TÉCNICO LISBOA

Startrail

Adaptative Network Caching for Peer-to-peer File Systems

João António Mateus Tiago

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

Supervisors: Prof. Dr. Luís Manuel Antunes Veiga
Eng. David Miguel dos Santos Dias

Examination Committee

Chairperson: Prof. Dr. Ricardo Jorge Fernandes Chaves
Supervisor: Prof. Dr. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Dr. João Nuno de Oliveira e Silva

October 2019

Acknowledgments

The completion of my master thesis remains as one of the toughest projects I have yet executed. Having caught me in the roughest phase of my life, my initial investment to the project was hindered. It required a tremendous amount of perseverance and determination to complete while working a full-time job. I would have given up if it was not for the support of the people close to me and for this, I have to thank you all.

To my beloved girlfriend, Joana Canhoto, you are my biggest motivation in life. You challenge me to do better everyday. Hadn't I met you, I'm sure I would not have finished the thesis. I have to thank you for the countless hours of FaceTime, supporting me until we would both fell asleep.

A special thank you to my teacher and supervisor, Professor Luís Veiga, who has shown to have incredible patience, giving me the time to finish my thesis. His availability for feedback, genuine interest and excitement for the project made this all possible.

I want to massively thank my family for the support and for allowing me to go to college even when the circumstances were not the best and we struggled so much.

To my friends, Henrique Sousa, Fábio Oliveira, João Antunes, Filipe Pinheiro, Renato Castro, João Almeida, Letícia Ferreira, Igor Soarez, Cátia Pereira, Rui Silva, David Dias thank you for never stop believing in me and for supporting me this whole time.

I also want to show my appreciation to my company, YLD, for supporting my investment to finish my thesis. It is a pleasure to work at such amazing company.

Abstract

The *InterPlanetary File System* (IPFS) is a new hypermedia distribution protocol, addressed by content and identities. It aims to make the web faster, safer, and more open. The *JavaScript* implementation of IPFS runs on the browser, thus benefiting from the mass adoption potential that the Web Browser yields. Startrail takes advantage of the ecosystem built by IPFS and strives to further evolve it, making it more scalable and performant through the implementation of an adaptive network caching mechanism. Our solution aims to add resilience to IPFS and improve its overall scalability. It does so by avoiding overloading the nodes providing highly popular content, particularly during flash-crowd-like conditions where such popularity and demand grow suddenly. With this extension, we add a novel crucial key component to enable an IPFS-based decentralized Content Delivery Network (CDN) following a peer-to-peer architecture, running on a scalable, highly available network of untrusted nodes that distribute immutable and authenticated objects which are cached progressively towards the source of requests.

Keywords: Caching, Peer-to-peer, Distributed File System, Content Distribution Network, Decentralized Distributed Systems, Web Platform, JavaScript, IPFS

Resumo

O *InterPlanetary File System* (IPFS) é um protocolo inovador para transmissão de hipermédia, endereçada por conteúdo e identidades. Ambiciona tornar a *web* mais rápida, segura e acessível. A implementação do IPFS em *JavaScript* é capaz de executar num *Browser*, beneficiando assim do potencial para adopção massiva que este consegue oferecer. O Startrail usufrui do ecossistema construído em volta do IPFS e pretende evolui-lo e melhorando a sua *performance* através da implementação de um mecanismo de *cache* de rede adaptativa. A nossa solução pretende tornar o IPFS mais resiliente e aumentar a escalabilidade do sistema. Fá-lo prevenindo sobrecarregar nós que estejam a servir conteúdo muito popular, especialmente quando sob condições tipo *flash-crowd*, onde a popularidade e procura dos objectos cresce de forma muito repentina. Com esta extensão adicionamos ao IPFS um inovador e crucial componente que é chave para viabilizar uma rede de distribuição de conteúdos (CDN) descentralizada e entre-pares, operando sob IPFS, uma rede escalável, altamente disponível mesmo na presença de nós bizantinos. Esta distribuindo assim objectos imutáveis e autenticados, que são progressivamente guardados na direção da origem dos pedidos.

