodic. See figure 3.8 on the next page (a).

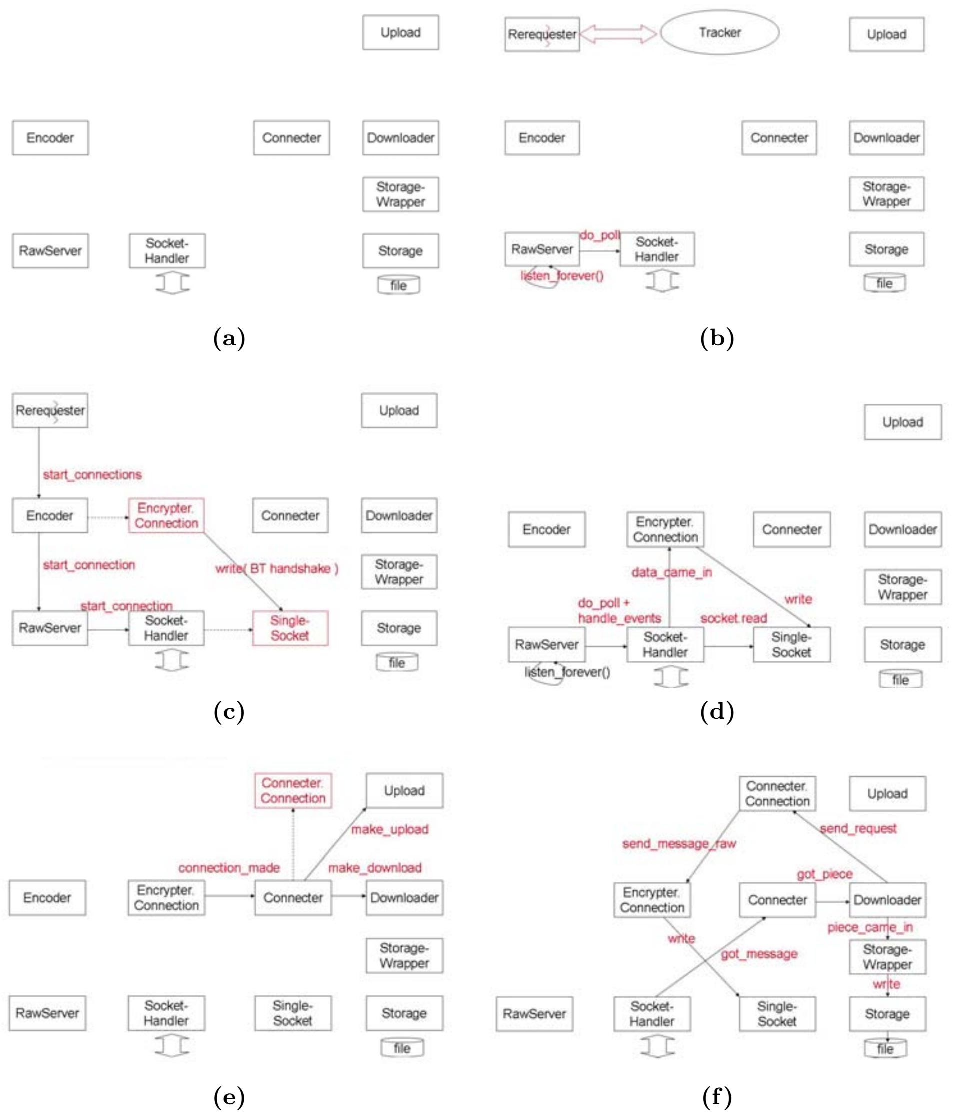


Figure 3.8: BitTorrent modules, classes and their interaction during downloading.

5. attempts talk to any firewall/NAT box via Universal Plug 'n Play (UPnP) to see if there is one.

6. calls `RawServer` to find and bind to a TCP listen port and to open this port on the firewall via UPnP.

7. reads the torrent from disk if already present or downloads it via HTTP, turning it into a metainfo Python dictionary, as described in the BitTorrent protocol spec.

8. creates a download `BTDownload1` object.

9. the constructor of the `BTDownload1` object creates a `PiecePicker` object. This latter object controls which piece to download next. It also creates a Choker object that will execute the optimistic unchoking policy. To this extent it schedules a task with the `RawServer`.

10. the client then calls `BTDownload1.initFiles()`.

11. `initFiles` creates a Storage object, which maps all bytes in the files in a torrent into a single linear address space. All indices used in the BitTorrent protocol refer to this address space. So a piece of data identified by an index is mapped to a particular offset in a particular file by this object. Furthermore, it creates an empty file for each file in the torrent, and handles file locking.

12. `initFiles` creates a `StorageWrapper` object. Its main task is to execute the disk-allocation policy. Multiple policies are supported:

    sparse uses sparse files; (i.e. the pieces are written on their correct place in a file, but only the actual data written is allocated on the file system).

    pre-allocate fills any gaps in the file with zeros.

    normal (the default policy) which writes data consecutively into the files and moves the data in the right location in the background.

    In addition, the `StorageWrapper` schedules a task with `RawServer` that checks the integrity of any data already on disk in case of a restart. It also schedules a task that automatically synchronizes the data to disk.

13. for a multi-file torrent, if enabled on the command-line, `initFiles` creates a `FileSelector` object that enforces user-specified download priorities on the files.

14. the integrity check scheduled by the `StorageWrapper` object is run.

15. the client calls `BTDownload1.startEngine()`.

16. `startEngine()` initializes the `PiecePicker` with information about the pieces already on disk, if any. It creates Measure objects for uploads and downloads. It creates a `RateLimiter` object. It also creates a `Downloader` object that keeps track of the download side of the BitTorrent download (e.g. which pieces have been received, which pieces are available from peers, etc.). At the end, it creates a similar object for the upload side: `Upload`. It creates a `Connecter` object. It creates an `Encoder` object which schedules a task with `RawServer` to send keep alives on all connections.

17. the client creates a `Rerequester` object which is the tracker client and which uses a separate thread. See figure 3.8 on page 76 (b).

18. when the `Rerequester` has obtained some peer addresses from the tracker it calls `Encoder.start connections()`. See figure 3.8 on page 76 (c).

19. the Encoder calls `Rawserver.start_connection()` and creates a `Connection` object for the created connection.

20. the `Rawserver` calls `SocketHandler.start_connection()`.

21. the `SocketHandler` creates a Python socket and connects to the peer. It then registers the socket with a poll object from Python's select module, and creates a `SingleSocket` object for the connection.

22. the client finally calls `RawServer.listen_forever()`, the object's mainloop.

23. the mainloop calls `SocketHandler.do_poll()` which calls Python select's `poll()`.

24. when data comes in on a connection, `SocketHandler.handle_events()` method is called. This method reads the data from the socket and reports it to the `Encrypter. Connection` for the connection via the `data_came_in()` method. See figure 3.8 on page 76 (d).

25. the `Encrypter.Connection` calls the next function pointer to handle the data. In the handshake phase of a BitTorrent connection this will call methods in `Encrypter.Connection` to handle the handshake. If the handshake is successful, the object calls the `Connecter`'s `connection_made()` method. All subsequently received messages are delivered to the `Connecter` via `got_message()`.

26. `Connecter.connection_made()` method creates its own `Connection` object which will handle all non-handshake BitTorrent messages. It also registers the connection with the `Downloader`, the `Upload` and

the `Choker`. (Note that there are 2 classes called `Connection`, in the `Encrypter.py` module and in the `Connecter.py` module.). See figure 3.8 on page 76 (e).

27. when messages come in, the `Connecter` will call the appropriate message handlers in these latter three objects and the `Connecter.Connection`. For example, when a `PIECE` message comes in, it calls the `Downloader`, which, in turn, calls the `StorageWrapper`. See figure 3.8 on page 76 (f).
28. in order to send a message, `Connecter.Connection` calls `Encrypter.-Connection`, which, in turn, calls the `SingleSocket` object, which, in turn, writes the message to the Python socket.
29. when a new connection arrives on a listening socket, the `SocketHandler` calls the `Encoder` which creates a `Connection` object for it, as with outgoing connections.

BitTorrent class organization overview The root directory contains the main Python scripts for:

- creating a torrent, via terminal (using `btmakemetafile.py`) or GUI (using `btmaketorrentgui.py`).
- starting the tracker: `bttrack.py`.
- seeding and downloading a file, via a terminal: `btdownloadheadless.py` or a GUI: `btdownloadgui.py`.

In the BitTorrent directory there is the source code, we give a little description of the most important files:

`CurrentRateMeasure.py` Defines class for measuring data rates, used in `btdownload_bt1.py` as argument to `BT1.Connecter` and `BT1.Upload`, and by `BT1.Rerequester` for progress reports to the tracker.

`HTTPHandler.py` A HTTP server skeleton, used by the tracker `track.py`.

`RateLimiter` Defines class offering some rate limit calculations, used in various classes, see `download_bt1.py`.

`RateMeasure.py` Defines a rate measure class, which is used by `BT1.Down-loaderFeedback` class.

`RawServer.py` This module controls `SocketHandler` (the network I/O handler), and executes tasks (i.e., a method executed at a particular point in time) on behalf of the BitTorrent client.

`SocketHandler.py` Main network I/O handler and dispatcher, handles the Python sockets.

`init.py` Initialization code for the client, contains code for creating a peer ID.

`bencode.py` Encoding/decoding functions for BitTorrent's on-the-wire format.

`bitfield.py` Defines `BitField` class used in BitTorrent protocol to report the available pieces to a peer at connection time.

`download_bt1.py` Main module for downloads via BitTorrent, defines `BT1-Download` class.

`launchmanycore.py` Main module of parallel downloader, used by `btlaunch-many.py`.

`natpunch.py` Module to open ports on NAT firewalls via UPnP.

`parsedir.py` Finds unique torrents in a directory tree, used by parallel downloader `launchmanycore.py` and the tracker `track.py` (to see for which torrents the tracker is allowed to track peers).

`piecebuffer.py` Defines `PieceBuffer` class used by `BT1.Storage` for reading from files.

`Choker.py` Defines `Choker` class.

`Connecter.py` Handles incoming non-handshake messages via `Connecter` and `Connection` classes.

`Downloader.py` Keeps track of download-side of a BitTorrent download.

`DownloaderFeedback.py` Keeps track of statistics and peer info (i.e., the so-called "spew" data) used by higher layers to display information about current peers.

`Encrypter.py` Defines the other `Connection` class that handles BitTorrent handshake messages. Defines the `Encoder` class that does connection setup of outgoing and handling of incoming connections and sends BitTorrent keepalives on existing connections. See `Connecter.py`.

`FileSelector.py` Allows for assigning priorities to specific files in a multi-file torrent download.

`PiecePicker.py` Defines `PiecePicker` class that selects the next piece to download.

`Rerequester.py` Defines `Rerequester` class, the tracker client.

`Statistics.py` Defines `Statistics` class that allows all statistics kept in the BitTorrent client to be accessed in one place.

`Storage.py` Defines `Storage` class.

`StorageWrapper.py` Defines `StorageWrapper` class.

`Uploader.py` Defines `Upload` class.

`track.py` Main tracker module.

3.6 Thunderstorm: a fast Digital Fountain library written in Python

Now that we have chosen a BitTorrent client, we must write a Digital Fountain library to be called inside BitTorrent code.

We initially made an effort to integrate CloudStorm with Mainline client, using a *bridge* like JPype[7], which allows Python programs full access to Java class libraries. This is achieved not through re-implementing Python, as Jython/JPython has done, but rather through interfacing at the native level in both virtual machines[8]. With this approach we experience a slowness in encoding and decoding, perhaps because of bridging. Again, we have no choice.

We start over and developed a new Digital Fountain library (named *ThunderStorm* because of its speed) written in Python, which offers support for Random Digital Fountain (like CloudStorm) and adds support for Raptor Digital Fountain. As we will notice, ThunderStorm achieves a great speed in encoding and decoding, keeping the implementation very simple and straightforward. To obtain such speed, we use some libraries to speed up the encoding process, like *NumPy* (it is the fundamental package needed for scientific computing with Python and can also be used as an efficient multi-dimensional container of generic data) and *Psyco* (it is a Python extension module which can massively speed up the execution of any Python code). ThunderStorm is also heavily tested: in the following graph you will see that we have tested it for at least 500×3 iterations. ThunderStorm is built with evolution in mind: it easy for developers to add a new type of encoding and decoding scheme. Due to all these motivations, ThunderStorm is faster, much more reliable and extensible than CloudStorm, and it is the only Digital Fountain library which includes the cutting edge technology of Fountain codes, especially Raptor codes.

Thunderstorm offers two way to generate random blocks (recall: chapter 2):

- using Random Digital Fountain. In this case, the number of input blocks which participate in XOR are 50% of the input blocks.

[7] `http://jpype.sourceforge.net`

[8] We can't use Jython because of C-modules used by BitTorrent, like twisted.

- using Raptor Digital Fountain. In this case, the maximum number of input blocks which are XOR'ed together is 40.

