A report on the decentralized web

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Specialization in Computer Systems

presented by
Eric Botter

under the supervision of
Prof. Fernando Pedone
co-supervised by
Leandro Pacheco De Sousa

June 2018

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Eric Botter
Lugano, 20 June 2018

Someone said . . .

Someone

Abstract

The World Wide Web is probably the most popular and used service of the Internet, sometimes even confused with the Internet itself. But the Web has one fundamental problem: it is based on a client-server model, which is not ideal considering the importance of Web services that we use everyday.

What if some Internet Service Provider decides to block access to a competitor's service, so that it can promote its own? What if some organized attacker wants to deny access to a stronghold of open source software, such as GitHub? What if some government decides to block an important knowledge repository, such as Wikipedia? What if all of these already happened?

The invention of the blockchain and its popularization through projects like Bitcoin and Ethereum brought renewed interest towards fully decentralized and trustless solutions, and this has also impacted the Web. In this thesis, we present and analyze some of the currently implemented and active projects which aim is to move the Web from its centralized client-server environment to a decentralized trustless system.

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras

ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Contents

Chapter 1

Introduction

The World Wide Web is probably the most popular and used service of the Internet, sometimes even confused with the Internet itself. It is very common nowadays to access the WWW (or most commonly known as simply "the Web") and browse websites from many platforms, from the typical desktop computer to the modern smartphone.

Let us define the scenario in which the Web lives. It is based on a client-server architecture, where Web servers provide objects (such as documents, images or files in general) to clients that request and display them, called *user agents* (e.g. Web browsers).

In the Web, documents and objects are identified by a Uniform Resource Locator (URL), whose most important component is the domain name: it is a human-readable label that identifies a device within the Internet. A domain name is composed by sequences of letters and symbols, separated by each other with dots. This separation is needed by the hierarchical structure of domain names, but we won't delve into the details of the Domain Name System here: we will talk more about it when we discuss naming, in Chapter 4.

To access a website, the client has to know the domain name associated to that website. This is usually provided by the user or by services such as search engines. The domain name is resolved to an IP address by using the Domain Name System. Once obtained, the client opens a TCP connection towards that address on port 80, and starts exchanging messages using the HyperText Transfer Protocol (HTTP).

HTTP is a client-server, request-response protocol. Clients specify the details of the needed resource in the request and the server replies with the content or an error status if something went wrong (e.g. 404 Not Found). We won't explore HTTP as none of the projects that we will see rely on HTTP or any of its properties.

There are different ways to setup a website. A content creator can either setup a custom server and upload a website there, or it can rent a server (either physical or virtual) from an existing provider.

1.1 Problems in the current web

The main problem in the current Web is vulnerability to **censorship**. Since we have a direct relationship from domain names to websites (or from IP addresses to websites), it is relatively easy for powerful parties (including governments and ISPs) to block communications from users

to a certain service. The main attacks that can be used to prevent communication towards a website are:

- Denial of Service (DoS): a large volume of requests is sent towards the targeted server, which quickly runs out of available resources (such as bandwidth, simultaneously open connections, memory or CPU). Requests can be sent from a single device, but in current days requests are typically sent from multiple sources, in order to both increase the volume of traffic and make it difficult to identify and stop the origin of the attack: this is known as Distributed Denial of Service (DDoS).

- IP address blocking: packets towards a given address or address range are blocked. This attack can be enacted by routers that exchange packets regarding the targeted IP address, which can interrupt forwarding of said packets thus preventing any sort of communication, making the server effectively disconnected from the Internet.

- DNS hijacking: by altering DNS resolutions, the domain of the targeted website can either be deleted or edited to make it refer to another IP address, thus preventing access to the original content. This attack can be carried out by both the owners of the DNS resolver (by directly editing their records), or by third parties through an attack called DNS cache poisoning: an attacker pretending to be a valid name server intercepts DNS requests from other name servers and provides fake responses to alter the address of the targeted domain, also setting a high time-to-live so that the redirection is active for as long as possible. Another vector for DNS hijacking, though unrelated to DNS itself, is to remotely edit the configuration of typical home routers through known vulnerabilities, changing the DNS resolver to a malicious one.