Palavras-Chave: Caching, Redes entre pares, Sistema de Ficheiros Distribuido, Rede de distribuição de conteúdo, Plataforma Web, JavaScript, IPFS

Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interface. 23, 24

CDN Content Delivery Network. iii, 3

CI Continuous Integration. 34

CID Content Identifier. 22, 25, 27–31

CLI Command-line Interface. 23, 36

GC Garbage Collector. 31

HTTP Hypertext Transfer Protocol. 3, 4

IPFS InterPlanetary File System. 4

K8s Kubernetes. 34

P2P peer-to-peer. 3

TTL Time To Live. 21

Chapter 1

Introduction

In the early days, the Internet was basically a mesh of machines whose main purpose was to share academic and research documents. It was predominantly a peer-to-peer (P2P) system. The Original ARPANET [1] connected UCLA, Stanford Research Institute, UC Santa Barbara and the University of Utah not in a client-server format, but as equal computing peers. Early popular applications of the internet, FTP [2] and Telnet [3], were themselves client-server applications but since every host on the Internet could FTP or Telnet to any other host, the usage patterns of the Internet were symmetric. In the early Internet, computers connected to the network played an equal role, each capable of contributing with as much as they utilised.

The Web enabling protocol, Hypertext Transfer Protocol (HTTP), presented by Sir Tim Berners-Lee, had in nature a decentralized design. It was only due to network topology constraints, mainly NATs, that users on World Wide Web lost the ability to directly dial other peers. Struggling to overcome obstacles in interoperability of protocols, the gap between client and server nodes widen and the pattern remained. Here, computers play either the role of a consumer - “client”- or producer - “server” - serving content to the network. Serving a big client base required enormous amounts of server resources. In this model, as demand grows, performance deteriorates and the system becomes fragile. The same number of servers has to meet the demands of a larger and larger number of clients. Sharing the same server resources with a growing crowd leads to server and network overload, causing performance degradation for each client. Moreover, such architecture is inherently fragile. Every single source of content at the servers is a single point of failure that can result in complete failure and lengthy downtime of the system.

To tackle such flaws, technologies like CDNs emerged to aggregate and multiplex server resources for many sources of content. This way, a sudden burst of traffic could be more easily handled by sharing the load. To further improve the quality of the service CDN providers were incentivized to move capacity closer to clients. Such innovations made the early client-server mode a little more robust, but at considerable cost. Still, despite its inefficiency, the client-server model remains dominant today and runs most of the web.

1.1 Motivation

The InterPlanetary File System (IPFS)[4] seeks to revert the historic trend of a client-server only Web and replace HTTP. It is a decentralized peer-to-peer content-addressed distributed file system that aims to connect all computers offering the same file system (contents and namespace). Due to its decentralized nature, IPFS is intrinsically scalable. As more nodes join the network and content demand increases, so does the resource supply. Such a system is incredibly fault tolerant and, leveraging economies-of-scale, actually performs better as its size increases. It uses Merkle DAGs[5][6], the concept of hash-linked Merkle[7] data structures to provide immutable, tamper-proof objects that are content addressed.

However, there are some crucial Content Distribution Network (CDN) enabling features are lacking in the InterPlanetary File System. In particular, the system lacks the capability of swiftly and organically approximate content from the request path, reducing the latency felt by future requests. It also does not prepare for the provider of an object to serve a sudden flood of requests, thus rendering content inaccessible to some.

1.2 Goals and Contributions

The goal of this work is to, taking advantage of the ecosystem build by IPFS, develop an extension to IPFS that implements an adaptive distributed cache that will improve the system's performance, further evolving it. We aim to:

- Reduce the overall latency felt by each peer;
- Increase the peer' throughput retrieving content;
- Reduce the system's overall bandwidth usage;
- Improve the overall balance in serving popular content by peers;
- Finally, improve nodes' resilience to flash crowds.

Considering the above mentioned goals, such extension will yet make the overall system more resilient, allowing smaller entities to server large consumer basis. The gains in performance improve IPFS' scalability.