As we said, the input blocks to be XOR'ed together is random; the choice of which data block to XOR is made by numbers generated by the random number generator. To synchronize the random number generator for the transmitter and the receiver, every block is paired with a seed for the random number generator. In this way, the seed tunes the random number generator to produce the same bitlist on both sides. As a consequence, we can simply transfer the block with its seed instead of transferring the block with its bitlist (this will lead to a save of bandwidth). Now, we said that Fountain codes must XOR randomly chosen blocks:

```
def create_random_block(bitlist,piece):
    assert len(piece) % len(bitlist) == 0
    blocksize = len(piece)/len(bitlist)
    random_block = ma.zeros(blocksize,numpy.uint8)
    for i in xrange(len(bitlist)):
        if bitlist[i] == 1:
            block = ma.fromstring(
            piece[i*blocksize:i*blocksize+blocksize]
            ,numpy.uint8)
            random_block ^= block
    return random_block
```

As you can see, this method must receive as input a bitlist: in this way, we can provide either a LT bitlist or a random (50% of ones in bitlist) bitlist. As a consequence, we can reuse this snippet of code when using, respectively, Raptor or Random Fountain Codes. Regarding decoding, ThunderStorm implements an equation system solver using Gaussian elimination to solve equation systems for both Random Digital Fountain and Raptor Digital Fountain. The implementation of the equation system solver is obviously reused for both of the two cases.

3.6.1 Random codes in ThunderStorm

Now, let's see a simple encoding and decoding example:

```
for m in xrange(ITERATIONS):
    piece = os.urandom(piecesize)
    transmitted = []
    decoded = False
    txsn = 0
    used = 0
    Y = RandomDF(K,BLOCKSIZE)

    while decoded == False:
        txsn += 1
        ###########################################################
        # avoid encoding/decoding if packet is lost ########
        if random.randint(0,99) < PLOSS-1:
                        txsn += 1
            continue
        ###########################################################

        seed = random.getrandbits(32)
        rblock = create_random_block(
                rand50list(seed, K)
                ,piece).tostring()
        transmitted.append((seed,rblock))
        txsn += 1
        ###########################################################
        # packet is being transmitted on the channel ...   #
        ###########################################################
        used += 1
        rxsn,block = transmitted.pop()

        if Y.use_random_block(rxsn, block) == True:
            piecedec = Y.retrievedata()
            if sha.new(piece).hexdigest()
             != sha.new(piecedec).hexdigest():
                raise Exception
            decoded = True
            print 'Random DF
             encoding/decoding process finished'
```

In this little example, we generate a transmitter and a receiver. The transmitter generates a random piece of data that it want to transmit to the receiver. In order to accomplish its need, the transmitter uses Random Digital Fountain. As a consequence, it must generate a seed and a random block (which is a Random Digital Fountain block obtained by encoding the original piece with the specified bitlist, obtained by inspecting the seed). As we saw from the previous snippet, the encoder takes the seed (bitlist) and then XOR only the subpieces (blocks) which have the corresponding bit raised in the bitlist. After the generation of one block, the transmitter puts the seed and the random block in the channel. We have also modeled a channel that can randomly loose some blocks with a specified loss percentage: if the block is lost, we avoid encoding and decoding times. If the block is received, the receiver tries to reassemble the original piece. If the received blocks are not enough, the transmission/receiving routine will start again: in this way, the receiver keeps accumulating random blocks. At one point, the receiver can decode and reassemble the original piece: after that, we check the integrity of the original piece by matching the two hashes. The transfer is now finished and transmitter and receiver can now disconnect.

3.6.2 Raptor codes in ThunderStorm

The equivalent Raptor encoding and decoding example is pretty straightforward:

```
for m in xrange(ITERATIONS):
    piece = os.urandom(piecesize)
    transmitted = []
    decoded = False
    txsn = 0
    used = 0
    X = Raptor(K,BLOCKSIZE)
    Y = Raptor(K,BLOCKSIZE)

    while decoded == False:
        ###########################################################
        # avoid encoding/decoding if packet is lost ########
        if random.randint(0,99) < PLOSS-1:
            txsn += 1
            continue
        ###########################################################
        rblock = X.encode(
                txsn,
                piece).tostring()
        transmitted.append((txsn,rblock))
        txsn += 1
        ###########################################################
        # packet is being transmitted on the channel ...     #
        ###########################################################
        used += 1
        rxsn,block = transmitted.pop()

        if Y.decode(rxsn, block) == True:
            piecedec = Y.retrievedata()
            if sha.new(piece).hexdigest()
            != sha.new(piecedec).hexdigest():
                raise Exception
            decoded = True
            print 'Raptor DF
             encoding/decoding process finished'
```

3.6.3 Round Robin example implementation

For the sake of completeness, we want to show the snippet of the code to handle Round Robin transmission:

```
for piecesize in PIECESIZE:
    piece = os.urandom(piecesize)
    blocks = []
    for i in xrange(0, len(piece), BLOCKSIZE):
        blocks.append(piece[i:i+BLOCKSIZE])
    mean = []
    for m in xrange(ITERATIONS):
        received = False
        transmitted = []
        receivedC = {}
        i = 0
        txsn = 0
        used = 0

        while not received:
            transmitted.append(blocks[i])
            txsn += 1
            if random.randint(0,99) < PLOSS-1:
                transmitted.pop()
            else:
                used += 1
                receivedC[i] = transmitted.pop()
                if len(receivedC) == len(blocks):
                    received = True
                    print 'Round Robin
                    transmission/receiving
                    process finished'
```

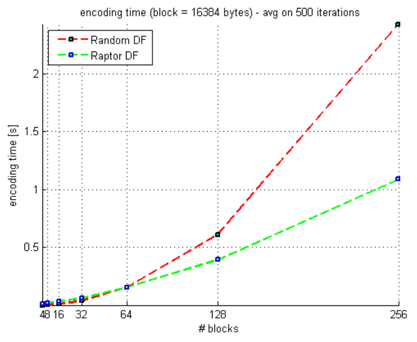
As we already said, in this approach we keep sending the blocks within the piece in continuous sequence on the channel: because of losses, the blocks may not arrive at destination. In this way, the receiver keeps accumulating "normal" blocks, which may be duplicates and thus useless (suppose a scenario where a receiver misses only the last block of a piece and for a series of coincidences it does not receive it because of losses, but it keeps receiving the first block: this block of information is duplicate, and thus useless).

We will illustrate the performance of the previous snippets in the following paragraphs.

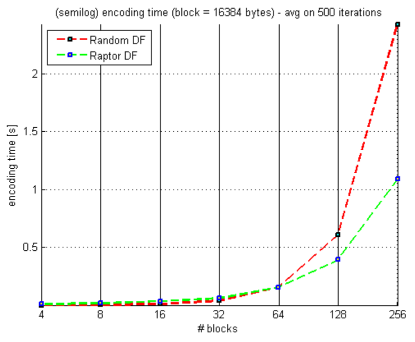
3.6.4 Benchmarking

In the following pages, you can see encoding and decoding time, for both Random Digital Fountain and Raptor Digital Fountain. We have benchmarked them with block sizes and piece sizes equal to BitTorrent's sizes to achieve a high likelihood.

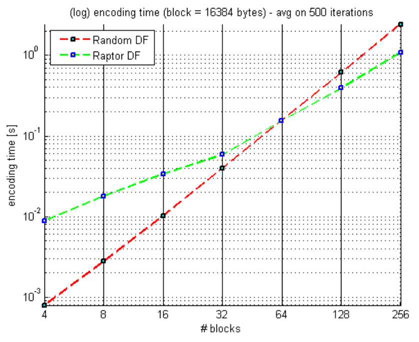
As you can see from figure 3.9 on the next page and figure 3.10 on page 87, Raptor Digital Fountain is much more faster during encoding and decoding than Random Digital Fountain. We have illustrated the reason in chapter 2 (recall: less blocks to XOR.). However, when the blocks are relatively few (when the number of blocks is less than 64), Random Digital Fountain take less time than Raptor Digital Fountain: we could think that using Random



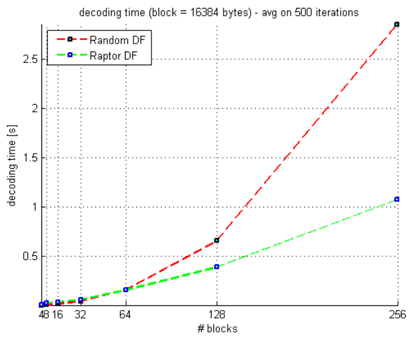
(a) Encoding time (notice the exponential growth of Random).



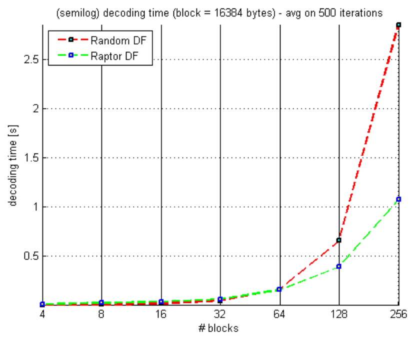
(b) Encoding time (semilogy).



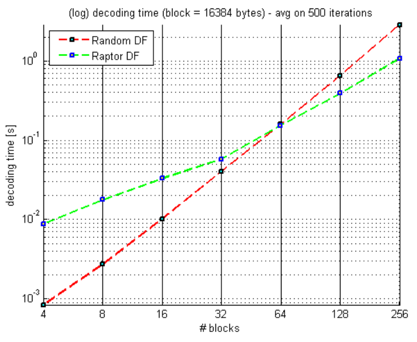
(c) Encoding time (log) (notable performances for Random with $K < 64$ and Raptor with $K \geq 64$).



(d) Decoding time.

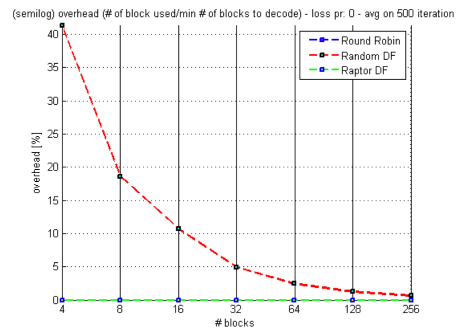


(e) Decoding time (semilogy).

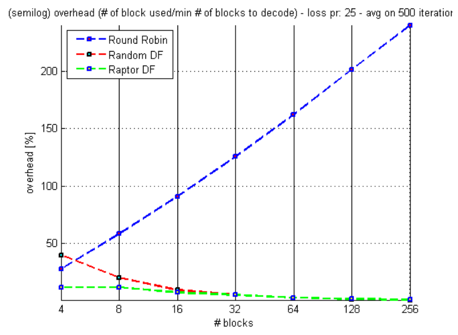


(f) Decoding time (log).

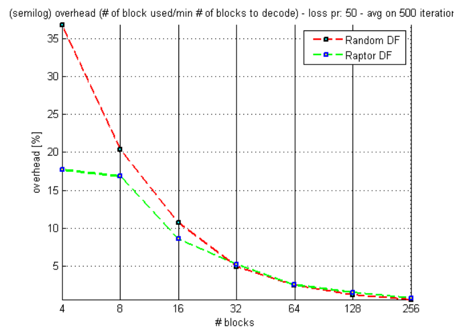
Figure 3.9: ThunderStorm benchmarking: Random & Raptor encoding and decoding time. Lower time is better.



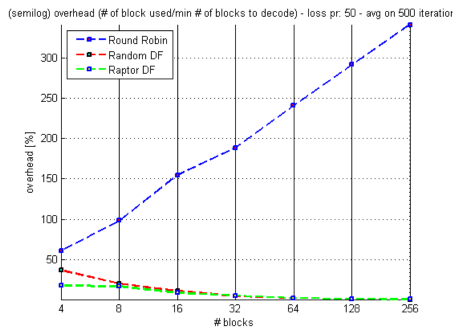
(a) Loss probability $p = 0$. Raptor and Round Robin have overhead equal to 0.



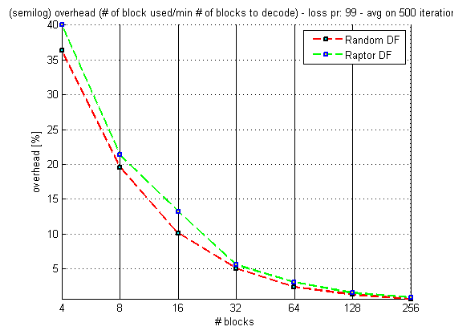
(b) Loss probability $p = 25\%$. Notice the growing overhead of Round Robin method.



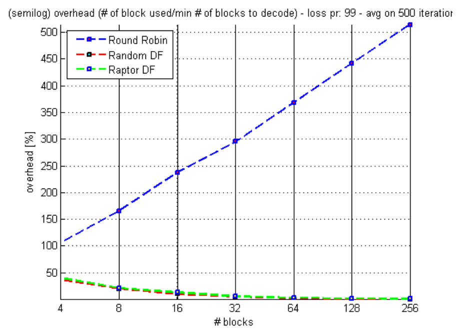
(c) Loss probability $p = 50\%$. Raptor has less overhead than Random.