We also have a problem of **trust**. When you access a website, there is no guarantee that the data you received is from the content creator, because HTTP is vulnerable to man-in-the-middle attacks. There is no mechanism to verify the authenticity of the transmitted data and the protocol does not use encryption, so anyone can forge a valid HTTP communication (even based on an ongoing one) and send it through the wires. We expand on this in Chapter 2.

HTTPS resolves this issue by asymmetrically encrypting the communication channel, and authenticating the data that is sent, but the current trust system (X.509) is based on certificate authorities and is considered weak, which might allow for identity theft.

Another important issue is **privacy** and handling of personal information: with the current scenario, whenever you connect to a website, that website privately stores data about you. This data can be either automatically collected from user interactions, or can be provided directly by the user: consider, as an example, a social network, where users provide personal information such as their generalities, and the website collects data such as post interactions, number and timestamps of logins, and so on. This effectively moves ownership of the data from the user to the company. Data that intrinsically belongs to the user (especially personal information such as name, address and phone number) are stored privately into the company server, and the user has limited control over it, since the only possible actions on the data are the ones defined by the company or required by law.

1.2   Thesis structure

This thesis has been structured as follows. The first chapter describes the Web in its current state and defines the open issues in its regard. The second chapter introduces some common

definitions and describes the models used to analyze the Web and any of the projects that will subsequently be explored; we also define the characteristics of the Distributed Web. The following three chapters describe and analyze some projects that focus on distributing storage, distributed naming and offer a complete implementation of a distributed web, respectively. The sixth chapter will elaborate on some upcoming projects that have the possibility to impact the distributed web scenario. We will conclude the thesis in the seventh chapter.

1.3   Thesis goals

The goal of this thesis is to present various projects that empower real-case implementations of the distributed Web, as we explain it above. We also aim to analyze notable features and possible defects of each system we present, with particular interest to the aspects that most closely relate to the distributed Web. This thesis is also meant to serve as a reference for seminars, lectures and possibly courses related to the distributed web.

Chapter 2

Models and Definitions

We have to rethink the Web if we want to move it to a decentralized environment. The current Web is a centralized system: each website is owned by a party that we'll define as *content creator*. The content creator owns the website and is responsible for distributing its content, either by using a self-owned and maintained web server or by publishing it to a dedicated service, known as *web hosting* service provider (there are too many services currently online to present a somewhat accurate list of examples here). When using *web server*, we will always refer to both these options, since in both cases there is always a web server that serves the website, whether it's owned by the content provider or by a company. Although it's not required, the content creator usually also obtains a domain name to associate with the website.

This system is centralized because the website is accessible only through the web server. If obtained, the domain name will always direct towards that server (even if it changes its IP address, since that's one of the main purposes of DNS).

Let us introduce a very important concept in distributed systems: **failure**, and its related models. We introduce it now to highlight the difference between a centralized environment and a decentralized (or distributed) one.

The failure model in which we could place the Web is a **stopping failure model**. In this model, a process can **fail**, i.e. it stops functioning as intended, and the way a process fails is by stopping, or *halting*. Once a process has halted, it remains in that state. Other processes are able to detect this failure, and the most common way that this is done is by giving a maximum response time to message exchanges with that process: if we expect a communication which does not happen within a time limit, we consider that process as failed (this is called detection by *timeout*).

This means that we know that each entity will behave as expected, and in case of failure it will not function at all. In other words, we consider that entity as **trusted**, as it will never give a response which is not conforming to the protocol.

This model allows Web clients (browsers) and content creators to make certain assumptions on the behavior of web servers and other components:

- Web clients assume that the content they request is returned without any modification;

- It is assumed that the channel through which the information is sent does not modify that data;

- Content creators assume that the website that they create is stored in the web server (and distributed) without any modification.

In case of failure of the Web server, a browser will try to communicate with it, and after some time without receiving a response it will give up and show to the user an error message, of which an example can be found in Figure 2.1.

One could argue that this model does not accurately represent the reality. For example, if a malicious third party takes over the communication channel, gaining complete control over each byte that it transports[1], it can interfere and change the data that is transmitted. This is transparent to both the clients and the web server, since there is no mechanism in place to ensure that the data is *integral*[2] – and rightfully so, given that the communication channel is assumed to not alter data arbitrarily. This is known as a *man-in-the-middle* attack.