Startrail can thus serve as a key enabling and core component for future deployment of IPFS-based CDNs

Furthermore, the objective of this document is also to survey the major areas of research that are relevant for the design of the proposed solution, as well as document the current state of the art regarding these areas of study.

1.3 Document Organization

This document is organized as follows. In Chapter 2 we review the state of the art algorithms and architectures used by the relevant systems. Chapter 3 describes the proposed solution: we start by

addressing the architecture of IPFS as a starting point to better understand the integration points of the proposed solution. Next, we further describe Startrail's architecture, its algorithms and data structures. Chapter 4 describes the testbed platform used, all the relevant metrics and obtained results. Some concluding remarks and extension proposals are presented in Chapter 5.

Chapter 2

Related Work

This chapter reviews relevant, related research work for designing features relevant to a distributed, highly available and scalable, peer-to-peer Content Distribution Network. Such topics comprise: Content Sharing Networks, Content Distribution Networks and Web Distributed Technologies.

2.1 Content Sharing

A Distributed File System is a file system that supports the sharing of files in the form of persistent storage over a set of network connected nodes. Multiple users who are physically dispersed in a network of autonomous computers share a common file system. The challenge is in realizing this abstraction in a performant, secure and robust manner. In addition, the issues of **file location** and **availability** assume significance. One way of increasing the availability is by using data **replication** and in order to increase performance **caching** can also be used.

2.1.1 Architectures

Distributed File Systems may follow different types of architectures [8].

Client-Server Architecture The simples architecture is the Client-Server architecture that exposes a directory tree to multiple clients (e.g. NFSv3 [9]). A communication protocol allows clients to access the files stored on a server thus allowing a heterogeneous collection of processes running on different operating systems and machines share a common file system. The clear problem with the Client-Server architecture is that the total capacity of the system is limited by the capacity of the server. The server is a single point of failure, meaning that in the case of a server crash, network outage or power cut the whole system goes down. An approach to overcome some of these limitations is to delegate ownership and responsibility of certain file system subtrees to different servers, as done by AFS [10]. In order to provide access to remote servers, AFS allows for loose coupling of multiple file system trees (“cells”). However this has reduced flexibility as the partitioning of a file system tree is static and changing it requires administrative intervention.

Object-based File Systems like the Microsoft DFS (MSDFS) [11] and the Google File System (GFS) [12], present a different approach. Metadata and data management are kept separated. A master server maintains a directory tree and keeps track of replica servers. It handles data placements and load balancing. As long as metadata load is much smaller than data operations (i.e files are large), this architecture allows for incremental scaling. Adding data servers, as load increases, with minimal administrative overhead.

These previously presented architectures are examples of **asymmetric architectures**. There is a clear difference between client and server. Because of this difference, maintaining a highly available and performant distributed file system presents huge costs in infrastructure. We need to consider that any viable distributed system architecture must support the notion of autonomy if it is to scale at all in the real world.

With the aim of solving this issue, **symmetric architectures** arise. Based on peer-to-peer technology, every node on the network hosts the metadata manager code, resulting in all nodes understanding the disk structures. Thus removing the single point of failure and enabling the system to grow continuously.

Peer-to-Peer File Systems (P2P) are distributed systems consisting of interconnected nodes (peers). These systems are fully decentralized and capable of accommodating transient populations of nodes. They serve the purpose of aggregating resources such as content, CPU cycles, storage, and bandwidth [13].

The following paragraphs synthesize the current state of the art regarding peer-to-peer organizations.

Overlay Network Structure In peer-to-peer networks structure means whether the overlay network is created non-deterministically - ad-hoc - as nodes and content are added or whether its creation and maintenance is based on a set of specific rules [14].