(d) Loss probability $p = 50\%$. Notice the growing overhead of Round Robin method.



(e) Loss probability $p = 99\%$. Raptor is still better than Random. Overhead is minimum when $K = 256$.



(f) Loss probability $p = 99\%$. Notice the growth of Round Robin: from equation (3.2), the number of rounds needed to accomplish a transmission is approximately 552. This will result in a terrible delay.

Figure 3.10: ThunderStorm benchmarking: overhead with Round Robin, Random and Raptor. Lower overhead is better.

in this case would be better than using Raptor. That's not true. We have graphed the overhead at varying loss percentage. Looking at those graphs, we can see that Raptor achieve a lower overhead than Random: when the loss percentage is zero, Raptor's overhead is zero (Raptor is systematic); Random's overhead, instead, is nonzero even if the loss probability is zero. In an epoch of quad core, having a less overhead is preferable than a little difference (a maximum of 0.1 seconds) of CPU usage; having less overhead will lead in lesser bandwidth usage, the real missing resource of our days. So, again, Raptor is better than Random.

There is only one case when we are required to use Random Digital Fountain: if the BitTorrent piece size is $32 \cdot 1024$, we have 2 blocks. Raptor Digital Fountain does not work with less than 4 blocks: so, if piece size is 32768, we must use Random Digital Fountain, as it is the only choice.

3.7 BitFountain: modifications to Mainline client

In the following paragraphs we will briefly describe our modifications to BitTorrent client that make the essence of BitFountain client. Please see the source code for implementation details.

3.7.1 Uploader

As the upload function only access `StorageWrapper` at the specified `index`, at offset `begin` and reads a total of `length` bytes, we have to do nothing here. This function is transparent to Digital Fountain.

3.7.2 Downloader

The `Downloader` reads every `PIECE` message, extracts the block and send it to the `StorageWrapper`. In return, the `StorageWrapper` notifies the `Downloader` if the piece is completed or not:

- (normal case) if the piece is not completed, enqueue more standard request (with `begin` and `length` as the BitTorrent protocol prescribes).
- (random case) if the piece is not completed but we have completed all the standard request, we must make additional request: they are *random requests*. In fact, the peer fill the begin of these requests with an `offset` that is over the piece size (e.g. if the piece size is 4×1024^2, the offset of the first random request will be 4×1024^2, then $4 \times 1024^2 + 16384$, then $4 \times 1024^2 + 32768$ ... and so on). In this way, the seed (sequence number) of the request will keep counting incrementally.
- (normal case) if the piece is completed, advertise other peers with `HAVE` message.

We show the `StorageWrapper` decoding area that we are going to describe in the following paragraphs:

```
def DF_piece_came_in(
self, index, begin, piece, source = None):
    [...]
     if self._piecelen(index) == self._piecelen(0):
          if self._piecelen(0) == 32768:
               try:
                    x = dfwrapper[index]
               except KeyError:
                    x = RandomDF(int(self._piecelen(0)
                    /self.config['download_slice_size'])
                    ,self.config['download_slice_size'])
                    dfwrapper[index] = x
               readyToDecode = x.use_random_block(
                    int(begin/len(piece))
                    ,piece)
          else:
               try:
                    x = dfwrapper[index]
               except KeyError:
                    x = Raptor(int(self._piecelen(0)
                    /self.config['download_slice_size'])
                    ,self.config['download_slice_size'])
                    dfwrapper[index] = x
               readyToDecode = x.decode(
               int(begin/len(piece))
               ,piece)
[...]
          if readyToDecode == True:
               print 'Decoding DF piece %d' %(index)
               self.storage.write(self.places[index]
               * self.piece_size, x.retrievedata())
               del dfwrapper[index]
               del x
               del self.stat_dirty[index]
               if sha(self.storage.read(self.piece_size
               * self.places[index],
               self._piecelen(index))).digest()
               == self.hashes[index]:
                    print 'DF Piece %d
                    validated OK' % index
                    print '#' * 50
[...]
               else:
                    print 'WARNING:
                    DF Piece %d validated KO.
                    \nRe-downloading...' % index
                    print '#' * 50
                    self.data_flunked(
                    self._piecelen(index),
                     index)
                    self.inactive_requests[index] = 1
                    self.amount_inactive +=
                    self._piecelen(index)
                    self.stat_numflunked += 1
[...]
                    return False
                return True
            elif not self.inactive_requests[index]
            and not self.numactive[index]:
                return 'request more'
        else:
            return self.piece_came_in(
            index,
            begin,
            piece,
            source)
```

As you can see, we use the correct decoding method (Random or Raptor) by retrieving the piece size: if the piece size is more than 32k, we use Raptor, otherwise we use Random. If we collected enough random blocks, we can decode the piece than hash check it to test its integrity. If we can not decode, we must request for additional blocks, and we have to distinguish if we are enqueuing standard request or additional requests:

```
def new_request(self, index):
    # returns (begin, length)
    if self.inactive_requests[index] == 1:
        self._make_inactive(index)
    self.numactive[index] += 1
    self.stat_active[index] = 1
    if index not in self.stat_dirty:
        self.stat_new[index] = 1
    rs = self.inactive_requests[index]
    r = min(rs)
    rs.remove(r)
    self.amount_inactive -= r[1]
    #if self.amount_inactive == 0:
    #    self.endgame = True
    return r

def new_request_DF(self, index):
    request_size =
    self.config['download_slice_size']
    try:
        begin = indexptr[index]
        begin += request_size
        indexptr[index] = begin
    except KeyError:
        indexptr[index] = self.piece_size
        begin = self.piece_size
    self.numactive[index] += 1
    self.stat_active[index] = 1
    if index not in self.stat_dirty:
        self.stat_new[index] = 1
    return (begin, request_size)

def _request_DF(self, interest):
    begin, length = self.downloader.storage.
    new_request_DF(interest)
    self.active_requests.append((interest, begin, length))
    self.connection.send_request(interest, begin, length)
```

Standard requests are those that comes with BitTorrent: we split the piece in blocks of the same length, and then we request until `begin` = `piecesize` − `length`. For DF requests, instead, we made requests characterized by having a `begin` that is greater or equal than `piecesize`. As we already said, in a Fountain capable client, $\frac{\text{begin}}{\text{length}}$ is used as a seed for the Fountain encoder or decoder.

```
def get_DF_piece(self, index, begin, length):
[...]
piece = self.storage.read(
self.piece_size * self.places[index],
self.piece_size)
if self._piecelen(0) == 32768:
    block = create_random_block(
    rand50list(int(begin/length),
    int(self._piecelen(0)/
    self.config['download_slice_size'])),
    piece).tostring()
    print '\tsending randomDF block'
else:
    try:
        x = dfwrapper_tx[index]
    except KeyError:
        x = Raptor(int(self._piecelen(0)/
        self.config['download_slice_size']),
        self.config['download_slice_size'])
        dfwrapper_tx[index] = x
    block = x.encode(int(begin/length)
    ,piece).tostring()
    print '\tsending raptorDF block'
return block
```

3.7.3 Storage

Reading a block to upload When the storage is requested to read a piece, it is called by the `Uploader`, because some peer wants a block. First of all, we must read the piece entirely. Then we choose between Random Digital Fountain and Raptor, remember the remarks made before:

- if the piece size is 32768, we must use Random Digital Fountain. Encode the random block by randomly XOR'ing all the blocks within the piece, using $\frac{\text{begin}}{\text{length}}$ as seed for the random number generator.
- otherwise, we use Raptor Digital Fountain because they can ensure better performances than Random. Encode the random block by randomly XOR'ing all the blocks within the piece, using $\frac{\text{begin}}{\text{length}}$ as seed for the random number generator.

Once finished, send the random block to the peer.

Writing a block We designed our client to maintain a decoding area for every piece (see figure 3.11 on the next page): when a random block comes in, the client tries to decode that piece, using $\frac{\text{begin}}{\text{length}}$ as seed for the random number generator:

- if the decode phase does not have success, because more random blocks are needed, the client will notify the `Downloader` to enqueue more *special requests* for that piece.
- if the decode phase have success, the client continue with normal BitTorrent operations: hash checking, etc.

3.7.4 Interoperability with "normal" clients

BitFountain is also interoperable with "normal" (e.g. not Digital Fountain capable) clients. To discriminate normal peers, we had two choices:

- send a BEP (BitTorrent Enhancement Proposal) to BitTorrent Community, asking to reserve a bit in the handshake message. This process is slow and relatively useless.
- recognizing the capabilities of each client by inspecting its peer ID:
  - if the client peer ID is BitFountain:
    - we answer to a block request by ignoring which specific data block is requested and we send out random blocks.
    - we request for random blocks to other peers.

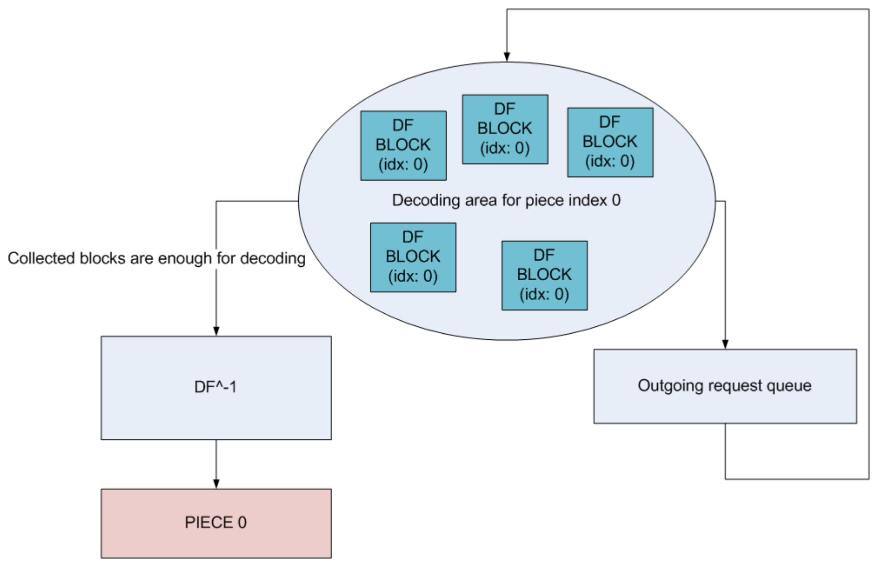


Figure 3.11: Piece decoding: if we have collected enough block, decode the piece. Otherwise, enqueue more requests. Every piece has a decoding area like the one outlined here.

- if the client peer ID is not BitFountain:
  * we answer to a block request in the normal way, slicing a piece of that `index` at the requested `begin` and reading a total amount of `length` data.
  * we request for normal blocks to other peers.

Obviously, we have chosen the second way. Generate a new peer ID is easy:

```
def make_id():
    myid = 'BF' +
    version.split()[0]
    .replace('.', '-')
    myid = myid +
    ('-' * (8-len(myid)))
    +sha(repr(time())+ ' ' +
    str(getpid())).digest()[-6:].encode('hex')
    print 'Client ID: ' + str(myid)
    return myid
```

Testing if a client is Fountain capable is straightforward too:

```
if 'BF2-0-0' in self.connection.id:
    insert = self.downloader.storage.
    DF_piece_came_in(index,
    begin,
    piece, self.guard)
    if insert == 'request more':
        self._request_DF(index)
else:
    insert = self.downloader.storage.
    piece_came_in(index,
    begin,
    piece, self.guard)
```

As you can see, in `Downloader` we call two different methods if the sender is Fountain capable or not. The same applies for the `Uploader`. Notice that BitFountain client can dummy-encode pieces downloaded from normal BitTorrent client in order to use such blocks:

– if we are using Random Fountain: “normal” data block can be injected in decoding process by simply associating them with a bitlist with one single 1 (corresponding to the single received block).

– if we are using Raptor Fountain: “normal” data block can be injected in decoding process by simply associating them with a proper seed (remember: first K output blocks of the coding system coincide with the original K blocks).

In this way, every received block from “normal” clients is useful for the decoding process.

Chapter 4

Security issues

4.1 BitTorrent security

Many peer-to-peer networks are under constant attack by people with a variety of motives. Examples include:

- poisoning attacks (e.g. providing files whose contents are different from the description).
- polluting attacks (e.g. inserting bad chunks/packets into an otherwise valid file on the network).
- leechers (users or software that make use of the network without contributing resources to it).
- insertion of viruses to carried data (e.g. downloaded or carried files may be infected with viruses or other malware).
- malware in the peer-to-peer network software itself (e.g. distributed software may contain spyware).
- denial of service attacks (attacks that may make the network run very slowly or break completely).
- filtering (network operators may attempt to prevent peer-to-peer network data from being carried).
- identity attacks (e.g. tracking down the users of the network and harassing or legally attacking them).
- spamming (e.g. sending unsolicited information across the network).

4.1.1 Anonymity

BitTorrent does not offer its users anonymity. It is possible to obtain the IP addresses of all current, and possibly previous, participants in a swarm from the tracker. An attacker obtaining these IP addresses can perform an attack on clients.