To allow this scenario in our model we need to weaken it, by removing assumptions on what the communication channel will do during failure. We now allow arbitrary behavior, which means that processes under this weaker model can produce any kind of message, whether it complies to the protocol or not: they can lie, omit information or do just about anything, which also includes stopping and behaving properly. This model is the **Byzantine failure model**. This is particularly useful to model malicious intentions: since failed entities can perform arbitrary actions, it is particularly important to consider situations that can cause the most damage to the system. Once we protect against these actions (and all possible other scenarios), we are sure to protect the system from any attacker that takes over the entities that we consider to be Byzantine.

Let's now place the communication channel in this Byzantine failure model. HTTPS, the more secure Web protocol, is built around this weaker scenario. It introduces encryption and authentication of transmitted data, which allows it to be transferred over *non-trusted* communication channels. If the data is altered by a malicious third party during transmission, the client or the web server can detect it and react accordingly.

But what if we extend the Byzantine model to other entities? For example, if a content creator publishes a website on a web hosting service, that platform is technically able to delete or alter any file of that website, effectively sending to clients different information than the one intended by the author, and the clients would not notice this difference. HTTPS would not be able to protect against this attack, since it only protects the communication channel and not the website files themselves, which are encrypted by the web server only when transmitting them to a client: to do this, the web server has to access the files directly, and can therefore do whatever possible action to them, including editing and deleting them. Not only that: the web server is also able to send whatever content it desires (or refuse to serve content at all), since the client has no way of verifying that the content is as intended by its author.

What would we achieve if we protect the data itself from Byzantine attacks? If the data can be distributed by a non-trusted entity, then we don't need to rely on some specific and trusted system: *anyone* can distribute it.

This effectively enables a **distributed Web**, where content is shared by a network of web servers that anyone can provide. We don't need to trust some hosting service provider with the website data, we can distribute our data to any of them and we will be sure that the information is going to be transmitted safely without modifications.

---

[1]An example of such a third party are Internet Service Providers.

[2]TCP provides data integrity against transmission errors, which are in the order of few bits per kilobyte, but this is too weak in this scenario, where the entire TCP packet can be rewritten to appear unaltered.

Let's see how this scenario can resolve the three problems described in Section 1.1:

- **Censorship**: to allow web servers to expose Byzantine behavior means that any one of them can refuse to serve content. But note that we can't assume that every single web server fails with Byzantine consequences, just like we don't assume that in the current Web every single server will crash when contacted. We have to guarantee that some percentage of the servers will behave properly, without failing.
  This is why we need a **network** of web servers and not just one: that one server not only is exposed to all of the attacks mentioned in Section 1.1, but can also fail by refusing to serve content; if, instead of one, we have multiple web server (of which we are guaranteed that a percentage behaves properly), we know that if some web server fails (either on its own or by means of an attack) the rest of the network can still operate.

- **Trust**: by distrusting the web servers, we have made impossible for them to successfully send data different from what the content creator intended. Clients have a way to detect information manipulation and can discard such manipulated data.

- **Privacy** and handling of personal information: in a distributed environment, we no longer have websites that can privately store data about their users. This is because every website now is hosted on multiple servers: if they want to provide some form of consistency, they have to share or synchronize that data in some way. This would imply that the data is no longer private, but we cannot have public personal data available to multiple servers, especially when some of them have malicious intents.
  Among the different possibilities, we mention two options here: either the clients (and the clients only) store the private data, so that it is not shared in the network, or the data is uploaded in the network but in an encrypted way, so that only its owner can access it. We will see how different projects tackle this problem, and whether they solve it.

Many projects claim to implement the distributed Web. Every one of them provides a specific strength or focuses on a particular issue instead of building a complete, all-round system. Unfortunately, such protocols are not well-known to the general public, and consequently are not widespread enough to be either standardized or integrated into modern state-of-the-art browsers. But most importantly, creating such a protocol is very difficult.