Unstructured P2P architecture. In this architecture, the system imposes no constraints on the links between different nodes, and therefore the systems have no particular structure. Systems like Gnutella [1], Kazaa [15]employ this kind of overlay. The placement of content is completely unrelated to the overlay topology. Although requiring little maintenance while peers enter and leave the system (i.e. churn), unstructured P2P systems suffer from the lack of an inherent way to index data. Because of this, resource lookup mechanisms consist of brute-force methods, like flooding, and random walk where the network propagates queries. This property limits the networks ability to scale to very large populations. In order to solve this problem systems like Napster and BitTorrent [16] use a hybrid centralized architecture, where server maintains directories of information about registered users in the network. While lookups are maintained by a single entity, file transferences are kept decentralized. Another similar approach is taken by systems like Kazaa and Gnutella 0.6 [2] which use the concept of 'supernodes'. Peers with sufficient bandwidth and computing power are elected supernodes. These nodes maintain the central indexes for the information shared by local peers connected to them, and proxy search requests on behalf of these peers. Queries are therefore sent to SuperNodes, not to other peers. Supernodes remain single points of failure and redundancy techniques should be employed to prevent downtime.

Structured networks emerged as a way to address the scalability issues of unstructured systems. In these networks, the overlay topology is tightly controlled and data is placed at precisely specified

[1] http://rfc-gnutella.sourceforge.net/
[2] http://rfc-gnutella.sourceforge.net/

locations in the overlay [17] . These systems provide a mapping between the data identifier and location, in the form of a distributed routing table, so that queries can be efficiently routed to the node with the desired data. Systems like Chord [18], Pastry [19] and Tapestry [20] use a Distributed Hash Table (DHT) in order to quickly and consistently perform lookups. The Oceanstore Protocol [21] is then built on top of the Tapestry DHT. In structured P2P systems each node is given a unique, uniformly distributed identifier in a large numeric keyspace using a cryptographic hash function SHA1 [22]. This identifier determines the position of the node in the hash ring where it becomes responsible for a segment, leveraging the responsibility of forwarding messages to its 'fingers' (nodes that it knows the whereabouts). Kademlia [23] organizes its nodes in a balanced binary tree, using XOR as a metric to perform the searches, while CAN [24] introduced and a several dimension indexing system, in which a new node joining the network, will split it's share of the coordinate space with another node that has the most to leverage. Since there is no centralized infrastructure responsible for placing new nodes, in structured P2P networks, the system has to always verify that the newly generated node-id does not yet exist, in order to avoid collisions. Another problem with this type of architecture is the overhead in maintaining the structure required for efficient routing in the face of a very dynamic population (i.e churn).

Some relevant systems previously mentioned are further analized in table 2.1.1.

Table 2.1: Distributed File Systems Overview

Systems vs. Features	Architecture	Lookup	Caching	Load Balancing	Data Replication	Reference
NFS	Client-Server	Central Metadata Server	Client-side caching	*	*	[9]
AFS	Client-Server	Central Metadata Server	Client-side caching	Manual	Manual, Read-only	[10]
DFS	Client-Server	Central Metadata Server	Client-side caching	*	*	[11]
GFS	Cluster-based	Central Metadata Server	Meta-data cached on the client	Periodic load redistribution on replicas	Replicated up to 3x on diff. machines	[12], [8]
IPFS	Structured P2P	DSHT	Peers that have fetched content may serve it; (and deduplication)	IDs uniform distribution across the key-space	Peers may 'pin' objects to replicate (off by default)	[4]
Kademlia	Structured P2P	DSHT	Push-based; Peers close to the key that not store the object	Load distributed over the nodes that cache/replicate objects	Periodic Re-Replication of k nodes	[23]
Oceanstore	Structured P2P	Tapestry's DHT	Peers cache pointers to host node	Randomness for load distribution	Multiple replicas of same object	[21]

- **Architecture**: What is the architecture of the system;
- **Lookup**: How data is located on the system;
- **Caching**: Systems' caching techniques;
- **Load Balancing**: How requests are distributed across the system's replicas;
- **Data Replication**: How or where is data replicated on the system

2.2 Content Distribution

With the ever-increasing Internet traffic [3] service providers work hard to maintain a high quality of service. Popular Web services often suffer congestion and bottlenecks due to large demand. Such a scenario may cause unmanageable levels of traffic, resulting in many requests being slowed down or even dropped. To Safeguard themselves from such events, web services usually resort to CDNs.