4.1.2 Integrity

It is also practically impossible to send garbled data to another client. Upon receipt of data, if the `index`, `begin` and the `length` of the data received do not fit those of the recipient's requests, then the bytes are discarded and will not be saved anywhere in the recipient's system. To prevent data poisoning and bad peers from sending garbled data to other downloaders, SHA1 hash check is done whenever the download of a piece is completed. We enumerate the steps during a typical data transfer:

1. the uploader packs the data to be sent in the `PIECE` format.
2. as this data streams into the connection of the downloader, the downloader reads the first 4 bytes to determine the number of bytes that come right after. Since this is a `PIECE` message, these 4 bytes would give a value of 9 + block length.
3. separating the incoming message into appropriate parts, the downloader gets the `index`, `offset` and actual block data. The downloader then compares the `index`, `offset` and block data `length` to those of the requests sent. If any of the 3 fields do not match, these bytes are discarded. If they match, the block is saved within the correct piece.
4. if all blocks within a piece are retrieved, the SHA1 hash of the complete piece is compared with the 20-byte hash recorded in the torrent metadata file:
   - if the SHA1 hash check fails, then the IP address of the sender is banned all blocks within the particular piece are discarded[1].
   - otherwise, the piece is saved into the receiver's file storage. The downloader's bitfield is updated and the downloader sends out to all peers a `HAVE` message containing the index of this piece.

It is assumed that it is unlikely for a poisoned piece to pass a SHA1 hash check. In fact, we can assume it is very difficult to obtain a SHA1 hash

[1]Each client maintains a ban list. A ban list contains the IP addresses of peers who have sent bad blocks to the client. Any peer that is in a client ban list will not be allowed to connect that client at all. This list is semi-permanent, which means that is maintained for the period of time that the client is running. When the BitTorrent client application exits, this list is lost.

collision, thus bad peers can not corrupt pieces and send them. In other words, there is no known way for bad peers to send fakes in the BitTorrent network, unless the file itself is a fake.

Discarding blocks

As we already said, the minimum transfer unit in BitTorrent is a 16k blocks, which is much smaller than full size piece. Pieces are what's referred to in `HAVE` messages and what the peers have hashes of. This causes a few problems:

- peers have no way of knowing which block was bad if a piece they download fails hash check, and if they're streaming data they can't display it until a full piece is downloaded for hash verification purposes.
- there is a huge waste of bandwidth when a piece does not pass hash check: in fact, the peer must discard all the blocks (even all the correct ones) and then re-download all the blocks within the piece. It is better to discard only bad blocks, and re-download only them: this will lead to a save of bandwidth.

The problem of the current situation is clear: we have to validate blocks, distinguishing between good and bad blocks.

A simple extension, proposed by BitTorrent team [48], could be an initial approach to validation. In addition to the piece hashes, the .torrent file includes another set of hashes, which also includes one hash per piece, but instead of a hash of the piece as a whole it is the hash of all the 16k blocks of that piece. Let's estimate the total amount of additional data to store. For example, a torrent file consists in 755 pieces of 2MB (thus total download size is 1.47GB). Every piece is made up by 128 blocks. Therefore, the torrent file has to contain a total of 96640 hashes (2MB): as we can see, this solution is suitable for normal BitTorrent.

However, this solution is inapplicable with Digital Fountain (either applied to blocks or pieces) and we will illustrate the reason. As a consequence, we propose another way to validate blocks. Keep reading for further details.

4.2 BitFountain security

4.2.1 Attack model

We assume that an object is transmitted from a sender to a remote receiver over an unsecured channel. We further assume that there is an intelligent and powerful attacker in this channel. This attacker has the full control of the unsecured channel and can potentially:

- intercept all the packets sent by the sender.

- delay, reorder and delete these packets.
- send chosen, forged packets to the receiver.

The goal of the attacker can be to corrupt as many transmitted objects as possible, in order to mount a Denial of Service (DoS) attack. This attack is trivial to launch since the attacker can corrupt all packets sent to the receiver (in practice corrupting a subset of them is sufficient). If a receiver does not know how to discard them, they are used in place of authentic ones. In the end, the receiver reassemble a file which is not the same as the encoded one because of corruption propagation. In fact, in order to recover from packet erasures, the erasure code decoder rebuilds the missing data with the received data and parity symbols, in a recursive way. Let us now consider the following constraint equation:

$$S_0 \oplus S_1 \oplus S_2 \oplus S_3 = 0$$

We assume that the value $s_1; s_2; s_3$ of the symbols $S_1; S_2; S_3$ have been received but not s_0. Then:

$$s_0 = s_1 \oplus s_2 \oplus s_3$$

Assume that an attack has been launched and that the received value of symbol S_3 is the corrupted value s'_3, such that: $s'_3 = s_3 \oplus \varepsilon$. The decoded value of symbol S_0 will be:

$$s_1 \oplus s_2 \oplus s'_3 = s_1 \oplus s_2 \oplus s_3 \oplus \varepsilon = s_1 \oplus \varepsilon$$

which has inherited the corruption of S_3. Therefore, if a corrupted symbol is used in a decoding operation, the decoded symbol inherits from the corruption. Furthermore, each newly decoded symbol can be used to decode other symbols recursively, so an avalanche of corruptions can happen during decoding. We call this the *corruption propagation phenomenon*. This will happen either using Random or Raptor.

Even if we primarily consider intelligent attackers, the attack might also be non-intentional. For instance, it might be caused by transmission errors that have not been detected and/or corrected by the physical layer erasure codes/CRC. Even though it is usually considered that these corruptions are rather infrequent, they are not totally impossible, especially in some harsh wireless environments. In our work, we consider those non-intentional attacks are special cases of the attack model, and the challenge here is to detect the corruption, even when a very small number of bytes are corrupted in the object.

4.2.2 DF applied to BT pieces

We are going to describe the corruption propagation phenomenon when using Digital Fountain applied to pieces. An attacker, A, injects Digital Fountain random pieces into the network, claiming to be a seeder with the random piece that another peer needs to get. When another peer asks A for a random piece, A sends back a corrupted piece $M^{'}$ instead of M. At some point, when the receiver has collected enough random pieces, it can decode the random piece received: at this point it will realize that there is something wrong. In fact, decoded pieces does not pass the hash check. In this way, an attacker can damage the distribution of a file, leading the receivers to a useless decoding process (receivers must also request the re-transmission of the random pieces from the beginning): it is a DoS (Denial of Service). In the worst scenario, a peer downloads a total amount of data equal to the size of the original file; then, it discovers that all the data is corrupted and has to be discarded, leading to a new download process that starts from the beginning.

4.2.3 DF applied to BT blocks

We are going to describe the corruption propagation phenomenon when using Digital Fountain applied to blocks. Even in this approach there is a security problem: A, the attacker, sends out corrupted random blocks. So, when another peer, B, asks A to upload a random block M, A will answer with $M^{'}$. When B has collected enough random blocks, it can decode the original piece (P). If just one single random block is corrupted, then B will reassemble the modified piece $P^{'}$. At this point, B realizes that $P^{'}$ is not what it wants because of the failed hash check process; so, it discards all the random blocks within $P^{'}$ and enqueues a new download for the blocks within P. Even in this approach, B wastes CPU cycle to decode random blocks and bandwidth to retrieve the same piece at least one more time. However, using this approach the amount of discarded data is bounded by the piece size: in the worst scenario, we are going to discard 4×1024 bytes.

4.3 Proposed solutions

We need to validate blocks (or pieces) as they are used. We are going to describe a group of solution that can be applied to BitTorrent or BitFountain with Digital Fountain applied to blocks or pieces.

An initial approach is to pair every block with an hash, calculated by the transmitter and sent to the receiver. This approach is *inapplicable*: since the verification information comes from the transmitter, an attacker could send a garbled block and calculate its hash correctly; the receiver calculates the hash and verifies that it's all correct (the transmission went successfully). However, the block is still garbled and the decoding/reassembling phase will does not finish correctly.

We are going to describe a list of possible solution to solve our problems, their strengths and drawbacks.

4.3.1 Hash publication

As we said, we could publish the hashes of all the blocks. However, in a Digital Fountain context, this approach is not applicable.

Indicating with N the number of source blocks, with Digital Fountain we can construct a number of random blocks equal to:

$$S = \sum_{k=0}^{N} NCk \tag{4.1}$$

where NCk is the binomial coefficient:

$$NCk = \binom{N}{k} = \frac{N!}{(N-k)!k!}$$

Now, the greatest number of blocks to XOR is $N = 256$ (corresponding to a piece size of 4×1024 bytes). We said in chapter 2 that the worst scenario for Raptor codes is when the degree is 40, which means 40 input blocks to XOR together. In this case, the summation of S is truncated to $N = 40$. This will result in a S equal to 1.2844×10^{47}. If we use SHA1 hash, this will require 2.0550×10^{49} bytes ($1.69985616 \times 10^{25}$ yottabyte) to store all hashes for *only* one single piece. This demonstrates that hash publication is not a solution of our problem.

Hash publication with message passing

If we have the hashes of the all the input data blocks, random block validation could be done on-the-fly: if we use message passing decoding method, we can validate a relatively large fraction of the random blocks (those encoded with Raptor and having a degree equal to 1 – there are many random blocks with degree unitary). However, this approach has many drawbacks: we can not use message passing with Random Fountain because their degree is always greater than 1. In addition, using message passing with Raptor will lead to an overhead equal to 20% (instead of 2% when we use Gaussian elimination).

4.3.2 Verification function distributive to XOR

We have to design a function to validate every single random block exchanged by peers. As we know, a random block is an arbitrary number of input blocks XOR'ed together.

Let's consider a simple example, denoting M as a random block and X and Y are two input blocks. So:

$$M = X \oplus Y$$

The hash of X and Y are published on the torrent file, as BitTorrent specification says. We are searching for a verification function f that satisfies the following equation:

$$f(M) = f(X \oplus Y) = f(X) \oplus f(Y) \tag{4.2}$$

In this way, we can validate M by simply verifying the hash of X and Y by looking for them in the torrent file. The validation function f could be:

- a CRC function. This kind of function solves the problem, because CRC function are distributive to XOR operation. This function is suitable to detect unintentional transmission errors. However, one attacker could simply generate a two messages M_1 and M_2 with the same CRC, invalidating our effort. Therefore, this solution is not robust, and thus, inapplicable.

- a hash function that is distributive to XOR. We are going to demonstrate that this kind of function is poorly designed and will run into hash collision very quickly[2].

  Let's consider an hash function H which output size is Q bits; let's also suppose that this function is distributive to XOR. To begin with, we randomly generate messages M_i of different sizes, and then we compute the hash of each message. After we have generated at most $Q + 1$ messages (one more than the dimension of the vector space) we can assert that some hashes are linearly dependent, which means that one of them could be expressed as a linear combination of the other Q messages:

$$\begin{aligned} H(M_x) &= A_1 H(M_i) \oplus A_2 H(M_j) \oplus \dots A_Q H(M_k) \\ &= H(A_1 M_i \oplus A_2 M_j \oplus \dots A_Q M_k) \\ &= H(M_y) \end{aligned} \tag{4.3}$$

  where A_i are binary coefficients that could be calculated by solving a linear equation system. When we have found this combination, we can identify two different messages (preimages) M_x and M_y that have the same hash, leading to a collision. Again, this solution is inapplicable.

4.3.3 Iterative subset decoding

Suppose that we have downloaded a number of blocks (either normal or random blocks) equal to M, and we use them to reassemble the piece; now

[2]A hash collision or hash clash is a situation that occurs when two distinct inputs into a hash function produce identical outputs. All hash functions have potential collisions, though with a well-designed hash function, collisions should occur less often (compared with a poorly designed function) or be more difficult to find.

suppose that the piece does not pass the hash check. This approach consists in downloading a slightly number of X additional blocks; then, we generate every subset of the downloaded blocks and we try to iteratively decode every subset until the reassembled piece pass the hash check. In this way, if some blocks are corrupted, we do not have to re-download all the blocks: we download only a subset of all the blocks, hoping that, at some point, we reassemble the correct piece. Just notice that we have no guarantee that the decoded piece will pass hash check process; in this way, X is not bounded and there is not a fixed value for X.

This approach has a main problem when we are dealing with Digital Fountain. Let's suppose that we are in the worst scenario for this experiment, that is a piece size equal to 4×1024^2 bytes, corresponding to 256 random blocks of 16384 bytes each. We denote the minimum number of random blocks to decode with M. We proceed as follows:

1. we download 256 random blocks.
2. suppose that we can decode these blocks without request additional blocks, which means that overhead is zero. So, $M = 256$.
3. we reassemble a piece which does not pass hash check.
4. we select a number of X. For example, we select X equal to 5.
5. we download additional a number of X blocks.
6. we calculate the number every possible decodable subset (B) of all the received blocks:

   $$B = \binom{M+X}{M} = \frac{(M+X)!}{(M+X-M)!X!} = \frac{(M+X)!}{X!^2}$$

   In our case, $B = 9.7115 \times 10^9$. Using ThunderStorm and Raptor Fountain codes, we can infer from figure 3.9 on page 86 that we need at least 1 second to decode 256 blocks. As a consequence, in our case we need at least B seconds, which means that in the worst case a total of 9.7115×10^9 seconds to decode a piece. If $X = 1$, we have that $B = 257$, which will lead to a total of 257 seconds to decode. Because of this waiting time, even with one additional random block, this solution is inapplicable.