Keep in mind that many of these projects do not define the model around which their platform is built. They mostly define the possible attacks and the related countermeasures they adopt. We will try to infer a model of the environment whenever we consider appropriate to do so, while trying stay as close to the project specifications as possible.
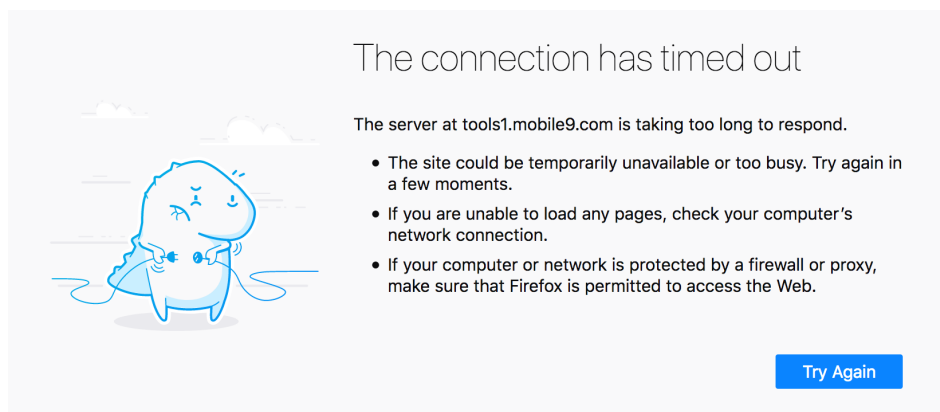
Figure 2.1. An error message for a connection timeout with a Web server given by Firefox 60.

Chapter 3

Decentralizing Storage

The first and most fundamental challenge in building the distributed Web is how to store the files that compose websites.

In the current Web, this is not generally a problem: you can either setup your own personal web server, exposed to the internet, that contains the website files, or you can upload those files to a web hosting service provider in order to serve them for you. But now we have introduced a network of untrusted servers that are not necessarily going to distribute your files as you intended, if at all. How do we make sure that the files are still getting to the clients? How can clients be sure that they received the files exactly as their author has written them? To answer these two questions, each project has to implement two features, respectively.

The first one is an **incentive scheme**, i.e. a way to encourage nodes in a network to share data. If the incentive is strong enough, there will be more people that genuinely participate in the system, thus reducing the relative amount of intentional Byzantine behavior. Furthermore, if there is no incentive to share data, nodes will just obtain what they need and then leave the network, without contributing to other node's downloads, thus reducing the overall performance of the system, possibly even rendering it useless if this behavior is intentionally brought to the extreme (in other words, an attack is performed).

The second one is **data verification**: each system must provide a way to verify the integrity of the data that is transmitted. This is needed because everyone in the system can fail and transmit arbitrary data: who receives the data must be able to verify the received information, in order to detect and possibly discard it (maybe going as far as taking action w.r.t. the failed sender, for example by blocking communications). It usually involves cryptographic hashing, sometimes combined with asymmetric cryptography.

Another key component is a distributed storage system, but in general of any peer-to-peer network, is **peer discovery**, i.e. a mechanism through which a client can obtain a list of currently active peers that can be contacted. This is because on the Internet, it is impossible to find which devices are offering which services (especially if these devices keep moving and disconnecting all the time), without either knowing them directly, or looking them up using a dedicated directory that lists and keeps track of such devices. This directory can be either centralized or decentralized, and comparing the two options leads to the same advantages and disadvantages that we discussed previously. It would make sense to prefer a decentralized directory to a centralized one, but a completely distributed directory cannot exist, since it also needs to be discovered: a compromise has to be made. Often, projects with a distributed direc-

tory have a different process just to make the directory known to a client: this process is often referred to as *peer discovery bootstrap*.

3.1 BitTorrent

BitTorrent is probably the most known and used peer-to-peer file-sharing protocol. We take a look at BitTorrent both because of its popularity and because it is at the basis of ZeroNet, another project discussed in Section 5.1.

3.1.1 Protocol overview

Let us start with some definitions related to the BitTorrent protocol. A collection of files is known as a **torrent**, and to make distribution easier, each file is split into **blocks** of equal size. A torrent is described in a dedicated file called, unsurprisingly, **torrent file** (they are also known as *metainfo files*), which contains a list of the files contained in that torrent and the cryptographic hash of each block (using the SHA-1 algorithm), plus some metadata, such as the name of the torrent, the block size, etc.