Content Delivery Networks (CDNs) are networks of geo-distributed machines that deliver web content to users based on their geographic location.[25] CDNs can improve the speed of the content delivery while also increasing the service availability at the cost of replicating it over several machines. When a user requests for content hosted on a CDN, a server inside the network will redirect the request to the replica that is closer to the user and deliver the cached content.

The three key components of a CDN architecture are the content provider, CDN provider, and end-users. A content provider or customer is the one who delegates the URI namespace of the Web objects to be distributed. The origin server of the content provider holds those objects. A CDN provider is a proprietary organization or company that provides infrastructure facilities to content providers in order to deliver content in a timely and reliable manner to a large number of end-users. End-users or clients are the entities who access content from the content provider's website.

In the next paragraphs, we will take a look at the various architectures and strategies CDN providers use.

2.2.1 CDN Taxonomy

CDN Organization There are generally two approaches to building a CDN: overlay and network approach. Most commercial CDNs like Akamai [26], AppStream [4], Limelight Networks [5] follow the **overlay approach**. Here, application-servers and caches are responsible for content distribution. Network elements such as routers and switches play no active role in the content delivery. This makes up for a simpler management. In the network approach, the provider also controls some network infrastructure equipped with code for identifying specific application types and for forwarding the requests based on predefined policies.

Content Selection and Delivery The right selection of content to be delivered is key to the effectiveness of a CDN. An appropriate content selection approach can assist in the reduction of client download time and server load. Content can be delivered to the customers in full or in partial. The simplest and most straightforward approach is the **full-site approach**. Here, the whole set of origin server's object is stored at the surrogates. The latter takes on the former's entire task of serving the object. The content provider configures the DNS resolution such that all client requests to its Web site are served by the CDN server. Although simple, this approach is not feasible considered the ever-growing size of Web objects. While the cost of physical storage is decreasing, sufficient storage space on the edge servers is never guaranteed to store all the content from content providers. Moreover, since the Web content is not static, the problem of updating such a huge collection of Web objects is unmanageable.

[3] https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html

[4] https://aws.amazon.com/pt/appstream2/

[5] https://www.limelight.com/

On the other hand, **partial-site selection and delivery** is a technique in which surrogates only host a portion of the whole object to deliver only the embedded objects - such as Web page images - from the corresponding CDN. With partial-site content delivery, a content provider modifies its content so that links to specific objects have hostnames in a domain for which the CDN provider is authoritative. Thus, the base HTML page is retrieved from the origin server, while embedded objects are retrieved from CDN cache servers. The Partial-site approach is better in the sense that it reduces the load on the origin server and on the site's content generation infrastructure. Plus, due to infrequent change of embedded content, partial-site exhibits better performance. There are, though, many different techniques for selecting which partial content to cache. The most straightforward one is **empirical-based** approach. Here, the Website admin selects which content will be replicated to the edge servers. In the **popularity-based** the most popular objects are replicated to surrogates. This technique is very time consuming and reliable object statistics are not guaranteed since the popularity of each object varies significantly. Furthermore, such statistics are not often available for newly added content. In **Object-based** approach, content is replicated, in units of objects, to the surrogate that gives the higher performance gain. This is a greedy technique and has a high complexity that makes it hard to implement in real applications. In the **cluster-based** approach content is grouped either on correlation or access frequency and is replicated in units of content clusters. When clustering, there may be a maximum number of clusters or a maximum cluster diameter. The choice of content to cluster can be either **users' sessions-based** or **-based**. In the former, the users' navigation sections which show similar patterns have their pages clustered together. In the URL-based approach, the most popular objects are identified from a Web site and are replicated in units of clusters where the correlation distance between every pair of URLs is based on a certain correlation metric. Experimental results show that content replication based on such clustering approaches reduce client download time and the load on servers. But these schemes suffer from the complexity involved to deploy them.

P2P systems like IPFS [4] and BitTorrent [16] use a different strategy called **Block level** replication: divides each file into an ordered sequence of fixed size blocks. This is also advantageous if a single peer cannot store the whole file. A limitation of block level replication is that during file downloading it is required that enough peers are available to assemble and reconstruct the whole file. Even if a single block is unavailable, the file cannot be reconstructed. To overcome this problem, Erasure Codes (EC), such as Reed-Solomon [27] are used.