4.4 Distributed validation

Past proposed solutions give one simple hint: verification information must not come from the sender of the block. So, we designed a peer wire protocol to validate blocks: we were inspired by the famous Byzantine Generals' Problem [14]. Our solution is suitable either for normal BitTorrent and for Digital Fountain applied to BitTorrent blocks or pieces.

4.4.1 Operation

Basically, we download blocks as we already did, but we ask to a *random* peer to send us the hash for the block (or piece) that we want to validate. If the hash is equal, then we may assume that the block (or piece) is valid. If not, we may assume that the block is corrupted and has to be re-downloaded. In this way we validate every block and we can also detect which peer is sending bad data, and take countermeasures accordingly. As a consequence, we avoid to re-download all the blocks within a piece in case of failed hash check; this will lead to a save of bandwidth.

There are three constraint for the successful application of the algorithm:

- the number of connected peers must be greater or equal to three (one peer for the upload, one peer for the download and one peer for the distributed validation).
- the peers who request for hash must send requests to peers who have that particular block (or piece)[3].
- obviously we must ask the validation to a peer whence we did not downloaded the block. The peer is selected in a random way. This ensures that two (or more) peers can not cooperate to hijack the reassembling of the piece.

In the following paragraphs, we suppose that there are two groups of peers in the network:

- good peers (G) which send clean blocks and answer to validation requests with the correct hash (confirmative hash).
- bad peers (B) which send corrupted blocks and answer to validation requests with faked hash (contradictive hash) or do not answer at all.

The network is composed only by good and bad peers ($N = B + G$). Notice that we can be deceived by bad peers in believing that a good block is corrupted (because we do not receive a confirmative hash); being deceived in believing that a bad block is clean it is much more harder: since we are using a strong cryptographic hash, there is a negligible probability that another bad peer confirms a corrupted block. In addition, it is too difficult for a bad peer to forge a block that has the same hash of one clean block. This means that it is nearly impossible for a good peer to validate a bad block.

First of all, we introduce some probabilities:

- probability of contacting a good peer: $p = \frac{G}{N}$.

[3]This can be easily determined by looking at BitTorrent `BITFIELD` or `HAVE` messages, as we said in chapter 1.

- probability of contacting a bad peer: $q = \frac{B}{N}$.
- probability of downloading a good block: p.
- probability of downloading a bad block: q.
- probability of having a good hash confirmation: p.
- probability of having a bad hash confirmation: q.

Obviously, $p + q = 1$. We also introduce the *round-trip time* (RTT), which is the amount of time it takes for a validation request to get from the sender to receiver i and then the hash back to the sender); we denote it with T_i.

Although there is still that possibilities of being deceived, we have one final checkpoint: the hash validation described by the BitTorrent protocol. In fact, even in the cases described in the previous paragraph, we realize that something is wrong when the reassembled piece does not pass hash check process.

We must avoid to discard a good block if it is not confirmed by a bad peer; thus, it is much more convenient to ask for a validation to another random peer. So, the main idea of this approach is to make at most R validation requests. At the end of those requests, if the block is not confirmed, we might assume that the block is corrupted and we must re-download it. After sending a request, we wait for a specified time: this is the *timeout* T_O. When the timeout has passed, we may assume that the request was lost and we can reiterate the request to another peer. We design two strategies when a peer asks for a validation:

Serial validation requests We request for the hash of a block to a random peer:

- if the received hash is equal to the hash computed on the received block, the block is confirmed and we might assume that is clean.
- if the peer does not answer or the hash is not equal, we reiterate the algorithm by asking for a validation request to another peer, until we have reached a total of R validation requests.

Serial validation minimizes the amount of request that a peer makes during validation process. In the better case, a peer experiences a delay D at most equal to:

$$D = T_1 \tag{4.4}$$

where T_1 is a confirmative hash. In the worst case, D is at most equal to:

$$D = \max((T_1 + T_2 + \ldots + T_R), T_O) \tag{4.5}$$

where $T_1, T_2, \ldots$ are RTT of contradictive hashes and T_R can be either the RTT of a confirmative or a contradictive hash (in this case we may assume that the block is corrupted and we must re-download it) and T_O is the timeout described before. If a request does not have an answer until the timeout, the peer proceed in asking the validation to the next peer.

Parallel validation requests We send R validation requests to R random peers simultaneously. We wait for the first confirmative hash to proceed. In this way we minimize the amount of time needed for validating. In the better case, a peer experiences a delay D equal to:

$$D = \min(T_1, T_2, \ldots, T_G) = T_{min} \tag{4.6}$$

where $T_1, \ldots, T_G$ are RTT of confirmative hashes that come from good peers. T_{min} is the answer of the fastest good peer. In the worst case, the delay D is highly bounded by the timeout:

$$D = \max((T_1, T_2, \ldots, T_R, T_G), T_O) \tag{4.7}$$

where $T_1, T_2, \ldots, T_R$ are RTT of contradictive hashes and T_G is the RTT for a confirmative hash coming from the only slowest good peer that is answering to our validation request and T_O is the timeout described before.

If we wait for all validation answers to arrive, we can estimate the percentage of good peers in the network in this way[4]:

1. a fraction of all the validation answers confirms the received block, and we denote it with r_{OK}. We assumed that good peers confirms the block, so:
$$r_{\mathrm{OK}} = R\frac{G}{N} \tag{4.8}$$
2. we can now deduce the fraction of the good peers in the network from equation (4.8) by dividing the requests which confirm the received piece with the total number of validation requests done:
$$\frac{G}{N} = \frac{r_{\mathrm{OK}}}{R} \tag{4.9}$$

Clearly, the greater is R, the smallest is the delay experienced by the requester, but the greater will be the overhead in sending and receiving validation messages.

Serial and parallel validation achieve the same probability of validate a good block (think to the equivalence of sequential and in-block extraction). The only difference is in timing (parallel is faster) and in the consumption of the resources (parallel uses much more resources).

[4]Under the hypothesis that $G < B$.

4.4.2 A serial validation example

We describe a simple example of the serial distributed validation algorithm with $R = 3$.

- A asks for a block to B.
- B sends back a block[5], j.
- A calculates the hash for j (by using its received copy from B) ($H(j_A = x)$).
- A asks C for the hash[6] of j.
- C calculates the hash for j (by using its local copy) and sends the hash to A ($H(j_C) = y$).
- A compares the two hashes ($x == y$?):
  - if the two hashes are identical ($x = y$), then A might assume that j is correct. The algorithm stops.
  - otherwise ($x \neq y$), A must reiterate the request[7], asking D for hash of j:
    - D calculates the hash for j (by using its local copy) and sends the hash to A ($H(j_D) = z$).
    - A compares the hashes that it has for j (x, y, z):
      - if the hash received by D confirms the block, A might assume that j is correct and C has sent a wrong hash.
      - if the hash does not correspond, A reiterates the validation request by asking the block hash to E: if the received hash confirms the block, A may assume that j is correct and C and D have sent a wrong hash. Otherwise, A may now assume that j is corrupted and B has sent a bad block. It discards the block and re-downloads it. Anyhow, A stops the algorithm (we have reached $R = 3$).

4.4.3 A parallel validation example

We describe a simple example of the parallel distributed validation algorithm with $R = 3$.

- A asks for a block to B.

[5] In case of Digital Fountain, A and B both know the bitlist of the random block by sharing its sequence number.

[6] In case of Digital Fountain, it is sufficient, for A, to specify the sequence number of j.

[7] We can not determine if B has sent a faked block or C has sent a faked hash

- B sends back a block, j.
- A calculates the hash for j (by using its received copy from B) ($H(j_A = x)$).
- A asks to C, D and E for the hash of j.
- C, D and E calculate the hash for j (by using their local copy) and sends the hash to A ($H(j_C) = y$, $H(j_D) = z$, $H(j_E) = w$).
- A compares the received hashes ($x == y == z == w$?):
  - if any of the three hashes confirm the block (see (4.10)), A may assume that the block is valid.

$$x == y \vee x == z \vee x == w \tag{4.10}$$

  - otherwise A may assume that the block is corrupted and discards it. A must re-download the block.
- A stops the validation algorithm because has reached the maximum number of requests R.

4.4.4 Practical considerations

Let's suppose that:

- we have downloaded a bad block with $(1-p)$ probability. No peers will confirm this block, in fact all the R requests do not confirm that block. Notice that the probability of discarding a good block because it is not confirmed is equal to the probability of choosing only bad peers during the download and validation process, which is very negligible and it is equal to $(1-p)^R = q^R$.
- we have downloaded a good block with p probability. We make R validation requests, and we get pR hash confirms (from good peers) and qR hash contradicts. Now we want to estimate the average number of hash contradictions before getting a confirm. In fact, suppose that we have downloaded a good block: if we get only hash contradictions from bad peers, we discard a clean block; we must avoid this situation. So, to receive a hash confirmation (a good answer), we must make an average number R of requests equal to (with some approximations on p, remember Geometric distribution mean):

$$R = \frac{1}{p} \tag{4.11}$$

In this way, if $p = 0.1$, $R = 10$; if $p = 0.9$, $R \approx 2$. Generally speaking, $R > 2$ is a good choice.

As we said, we want to avoid the situation where a peer download a clean block and then discards it because of failed validation (in this case the peer asked for validation hashes only to random bad peers – by the way, it is a random selection of which peers to ask for validation, so a failure can occur). The key point is a good selection of the number of requests R. So, to have an high probability (99.99%) of a confirm (good hash) when the block is clean, we remember the Geometric cumulative distribution function (with some approximations on p):

$$F(X, p) = 1 - (1 - p)^X \qquad 0 \leq p \leq 1 \tag{4.12}$$

We deduce X from the following disequation:

$$\begin{aligned} F(X, p) &> 0.9999 \\ 1 - (1 - p)^X &> 0.9999 \\ -(1 - p)^X &> 0.9999 - 1 \\ (1 - p)^X &< 0.0001 \\ \log(1 - p)^X &< \log(10^{-4}) \\ X \log(1 - p) &< -4 \end{aligned}$$

Now, remember that $\log(1 - p)$ is a negative quantity, thus:

$$X > -\frac{4}{\log(1 - p)} \qquad 0 < p < 1 \tag{4.13}$$

From equation (4.13), the number of requests R is equal to:

$$R \approx \lceil X \rceil \tag{4.14}$$

We graphed R on p in figure 4.1 on the next page. In this way, given the fraction of good peers in the network, if a peer makes at most R requests, there is a 99.99% probability that we have a block confirm if the block is clean.

Now, we want to estimate the probability of validate correctly a clean block: this occur only if we contact at least one good peer (which sends a confirmative hash) during our distributed validation. We graph this probability with various values of N and with different values of R in figures 4.2 on page 110 and 4.3 on page 111. Notice that, when choosing which peers ask for a validation, already contacted peers are excluded from the selection.

4.4.5 Annexes to BitTorrent peer wire protocol

To implement distributed validation, we need a peer wire protocol to ask, to send and to receive hashes. We can add those messages to already existent

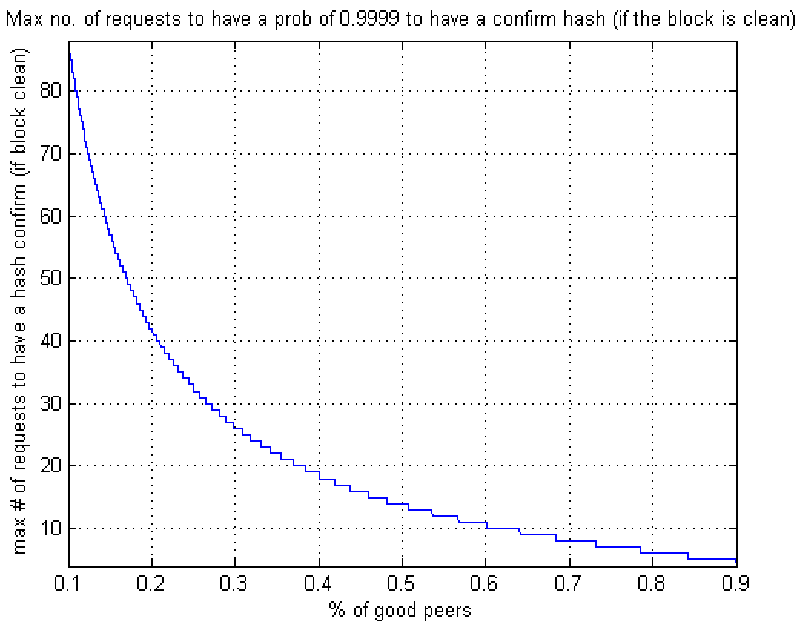


Figure 4.1: The maximum number of requests to have a 0.9999 probability to have a hash confirm (if the block is clean).