A BitTorrent client connected to and participating in the network is called a **peer**. Peers communicate with each other when they are aware of the same torrent: that group of peers is called a **swarm**. When a peer has all the parts of a torrent and is sharing them with the rest of network, it is called a **seeder**, and the operation of sharing data is called *seeding*.

In BitTorrent, torrents are distributed out-of-band. It is assumed that when a peer obtains a torrent file, that file is legitimate and as intended by the torrent creator. Usually, torrent files can be obtained from the Web, often from dedicated websites and forums where users can upload their own torrent files: the documentation of BitTorrent only mentions the Web as a way to obtain torrent files.

To perform peer discovery, BitTorrent relies on **trackers**, dedicated servers that keep track of swarms for different torrents. Each torrent file lists the trackers that peers should contact (by announcing themselves) for that particular torrent. Although different torrents can rely on different (and multiple) trackers, this is a centralized peer discovery system: it is often the case that few trackers (in the order of tens) track the majority of torrents. If these trackers are shutdown or blocked, peers will have a hard time discovering each other. To counter this, BitTorrent later integrated two decentralized discovery mechanisms: DHT (which we cover in Appendix B), and **Peer Exchange** (or PEX) that allows peers to trade information about their common swarm, so that each client can discover more peers, leading to a better connected swarm. Note that peers cannot join a swarm only through PEX: they have to first contact either a tracker or the DHT.

BitTorrent does not define how to bootstrap the connection to the DHT, leaving it as an implementation detail. Different clients employ different techniques, among which we can find the following:

- Keep a long-lived cache of peers encountered: when the client is started, it tries to contact the peers in his cache to enter the DHT;

- Contact a known IP address/domain name that offers a dedicated DHT bootstrapping service;

- Retrieve an initial list of peers from a tracker, then access the DHT through them;

- Torrent files can contain information about nodes of the DHT or peers in the BitTorrent network.

To better describe the functionalities and mechanisms of BitTorrent, let's explore two fundamental scenarios: uploading a torrent into the network and downloading a torrent from the network.

To upload a torrent into the network, a user has to collect a set of files that will compose the torrent. By using a client that supports this functionality, the user can create a corresponding torrent file, which can then be distributed by the user on their preferred platform. The user's client now is aware of the torrent and has all its files: it is therefore seeding the torrent to other users.

To download a torrent from the network, a user has to obtain the torrent file first. The file is then passed to the client that will start to contact the swarm (through trackers, DHT or whichever system the torrent file specifies) and gather information about which peers have which data. It will then start downloading blocks from each peer, and will become a seeder when the download is complete.

BitTorrent's **incentive scheme** is well explained and analyzed by Cohen [2003], the creator of BitTorrent, and we will just summarize it here. It is based on a *tit-for-tat* algorithm, and uses a mechanism known as **choking** to select the best peers for collaboration. Choking is refusal from a peer to share its blocks with another peer, who is said to be *choked*; *unchoking* is the opposite operation.

BitTorrent peers usually do not share data with all known downloading peers to reduce overhead of communications and best utilize the available bandwidth. Peers will start sending blocks to some other peer in the swarm, hoping that they return the favor and start sending data back. If the favor is not returned, the other peer is choked, and some other peer is chosen instead; otherwise, the peer will increase its bandwidth towards the other peer, and those peers will continue to share data between each other. This is done until a limit of simultaneously connected peers is reached, usually four but this is user-configurable. Every now and then, a peer will unchoke some random peer, hoping that a benevolent uploader is found: this is called an *optimistic unchoke*, and it is done by replacing the worst-performing peer, which becomes choked. This allows to improve performance if the current transfers do not maximize a peer's bandwidth, but it can also degrade it if the new peer does not perform as good as the replaced one: this is why it is said to be *optimistic*.

BitTorrent **data verification** system is based on cryptographic hashing, and is pretty straightforward: each torrent file contains the SHA-1 hash of every block in the torrent, so that peers can verify the integrity of each block upon downloading it.

3.1.2   Analysis

Let's try to infer the model in which BitTorrent is set. We have two main entities specific to the BitTorrent protocol: peers and trackers.