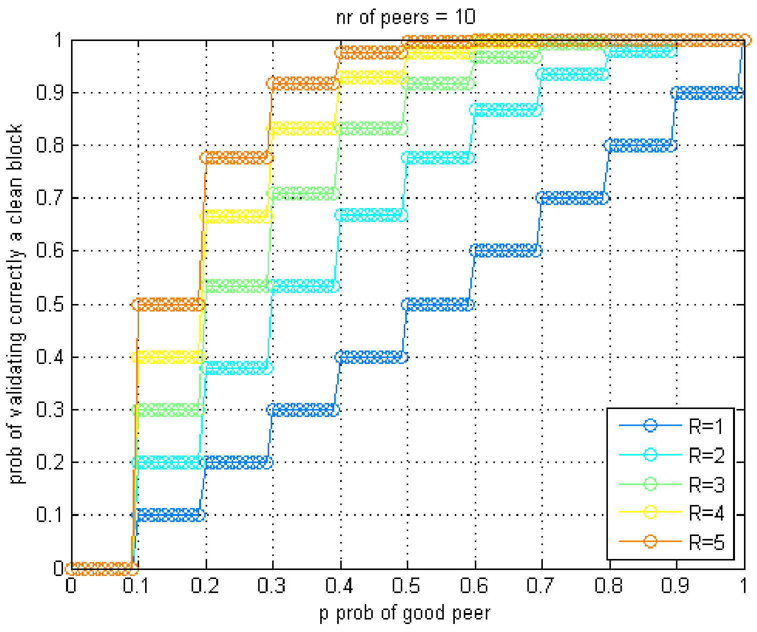
underneath BitTorrent's peer wire protocol; in this way, peers exchange block hashes utilizing the already existent protocol. We designed two messages to be added in the peer wire protocol. Notice that the specified sizes are congruent to the size of existent BitTorrent messages and include the size of the SHA1 hash:

- `HASHREQ`: `<000d><6><[index][begin][length]>`
  Request for the hash of a block. `index`, `begin` and `length` are 4 bytes each. In case of Digital Fountain, we use $\frac{\text{begin}}{\text{length}}$ as sequence number (bitlist), as we already said in chapter 3.

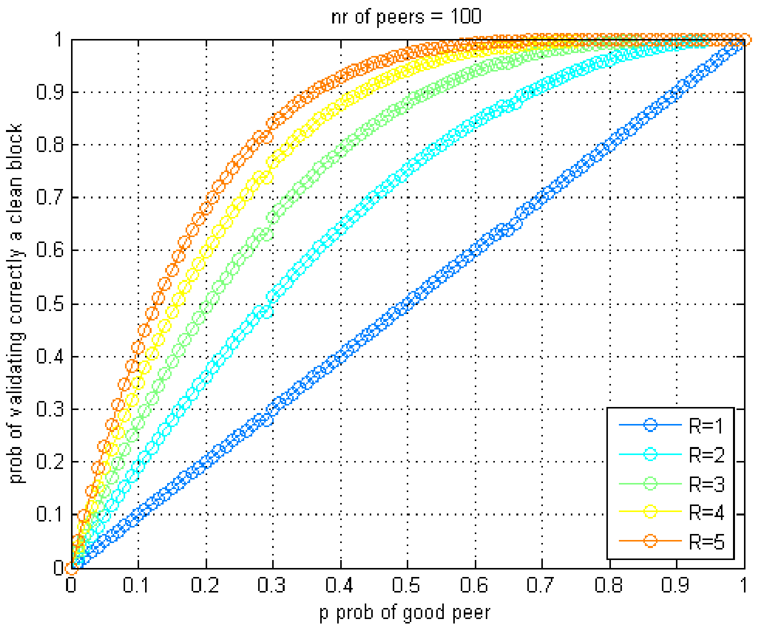
- `HASH`: `<0009+20><7><[index][begin][block]>`
  Send a SHA1 hash (20 bytes) of a specified block. This corresponds to a `HASHREQ` message sent by the recipient earlier. `index` and `begin` are 4 bytes each. In case of Digital Fountain, we use $\frac{\text{begin}}{\text{length}}$ as sequence number (bitlist), as we already said in chapter 3.

4.4.6 Validation timing

Validation process (either for normal or Digital Fountain approach) triggers an important question: when we have to validate received blocks? We have two choices:

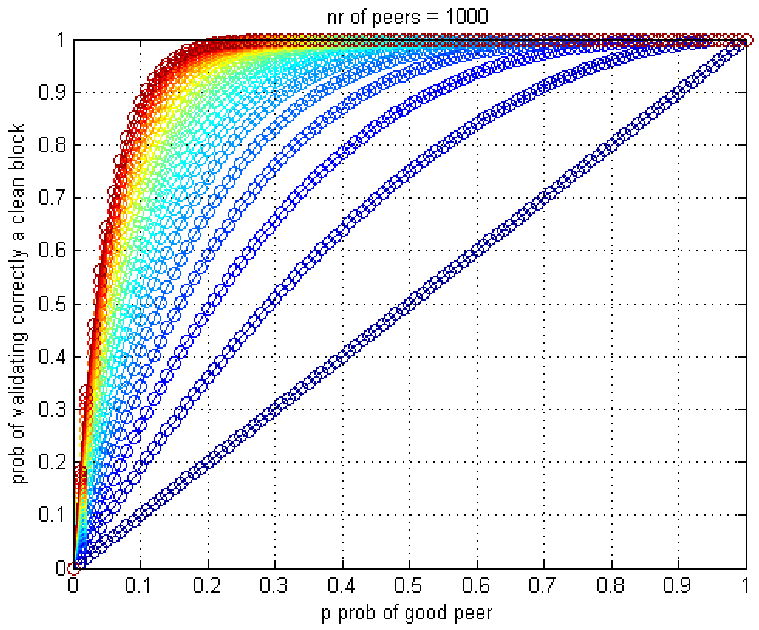


(a) $N = 10$, R varying from 1 to 5.

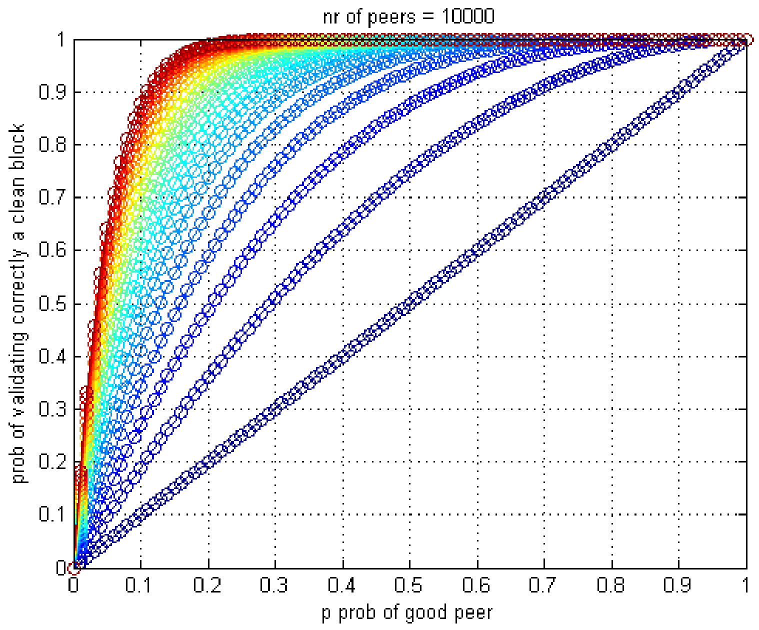


(b) $N = 100$. R varying from 1 to 5.

Figure 4.2: A comparison of the probability of validating a clean block correctly with different values of R. In these plots, we are considering small P2P networks ($N = 10, 100$).



(a) $N = 1000$. R varying from 1 to 20.



(b) $N = 10000$. R varying from 1 to 20.

Figure 4.3: A comparison of the probability of validating a clean block correctly with different values of R. In these plots, we are considering large P2P networks ($N = 100, 1000$).

- pre-validation.
- after-validation.

Every approach has its strengths and drawbacks. Let's discuss them.

Pre-validation

With a pre-validation approach, we validate every single block received using the distributed validation described before.

Strengths

- the reassembled piece will *always* pass the hash check process (if we suppose that we were not deceived during the validation process).
- in case of Digital Fountain, decoding is only executed once, thus we do not waste computational power uselessly.

Weaknesses

- the validation process will certainly run into modest delay (because of request and answer for hash), even if all the blocks are correct.

After-validation

In this approach, we validate all the blocks within a piece *only* if the reassembled piece does not pass the hash check.

Strengths

- we do not experience delay if the received blocks are all correct.

Weaknesses

- in case of Digital Fountain we are going to invoke the decoding process at least twice.

Considerations Let's suppose that we use after-validation and we are in the worst scenario for this experiment, that is a piece size equal to 4×1024^2 bytes, corresponding to 256 blocks of 16384 bytes each. No matter if they are normal block or random block. Denoting with c the number of corrupted blocks, we compare BitTorrent with BitFountain with distributed validation (see figure 4.4 on page 114):

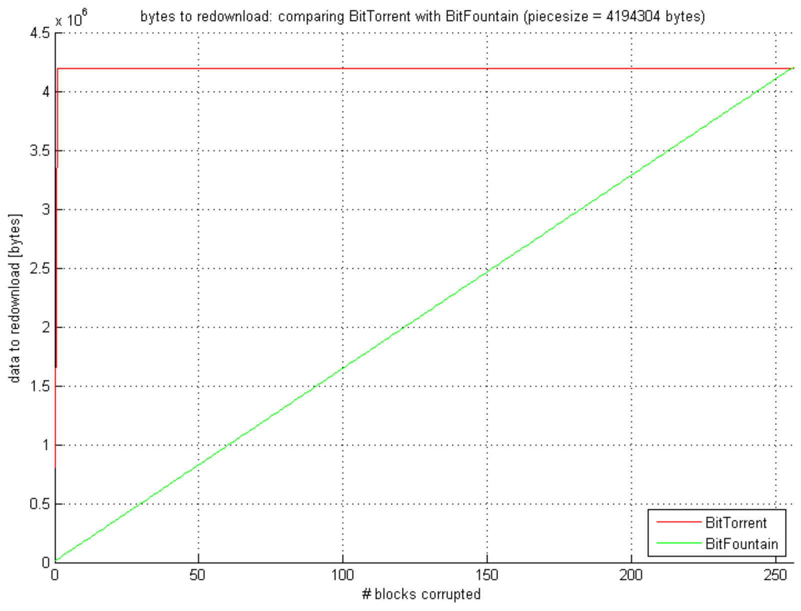
- in BitTorrent, if just one single block is corrupted, the client will discard all the 256 blocks. So, the total amount of data to re-download t ranges from 0 bytes (when $c = 0$) to 4×1024^2 bytes (when $c \neq 0$).

- in BitFountain, t depends on how many blocks are corrupted. Suppose that we discriminate if every block is corrupted or not by using only one request-for-hash round, and then we re-download every corrupted block. The total amount of data could be easily calculated:

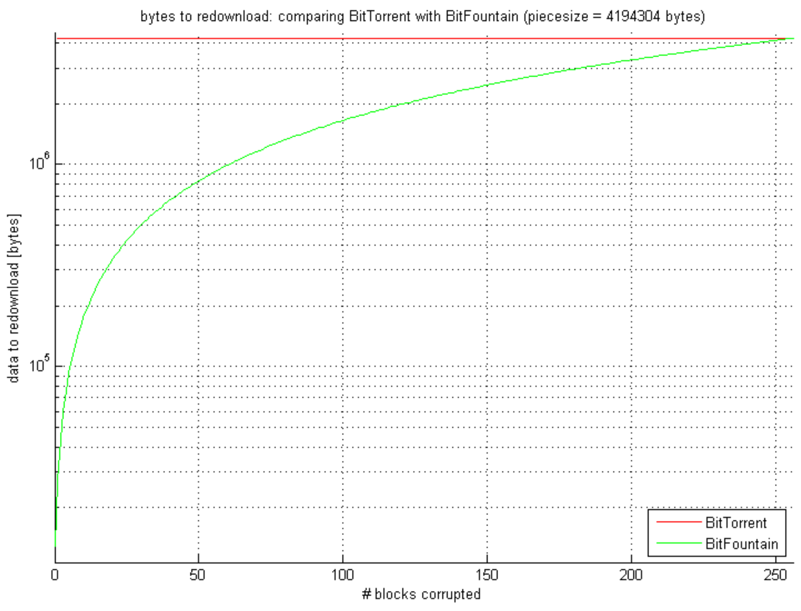
  $$t = 16384c + 256(13 + 29)$$

  where t is the total amount of data to re-download, $16384c$ is the total amount of bytes to be re-downloaded and (13+29) are the bytes corresponding to a hash request and answer. In this case, t ranges from 10752 bytes (when $c = 0$) to $4 \times 1024^2 + 10752$ bytes (when $c = 256$).