We can safely place peers in a Byzantine failure model. This is because BitTorrent implements a data verification mechanism to ensure that the blocks received by the network are valid, by computing their hash and matching it with the one reported in the torrent file. It also discourages malicious activity by implementing a *tit-for-tat*-like incentive scheme, of which we'll talk about later.

Identifying the failure model for the trackers is not so easy. This is because there is no

method in place to ensure data validity: technically, a rouge tracker can omit data during communication with a peer or craft arbitrary responses, and a BitTorrent client would not notice this misbehavior. But what kind of damage can a Byzantine response from a tracker possibly cause? The simplest one is refusing to communicate, but this can also happen in a stopping failure model: a solution is to use data from another tracker (also note that clients can detect a missing response through timeout). A similar but more elaborate attack would be to provide a valid response, but without most if not all peers of the requested swarm: this is practically a denial of service attack, because a client would not be able to perform a download since it can't find any other peer to get the data from. To counter this attack, a client would have to lookup multiple trackers: it is often the case that a torrent file has references to more than one tracker, besides, a client can contact additional trackers than the ones specified in the torrent file, either automatically (a list of trackers that is added to torrents when starting the download process) or by user interaction. Lastly, a malicious tracker can include in a response fake peers that don't actually exist, that are currently unavailable or that only exhibit Byzantine behavior. Also this kind of attack falls under denial of service, since a client has to waste time contacting all these fake peers that will never respond correctly, if at all: a solution to this is, again, to combine data from multiple trackers.

Let's not forget that BitTorrent also has decentralized peer discovery methods, including a DHT. These systems, if available, will always be used alongside the tracker-based system, and since the DHT runs in a Byzantine failure model, we can consider it more reliable and trustworthy. Therefore, we can also place the trackers in a Byzantine model, since we won't be losing functionalities in doing so, although we might suffer from a performance perspective.

But there's still one element that needs consideration: the source of torrent files. The BitTorrent documentation (specifically in BEP 3) explicitly specifies that torrent files must be served over the Web, but this is not always the case, and it surely isn't enforced by BitTorrent clients: as long as they receive a torrent file, they will start to download the corresponding torrent. It is pretty clear that this source must be trusted, otherwise we would have an invalid torrent definition and the client would download data that is not what the torrent file or its source might suggest to be.

So far the weakest link in this model is the source of the torrents. A user must find a reputable and trustworthy source from which torrents can be obtained. This is particularly an issue because of the controversial (and often illegal) content that can be obtained through the BitTorrent network: for example, malicious agents might setup websites which offer counterfeit software, while they actually distribute malware, or government agencies might take control of a popular torrent distribution website to track their users and alter or remove the torrent files that are offered (potentially affecting legitimate torrents).

But this is not the only issue in the network. Its incentive scheme has been thoroughly analyzed by the academic community and many publications demonstrate extensive research on the subject (we'll cite Zghaibeh and Harmantzis [2008]), outlining defects in the scheme and the possibility of downloading content without contributing to the network: this is known as **free riding**. Locher et al. [2006] describe a BitTorrent client named BitThief, that exploits optimistic unchokes to successfully download torrents while avoiding to share any resource, by always presenting itself as a new peer. Despite the presence of such behavior, the network is still functional and largely used today. The BitTorrent community has come up with *etiquette* rules to maximize benign usage: for example, it is considered good practice to seed a torrent after downloading it, at least until the same amount of downloaded data has been uploaded (i.e. until a *seed ratio* of 1 is reached: most clients will compute this value for the users).

3.2   IPFS

The InterPlanetary FileSystem (IPFS for short) is a decentralized file system based on content-addressing coupled with a hypermedia distribution protocol. It has become popular as the *Permanent Web*, as one of the goals of the project is "to make the web faster, safer, and more open" and "where links do not die".

3.2.1   Protocol Description

IPFS is a peer-to-peer system, and each peer is called a **node**: all nodes share the same roles and no node is privileged. Each node keeps a set **objects**, that can represent files or other kinds of data: objects are stored in **local storage**, i.e. some external system on which raw data can be stored or retrieved. This can range from a hard disk drive to some dedicated RAM, but it can also be some more complex, possibly networked system as well.

On initialization, a node generates an asymmetric key pair, then it computes the cryptographic hash of their public key: that hash becomes the node's **identifier** within the network.