Figure 4.4 on the next page shows the difference between the two approaches.



(a) A validation approach saves a lot of bandwidth.



(b) A semilogy chart emphasizes the difference between unconditional discard and selective re-download.

Figure 4.4: A comparison of BitTorrent and BitFountain with distributed validation.

Chapter 5

Conclusions and future developments

5.1 Conclusions

At the end of this work we have two products:

- ThunderStorm, a quick and fast Digital Fountain library which includes the cutting edge technology of Fountain Codes (Raptor codes). ThunderStorm can decode *every* code with a binary matrix in GF(2). The library is also extensible, allowing developers to include other kind of Fountain Codes very easily.

- BitFountain, a BitTorrent client which exchanges random blocks, encoded and decoded using Random or Raptor Fountain Codes. This client takes advantage on BitTorrent client because it can download more blocks in parallel from more peers than normal BitTorrent, leading to a faster reassembling of the piece. Also, when a client loose some incoming requests for blocks, BitFountain client does not reckon if it happens: as we said, the only matter is downloading a number of random blocks enough to invoke decoding process successfully. BitFountain also allows the implementation of a torrent distribution with UDP because we can recover the losses. Tribler developers recently developed a BitTorrent client that uses UDP: at application level they have implemented a timeout mechanism and ARQ like TCP, obtaining the same speed as using TCP (but they are using lighter sockets than UDP, bypassing OS limitation on TCP connections). μTorrent developers are moving in the same direction: in a new alpha version of the popular BitTorrent client, UDP has been made the default instead of TCP. Summarizing, there is a global interest in BitTorrent for real-time communication. In order to use resources efficiently and minimize latency, these applications should use BitTorrent over UDP rather than

TCP.

In addition, we have also studied a distributed validation protocol which is suitable for BitTorrent or BitFountain. We designed it to validate random blocks (since it is impossible to publish all the possible random blocks) and also to save bandwidth when some BitTorrent/BitFountain blocks are corrupted.

This is only the beginning of this project: first of all this thesis is the most complete and detailed specification of the BitTorrent protocol. Then, we have designed the characteristics of the future BitFountain client. BitTorrent developers came up with some interesting ideas [47] [48] to counter the weaknesses in BitTorrent protocol that we have identified, but their ideas do not have the power of erasure codes. We do believe that the development of this client has to be done in a quickly way following the design that we have described. Showing our results persuade developers to join this project.

5.2 Future developments for ThunderStorm

In the following paragraphs we are going to illustrate how we can improve ThunderStorm performances.

5.2.1 Reduce XOR operations

The Gaussian elimination algorithm implemented in BitFountain can be improved: in fact, when we receive a random block with its bitlist, we insert them in the decoding matrix without conditions; if we want to improve the speed of ThunderStorm, we must compare the already inserted bitlist with the inserting bitlist. In order to lessen the XOR operations, we must insert (or leave) the bitlist with less 1's in the decoding the matrix.

5.2.2 Support for tail block

In the current implementation, ThunderStorm require that all input blocks have the same size. However, in BitTorrent, there is an high probability that there is a tail piece (and consequently a tail block, see figure 5.1 on the facing page): BitFountain solves this matter by not encoding and decoding the last piece and use BitTorrent's normal transmission and receiving methods. In the future, ThunderStorm will support tail blocks by performing a sort of virtual padding: in case of tail block, ThunderStorm pads the last block and then creates the encoded random block. The padding is only virtual, which means that no padding data will be transferred over the wire (it is a waste of bandwidth). On receiver side, the decoder reassemble the piece using random blocks, and then removes the padding. Obviously, encoder and decoder must synchronize together on the size of the padding.

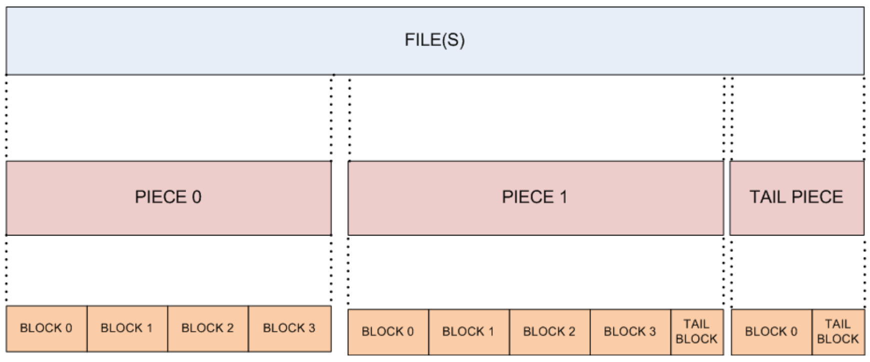


Figure 5.1: In BitTorrent, tail piece is the latest piece (and consequently we have a tail block). This occurs if the download size is not a multiple of the piece size.

5.2.3 Using ctypes for data intensive operations

ThunderStorm can be easily profiled using cProfile, a Python profiler written in C. As you can imagine, even from figure 3.9 on page 86, XOR operation is the main CPU intensive operation in ThunderStorm, written in Python. CPython is most-widely used implementation of Python and it written in C. To speed up XOR operation, we can use *ctypes*, that is an advanced FFI (Foreign Function Interface) package for Python. In other words, *ctypes* allows to use C from Python programs. In this way, ThunderStorm maintains its tight structure and use *ctypes* to XOR data using C; this will allow ThunderStorm to achieve lower encoding and decoding time.

5.2.4 Extend ThunderStorm

Since we designed ThunderStorm with evolution in mind, developers can integrate other types of Fountain Codes very easily. Because Raptor codes are the cutting edge technology in Fountain Codes, there is no reason to innovate at this moment. However, in the future there will be room for improving (adding the newest Fountain codes that will come up).

5.3 Future developments for BitFountain

In BitFountain there is large room for improvement: as we said, we have completely designed it; however, we developed a fraction of the total big picture that we had in mind.

5.3.1 UDP sockets

First of all, we must change BitFountain to use UDP sockets instead of TCP ones. The BitTorrent version that we have modified to build BitFountain is the 4.4.0, which uses twisted[1] to handle connections (previous versions of BitTorrent do not use twisted, using the Python sockets methods). Twisted is an event-driven network programming framework written in Python, which means that users of Twisted write short callbacks which are called by the framework. In this way it will be very straightforward to change BitFountain sockets from UDP to TCP.

On the other side, Tribler developers are designing a BitTorrent protocol which could exploit the currently unused potential of peers behind NAT firewall. This means building the TCP functionality in user-space on top of UDP with NAT firewall puncturing for HD streaming.

5.3.2 Endgame mode

As we said in chapter 3, BitFountain can be in endgame mode during the whole process of downloading. In fact, this will guarantee that all unchoked peers send random blocks to the client, leading to a faster download process. The client must not send `CANCEL` messages during download, because we do want "duplicates" (remember: in Digital Fountain, there are not duplicates, only useful blocks). We are only required to send `CANCEL` messages when a piece is reassembled correctly, to stop other peers from sending us additional (and only in this case useless) blocks for that piece. Lost requests do not have to be handled: as we said, in Digital Fountain it is not important if some blocks are lost. So, the most interesting thing of this approach is to estimate the total number of requests that we can send in parallel, basing on the estimated loss percentages: for example, if we need N blocks and we expect that only a fraction $\frac{1}{Z}$ of our block requests will be answered, we request for NZ random blocks to NZ peers. In this way we receive N random blocks and we reassemble the original information.

5.3.3 Digital Fountain on various data levels

We have implemented BitFountain by applying Digital Fountain encoding and decoding blocks within the same piece, obtaining random blocks. There are more alternative approaches that can be implemented:

1. Fountain codes applied to all data pieces, obtaining random pieces (see chapter 3).
2. Fountain codes applied to all data pieces, obtaining random pieces and then reiterate the encoding on all blocks within the same random piece, obtaining random blocks.
3. Fountain codes applied to all data block of chosen data pieces, obtaining random blocks. To generate a random block, we reiterate the encoding on the random blocks generated on the previous step.

[1]A framework for networked applications. `http://twistedmatrix.com/trac/`.

That being said, we could visualize the third approach as a cascading technique composed by the second approach followed by the first one; in the same way, the fourth approach can be seen as a cascading procedure composed by the first approach followed by a procedure that resembles the second technique.

5.3.4 Extend the peer wire protocol

We designed, in chapter 4, a distributed validation approach; we have also designed an extension to BitTorrent protocol to allow the exchanging of the validation messages. To save bandwidth even in BitTorrent and in BitFountain approach, and to also guarantee block (or piece) integrity, we need to implement this validation approach in BitFountain.

5.3.5 Development of error correction with Digital Fountain

Digital Fountain can also correct errors. In this case, the iterative subset decoding (see chapter 4) is a suitable solution to validate blocks (we correct blocks using blocks, in this way we eliminate corrupted blocks). As an example a simple error correction algorithm will be illustrated: hard-decision majority voting.

1. every parity bit send an estimate of the correct value to each connected data bit; for a given bit the correct value is the XOR of the parity and all others connected bits.

2. every data bit receive the estimate from each connected parity bit (and from the channel, the previous value) and takes an hard-decision using majority voting (the next value).

The loop is repeat until all checks are satisfied (no flipping in step 2) or until a maximum number of iterations is reached. A more sophisticated algorithm uses soft values e.g. floating point values ranging from -1 (certainly zero), 0 (maximum uncertainty), to $+1$ (certainly one).

Now, we can also erasure recovery with message passing: each parity bit form a group with its connected data bits. If one data bit is missing in the group, it can be recovered by XOR'ing the parity and remaining connected data bits (if more than one data bit is missing, no recovery can be done).

Appendix A

Summary (in Italian)

Capitolo 1: P2P e BitTorrent

In questo capitolo illustriamo le reti peer-to-peer (P2P), per cui si intende una rete di computer o qualsiasi rete informatica che non possiede nodi gerarchizzati come client o server fissi (clienti e serventi), ma un numero di nodi equivalenti (pari, in inglese peer appunto) che fungono sia da cliente che da servente verso altri nodi della rete. Questo modello di rete è l'antitesi dell'architettura client-server. Mediante questa configurazione qualsiasi nodo è in grado di avviare o completare una transazione. I nodi equivalenti possono differire nella configurazione locale, nella velocità di elaborazione, nella ampiezza di banda e nella quantità di dati memorizzati. L'esempio classico di P2P è la rete per la condivisione di file (file sharing), ma non solo.

Inoltre, introduciamo anche il protocollo BitTorrent, un protocollo peer-to-peer (P2P) che consente la distribuzione e la condivisione di file su Internet. A differenza dei tradizionali sistemi di file sharing, l'obiettivo di BitTorrent è di realizzare e fornire un sistema efficiente per distribuire lo stesso file verso il maggior numero di utenti disponibili sia che lo stiano prelevando (download) che inviando (upload). Si tratta quindi di un meccanismo per coordinare in automatico il lavoro di moltitudini di computer, ottenendo il miglior beneficio comune possibile. Dato che esiste scarsa documentazione (ufficiale o meno) che spieghi in dettaglio il protocollo BitTorrent, nella seconda parte del capitolo illustriamo nei minimi dettagli il funzionamento e il protocollo dei client BitTorrent.

Capitolo 2: Codici a fontana digitale

La trasmissione dell'infomazione su un canale disturbato da un rumore è un famoso problema nell'ambito della teoria dell'informazione. Essa propone come soluzione la cosiddetta codifica di canale, ovvero una rappresentazione espansa dell'informazione, cosicché la ridondanza introdotta mitighi la corruzione dei dati. All'interno di questo capitolo si tratteranno alcune codifiche di canale esistenti, illustrando pregi e difetti di ciascuna, fino ad introdurre il concetto di fontana digitale. Una fontana digitale è un codice quasi ottimale che, dato un set di K simboli di ingresso, produce uno stream potenzialmente infinito di simboli in uscita, ottenuti con la combinazione lineare di parte dei simboli di input. I simboli di input possono essere bit o sequenze di bit aventi lunghezza fissa e l'operazione di somma corrisponde allo XOR bit a bit. La scelta dei simboli di input che prendono parte allo XOR per la creazione del simbolo di uscita è tipicamente guidata da una distribuzione casuale, che determina quanti e quali simboli partecipano. Il trasmettitore divide il dato in K blocchi, che vengono messi in XOR fra di loro in vario modo per formare N pacchetti, inviati sul canale. Ad ogni istante il trasmettitore può generare pacchetti supplementari, semplicemente calcolando una nuova combinazione lineare fra i blocchi. All'altro capo, il ricevitore si mette in ascolto e colleziona K' pacchetti, poi lascia la trasmissione e decodifica i K blocchi del dato originale. Lungo il percorso che collega ogni client al server, si possono verificare errori e anomalie che portano alla perdita di pacchetti sempre diversi per ogni client. Grazie alle proprietà delle fontane digitali, questa perdita non è un problema e non necessita di un meccanismo di ritrasmissione, in quanto al client basta recuperare K' pacchetti qualsiasi per la ricostruzione del file. Questo meccanismo, per il quale i ricevitori non necessitano di un canale di feedback, è chiamato FEC (Forward Error Correction).

Capitolo 3: BitFountain

L'obiettivo di questa tesi è quello di integrare il meccanismo delle fontane digitali in un client BitTorrent. L'integrazione è possibile applicando la codifica (e la relativa decodifica) ai due livelli di gestione dei dati di BitTorrent: a livello di piece o a livello di block. Abbiamo scelto di applicare le fontane digitali XORando casualmente i blocchi di uno stesso pezzo. Continuiamo quindi con il design di un client, chiamato **BitFountain** (**Bit**Torrent + Digital **Fountain**) che prevede l'utilizzo di UDP anziché TCP (connessioni molto più leggere per il sistema operativo, non c'è ritrasmissione, non garantisce l'ordine di arrivo, prevede delle perdite – caratteristiche che sono ideali per l'applicazione di fontane digitali). Inoltre, il client può funzionare sempre in modalità endgame, richiedendo tutti i blocchi a tutti i peer connessi. Nel caso di BitTorrent "normale", questo porta al download di blocchi duplicati (e, di conseguenza, i client mandano messaggi `CANCEL` per evitare di scaricare blocchi duplicati). Nel caso di fontane digitali, invece, non esistono blocchi duplicati: tutti i blocchi sono utili. Quindi, in modalità endgame si massimizza il numero di blocchi che il client riceve, garantendo così un più rapido download rispetto ai client BitTorrent normali. Per realizzare BitFountain abbiamo scritto una libreria di codifica e decodifica a fontana digitale in Python. Tale libreria, chiamata ThunderStorm, permette di codificare e decodificare codici a fontana digitale del tipo Random e del tipo Raptor (lo stato dell'arte). Inoltre, abbiamo integrato tale libreria in un client BitTorrent 4.4.0 Mainline, producendo un prototipo iniziale che scambia blocchi casuali di fontana digitale codificati utilizzando la libreria ThunderStorm. Il client è compatibile con i client BitTorrent normali, scambiando blocchi normali quando si connette ad un client normale e scambiando blocchi codificati quando si connette ad un client Fountain capable.

Capitolo 4: Security

Lo scambio di blocchi random apre un'interessante problema di sicurezza: nel caso in cui i blocchi siano corrotti (volontariamente o meno), il decoder ricostruisce un oggetto diverso da quello di partenza. In questo modo un attaccante potrebbe impedire o sabotare il download semplicemente immettendo sulla rete dei blocchi falsi. Abbiamo quindi studiato diversi schemi di protezione contro questi attacchi, valutandone pregi e difetti di ognuno. La pubblicazione degli hash è impossibile: dato che si devono considerare tutte le possibili combinazioni di blocchi in input (ricordiamo che il processo di codifica sceglie casualmente dei blocchi in input e ne fa lo XOR), si dovrebbe pubblicare un numero elevato di hash (nell'ordine degli yottabyte nel caso peggiore). L'unica soluzione attendibile è un protocollo di validazione distribuito: una volta ricevuto un blocco, si richiede l'hash di tale blocco ad un peer diverso da quello da cui si è ricevuto il blocco, e scelto in modo totalmente casuale. Se si riceve un hash che conferma il blocco, si può assumere che il blocco sia valido e l'algoritmo termina; altrimenti si ripete la richiesta ad un altro peer fino ad arrivare ad un massimo di hash ricevuti che non confermano il pezzo. In questo caso il blocco è corrotto e va riscaricato. La verifica può essere svolta in seriale o in parallelo; dato che usiamo un hash crittografico, è molto difficile che un peer malintenzionato possa costruire ad-hoc un blocco falso che abbia lo stesso hash di un blocco valido (e quindi è molto difficile che un peer buono validi un blocco non valido). L'algoritmo può essere invocato sia ad ogni ricezione di un blocco oppure soltanto nei casi in cui il pezzo ricostruito dà luogo ad un failed hash check. Inoltre, l'algoritmo di validazione distribuita che proponiamo è adatto anche ai client BitTorrent, che scartano tutti i blocchi nel caso in cui il pezzo ricostruito non passi il processo di hash check. In entrambi i casi (BitFountain e BitTorrent normale), si riscaricano solo i pezzi non validi: questo si traduce in un migliore utilizzo della banda disponibile.

Capitolo 5: Conclusioni e sviluppi futuri

Molto è stato fatto e molto rimane da fare. Abbiamo gettato le basi per il futuro sviluppo di questo progetto: innanzitutto, abbiamo creato una libreria di codifica e decodifica che include lo stato dell'arte per quanto riguarda i codici a fontana digitale. Poi, abbiamo iniziato a delineare il client BitFountain integrando tale libreria all'interno di un client torrent.

La libreria è stata creata tenendo conto degli sviluppi futuri: è molto facile, infatti, includere un nuovo tipo di codifica e decodifica. Inoltre, la libreria può essere resa più veloce integrando opportune ottimizzazioni [39] e utilizzando codice C per le operazioni di XOR.

Per quanto riguarda BitFountain, invece, si devono implementare le idee che abbiamo discusso nei capitoli precedenti, come ad esempio l'utilizzo di UDP anziché TCP, l'utilizzo della modalità endgame, l'applicazione delle fontane digitali non solo ai blocchi ma anche ai pezzi, e così via.

Bibliography

[1] F. Benedetto, G. Giunta. *Elementi di Reti per Telecomunicazioni, (Parte III), Controllo degli Errori e Internet Protocol.* Corso di Telecomunicazioni, Università Roma Tre. 2004

[2] A. Bharambe, C. Herley, V. Padmanabhan. *Analyzing and Improving BitTorrent Performance.* Microsoft Research. 2005

[3] J. Byers, M. Luby, M. Mitzenmacher, A. Rege. *A Digital Fountain Approach to Reliable Distribution of Bulk Data.* Harvard University, Division of Engineering and Applied Sciences. 2002

[4] B. Cohen. *Incentives Build Robustness in BitTorrent.* 2003

[5] L. Cottrell, W. Matthews, C. Logg. *Tutorial on Internet Monitoring & PingER at SLAC.* `http://www.slac.stanford.edu/comp/net/wan-mon/tutorial.html`. 2007

[6] M. Cunche, V. Roca. *Adding Integrity Verification Capabilities to the LDPC-Staircase Erasure Correction Codes.* I.N.R.I.A. Rapport de recherce. 2007

[7] D. Erman, D. Ilie, A. Popescu. *BitTorrent Session Characteristics and Models.* School of Engineering, Bleking Institute of Technology, Sweden. 2007

[8] C. Gkantsidis, T. Karagiannis, P. Rodriguez, M. Vojnovic. *Planet Scale Software Updates.* Technical Report. Microsoft Research. 2007

[9] J. Kleinberg. *The small world phenomenon: an algorithmic perspective.* Cornell Computer Science Technical Report 99-1776. 1999

[10] A. Legout. *Understanding BitTorrent: An Experimental Perspective.* Technical Report, I.N.R.I.A. 2005

[11] S. Hart. *Robert Aumann's Game and Economic Theory.* Stockholm School of Economics. 2005

[12] A. Harwood, T. Jacobs. *Localhost: a browsable Peer to Peer file sharing system.* 2005

[13] E. Hyytia, T. Tirronen, J. Virtamo. *Optimizing the Degree Distribution of LT Codes with an Importance Sampling Approach.* Networking Laboratory, Helsinki University of Technology Finland. 2006

[14] L. Lamport, R. Shostak, M. Pease. *The Byzantine Generals Problem.* ACM Trans. Programming Languages and Systems 4. 1982

[15] E. Lazowska. *Quantitative system performance - Computer system analysis using queuing network models.* Prentice-hall Inc. 1984

[16] B. Leiner. *LDPC codes, a brief tutorial.* Department of Electrical and Computer Engineering, University of Victoria, Canada. 2005

[17] Y. Liu. *On the minimum delay peer-to-peer video streaming: how real-time can it be?.* ACM. 2007

[18] T. Locher, P. Moor, S. Schmid, R. Wattenhofer. *Free Riding in BitTorrent is Cheap.* Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Zurich, Switzerland. 2006

[19] M. Luby. *LT Codes.* Digital Fountain Inc. 2002

[20] M. Luby, L. Vicisano. *RFC 3695: Compact Forward Error Correction (FEC) Schemes.* The Internet Society. 2004

[21] M. Luby, L. Vicisano, J. Gemmel, L. Rizzo, M. Handley, J. Crowcroft. *RFC 3452: Forward Error Correction (FEC) Building Block.* The Internet Society. 2002

[22] M. Luby, L. Vicisano, J. Gemmel, L. Rizzo, M. Handley, J. Crowcroft. *RFC 3453: The Use of Forward Error Correction (FEC) in Reliable Multicast.* The Internet Society. 2002

[23] D. MacKay. *Information Theory, Inference, Decoding Algorithms.* Cambridge University Press. 2003

[24] M. Mitzenmacher. *Digital Fountains: A survey and Look Forward.* Harvard University, Division of Enineering and Applied Sciences. 2004

[25] J. Mol, A. Bakker, J. Pouwelse, D. Epema, H. Sips. *The Design and Deployment of a BitTorrent Live Video Streaming Solution.*

Department of Computer Science, Delft University of Technology. 2008

[26] A. Perrig, R. Canetti, J. Tygar, D. Song. *The TESLA Broadcast Authentication Protocol.* Network and Distributed System Security Symposium. NDSS. 2002

[27] M. Piatek, T. Isdal, T. Anderson et al. *Do incentives build robustness in BitTorrent?.* Dept. of Computer Science and Engineering, Univ. of Washington. 2007

[28] R. Riva. *Studio ed implementazione di codici a fontana digitale LDPC.* Politecnico di Milano. 2005

[29] V. Roca, Z. Khallouf, J. Laboure. *Design, evaluation and comparison of four large block FEC codes, LDPC, LDGM, LDGM Staircase and LDGM Triangle, plus a Reed-Solomon small block FEC codec.* I.N.R.I.A. Rapport de recherce. 2003

[30] V. Roca, Z. Khallouf, J. Laboure. *Design and evaluation of a low density generator matrix (LDGM) large block FEC codec.* I.N.R.I.A. Rapport de recherce. 2003

[31] J. Rosenberg, H. Schulzrinne. *RFC 2733: an RTP Payload Format for Generic Forward Error Correction, Standards Track.* The Internet Society. 1999

[32] A. Rota. *DropBox, CloudStorm, SiNBD: Memorizzazione distribuita, Dispositivi Virtuali e Fontane Digitali.* Università di Bergamo. 2008

[33] A. Shokrollahi. *Raptor codes.* Laboratoire d'algorithmique, Ecole Polytechnique Federale de Lausanne, Switzerland. 2003

[34] E. Sirer, A. Slivkins, B. Wong. *Approximate Matching for Peer-to-Peer Overlays with Cubit.* Dept. of Computer Science, Cornell University. 2008

[35] E. Sirer, Y. Vigfusson, B. Wong. *Hyperspaces for Object Clustering and Approximate Matching in Peer-to-Peer Overlays.* Dept. of Computer Science, Cornell University. 2007

[36] C. Shannon. *A mathematical theory of communication.* Bell System Technical Journal. 1948

[37] R. Tanner. *A recursive approach to low complexity codes.* IEEE Trans. Inform. Theory IT-27. 1981

[38] T. Turocy, B. von Stengel. *Game Theory.* CDAM Research Report LSE. 2001

[39] R. Villa. *Digital Fountain Codes for Peer-to-Peer Networks.* Politecnico di Milano. 2007

[40] *Liebel-Lab @ Karlsruhe institute of technology KIT.* `http://liebel.fzk.de/`. 2008

[41] *Presentation of Nobel Prize in Economic Sciences 2005 Robert Aumann's and Thomas Schelling's Contributions to Game Theory: Analyses of Conflict and Cooperation.* Royal Swedish Academy of Sciences. 2005

[42] Internet Engineering Task Force. *Symmetric NAT Traversal using STUN.* `http://www.cs.cornell.edu/projects/stunt/draft-takeda-symmetric-nat-traversal-00.txt`. 2003

[43] *Ipoque Internet Study 2007.* 2007 `http://www.ipoque.com/resources/internet-studies/internet-study-2007`. 2007

[44] *Sandvine Releases Global Internet Traffic Trends Report.* `http://www.sandvine.com/news/pr_detail.asp?ID=203`. 2008

[45] *Gaussian elimination.* Wikipedia, The Free Encyclopedia. 2008 `http://en.wikipedia.org/wiki/Gaussian_elimination`

[46] BitTorrent developers community. *Comcast Throttles BitTorrent Traffic, Seeding Impossible.* `http://forum.bittorrent.org/viewtopic.php?id=7`. 2008

[47] BitTorrent developers community. *XOR pieces.* `http://forum.bittorrent.org/viewtopic.php?pid=443`. 2008

[48] BitTorrent developers community. *Sub-piece hashing.* `http://forum.bittorrent.org/viewtopic.php?id=72`. 2008