In IPFS there is one global (rather, *interplanetary*) space where all objects are stored, unlike in BitTorrent where each torrent is completely separated from the others. Each object is therefore uniquely identifiable, and the system to generate such an identifier is cryptographic hashing. This is known as *content-addressing*: the *address* that identifies and allows to retrieve some data is directly derivable from the data itself. This method enables the file space to be modeled as a *Merkle DAG*, i.e. a directed acyclic graph where the edges are implemented as cryptographic hashes of the content of the node they are pointing at.

Objects can be of four types:

- **blocks**, or *blobs* (the two terms are used interchangeably in the whitepaper): they carry a variable amount of data, of which IPFS does not need to understand the format, which makes it suitable for any kind of application;

- **lists**: collections of blocks or other lists, usually used to represent large or deduplicated files (i.e. files that have been divided in multiple blobs because of data repetition within the file itself or with other blocks already in IPFS, so that such repeated data is stored only once);

- **trees**: collections of blocks, lists, trees or commits, but with a name for each object, unlike in lists where only the hashes are present;

- **commits**: snapshots in the history of some other object, used to enable a form of version control over objects; this allows IPFS to be compatible with Git (in the sense that Git repositories can be modeled as IPFS objects while IPFS trees can be exposed as Git repositories).

Each object is identified by a UNIX-style path. Each path starts with `/ipfs`, but this path alone does not refer to a valid object, as in IPFS there is no "root" object, and it must be followed by an hash which identifies the object (and can validate its contents). If the object contains named links (such as trees), then the path can be extended with the names of the child objects to reach these items. Note that such child objects can also be accessed directly by their hash.

This path-style addressing of objects allows IPFS to be mounted through a FUSE interface, or to directly map to other existing path-style identification mechanisms, such as URLs used in the Web.

In IPFS, peer discovery is carried out entirely by a DHT: nodes will publish which objects they have along with a reference to themselves (their identifying information) on the DHT and other nodes can retrieve this information when they need to obtain those items. There is no centralized system like BitTorrent's trackers in IPFS. At the time of writing, the DHT is bootstrapped by accessing a list of known nodes, hard-coded in the IPFS source files, as found in the reference implementation in Go. Users of this client can configure the nodes they bootstrap from by using a dedicated command (`ipfs bootstrap`). As commented in the source code, this is a potential security problem, but we will discuss this later.

To exchange blocks between peers, IPFS introduced a custom protocol named BitSwap, inspired by BitTorrent. Each node has some blocks in local storage, and needs blocks from other peers (for example, they need an HTML page that a browser has requested to the IPFS client). To obtain such needed blocks, a node would search for the corresponding hashes in the DHT and discover a set of nodes that have it: note that the requesting node already knows that hash because it's the locator (or the address) of that block (to stay on the previous example, the hash would be part of the URL that the browser was requested to open). Other nodes can accept or deny requests for blocks, based on two factors:

1. a **ledger** that keeps track of previous block exchanges, in particular of the amount of bytes exchanged and verified between each node;

2. a **strategy**, i.e. a function that given the ledger and the requesting node, returns the probability of accepting the request and sending a block to that node.

In BitSwap, different ledgers and strategies can be chosen, and each one has large impacts on the performance of the entire IPFS network. We will describe here the default options, integrated in the reference IPFS client.

The default ledger is not a distributed globally-synchronized one, such as, for example, the blockchain-based ledger in Bitcoin, Ethereum, or many other cryptocurrencies. Instead, each node keeps its own ledger of the exchanges that took place between itself and other nodes. This allows to keep track of debit or credit towards any node, and to compute a value known as the *debt ratio* in the following manner:

$$debt\ ratio = \frac{bytes\ sent}{bytes\ received + 1}$$

This value is then used in the strategy function to determine the likelihood of sharing blocks with the node from which the debt ratio was computed. In the reference implementation, the strategy function is as follows:

$$P\left(send \mid debt\ ratio\right) = 1 - \frac{1}{1 + e^{6 - 3 \cdot debt\ ratio}}$$

This is a type of function known as sigmoid, more specifically it is an inverted logistic function. It yields values between 0 and 1, being close to 1 with input of 0 or below, and rapidly decreasing towards 0 as the input increases. It's easy to infer that nodes will be able to obtain blocks from a node towards which there is low debt or no debt at all; also, nodes that have exchanged many blocks in the past will be more tolerant of recent debt, allowing for some form of trust between long-lived well-behaved nodes that is directly proportional to the amount of bytes received from that given node.

When two nodes connect, they exchange their ledgers to verify that the information about previous block transfers matches: if it doesn't, both nodes clear all the information they have on the other node, losing both debt and trust that was accumulated before. A mismatch might also happen when a node is trying to clear its debt, so that another node is willing to transfer blocks with it as if it was a new node, therefore nodes are allowed to refuse communication and disconnect when a ledger mismatch happens, but this is not mandatory.

If the ledgers match, peers are connected and allowed to request and transfer blocks. Each block request is evaluated with the strategy function and accepted or denied accordingly: if a request is refused, the requesting node enters in an *ignored* state and is no longer allowed to communicate for a specific amount of time (10 seconds in the reference implementation); this also happens if a received block fails the hash-based data verification step. After every completed and verified block transfer, both nodes update their ledgers.

3.2.2   Analysis

3.3   Ethereum Swarm

3.4   Filecoin

3.5   Summary

Chapter 4

Decentralizing Naming

Chapter 5

Distributed Web Projects

5.1 ZeroNet

5.2 Blockstack

5.3 Analysis

5.3.1 Is decentralization the only solution?

Chapter 6

Upcoming projects

6.1 Substratum

6.2 Hashgraph

Chapter 7

Conclusions

Appendix A

The Domain Name System - A brief overview

The Domain Name System is a decentralized naming system for devices connected to a network (including the Internet), currently defined with RFC 1034rfc [a] and RFC 1035rfc [b] and updated with successive RFCs throughout the years.

The DNS defines three components:

- The *domain name space* is a tree data structure, where nodes are identified by *labels*: labels compose the domain names in a hierarchical way, by concatenation of labels separated by dots. For example, for the domain "www.example.com", "example.com" is a child of "com" and "www.example.com" is a child of "example.com".

- *Name servers* are programs which store information about a subset of the domain space and references to other name servers which have information about the rest of the tree. Name servers have *authority* over the parts of the tree of which they have complete information.

- *Resolvers* are programs which receive queries from clients and respond with information extracted from the name servers. Resolvers only need to know directly just one name server to complete all possible queries: if that name server does not contain the requested information, the resolver uses its references to reach other name servers.

The domain name space has one root node, labeled with an empty string. Childs of this node are called *top-level domains*, among which we can find ".com" and ".org", and two lettered *country codes*, such as ".ch" and ".it". Currently, there are about one thousand different top-level domainsweb.

When resolving a hostname, resolvers query the root name server with the whole domain. The root name server usually replies with the address of the name server which has authority over the top-level domain of the hostname, but it also has facility to reply with the address of the actual server associated with the whole hostname. If the query has not been completed, the query is repeated with the correspondent top-level domain name server, and so on.

To reduce traffic towards the root name servers (and all other name servers), DNS resolvers implement a *caching* system: results from name servers are stored for reuse, together with a time-to-live value specified from the name servers themselves.

Appendix B

Distributed Hash Table

Glossary

- TCP

- ISP

Bibliography

RFC 1034 - Domain names - concepts and facilities. `https://tools.ietf.org/html/rfc1034`, a. [Retrieved 8-June-2016].

RFC 1035 - Domain names - implementation and specification. `https://tools.ietf.org/html/rfc1035`, b. [Retrieved 8-June-2016].

List of Internet top-level domains - Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/List_of_Internet_top-level_domains`. [Retrieved 11-June-2016].

Bram Cohen. Incentives build robustness in bittorrent. 2003.

Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free riding in bittorrent is cheap. Citeseer, 2006.

Manaf Zghaibeh and Fotios C. Harmantzis. Revisiting free riding and the tit-for-tat in bittorrent: A measurement study. *Peer-to-Peer Networking and Applications*, 1(2):162–173, Sep 2008. ISSN 1936-6450. doi: 10.1007/s12083-008-0013-7. URL `https://doi.org/10.1007/s12083-008-0013-7`.