Content outsourcing Given a set of properly placed surrogate servers in a CDN infrastructure and a chosen content for delivery, choosing an efficient content outsourcing practice is crucial. Content outsourcing is performed using either of cooperative push-based, non-cooperative pull-based and cooperative pull-based approaches. **Cooperative push-based**: This approach is based on the pre-fetching of content to the surrogates. Content is pushed to the surrogate servers from the origin, and surrogate servers cooperate to reduce replication and update cost. In this scheme, the CDN maintains a mapping between content and surrogate servers, and each request is directed to the closest surrogate server or otherwise, the request is directed to the origin server. Under this approach, a greedy-global heuristic algorithm is suitable for making replication decision among cooperating surrogate servers. Still, it is considered as a theoretical approach since it has not been used by any CDN provider. **Non-cooperative pull-based**: In this approach, client requests are directed (either using DNS redirection or URL rewriting) to their closest surrogate servers. If there is a cache miss, surrogate servers pull content from the origin server. Most popular CDN providers (e.g. Akamai [26]) use this approach. The drawback of this approach is that an optimal server is not always chosen to serve content request. Many CDNs use this approach since the cooperative push-based approach is still at the experimental stage. **Co-**

operative pull-based: The cooperative pull-based approach differs from the non-cooperative approach in the sense that surrogate servers cooperate with each other to get the requested content in case of cache miss. In the cooperative pull-based approach client requests are directed to the closest surrogate through DNS redirection. Using a distributed index, the surrogate servers find nearby copies of requested content and store it in the cache. The cooperative pull-based approach is reactive wherein a data object is cached only when the client requests it. An academic CDN Coral [28] has implemented the cooperative pull-based approach using a variation of Distribution Hash Table (DHT).

Caching Techniques In a **query-based scheme**, on a cache miss a CDN server broadcasts a query to other cooperating CDN servers. The problems with this scheme are the significant query traffic and the delay because a CDN server has to wait for the last 'miss' reply from all the cooperating surrogates before concluding that none of its peers has the requested content. In **digest-based scheme**, each of the CDN servers maintains a digest of content held by the other cooperating surrogates. The cooperating surrogates are informed about any sort of update of the content by the updating CDN server. On checking the content digest, a CDN server can take the decision to route a content request to a particular surrogate. The main drawback is that it suffers from update traffic overhead, because of the frequent exchange of the update traffic to make sure that the cooperating surrogates have correct information about each other. In a **hashing-based scheme** the cooperating CDN servers maintain the same hashing function. A designated CDN server holds a content based on content's URL, IP addresses of the CDN servers, and the hashing function. All requests for that particular content is directed to that designated server. A hashing-based scheme is more efficient than other schemes since it has smallest implementation overhead and highest content sharing efficiency. However, it does not scale well with local requests and multimedia content delivery since the local client requests are directed to and served by other designated CDN servers. Under the **semi-hashing-based scheme**, a local CDN server allocates a certain portion of its disk space to cache the most popular content for its local users and the remaining portion to cooperate with other CDN servers via a hashing function. Like pure hashing, semi-hashing has small implementation overhead and high content sharing efficiency. In addition, it has been found to significantly increase the local hit rate of the CDN

Cache Update Cached objects in the surrogate servers of a CDN have associated expiration times after which they are considered stale. Ensuring the freshness of content is necessary to serve the clients with up to date information. The most common cache update method is the **periodic update**. To ensure content consistency and freshness, the content provider configures its origin Web servers to provide instructions to caches about what content is cacheable, how long different content is to be considered fresh when to check back with the origin server for updated content. This approach suffers from significant levels of unnecessary traffic generated from update traffic at each interval. On another approach, an **update propagation** is triggered by a change in content. It performs active content pushing to the CDN cache servers. In this mechanism, an updated version of a document is delivered to all caches whenever a change is made to the document at the origin server. For frequently changing content, this approach generates excess update traffic. Another cache update approach **is invalidation**, in which an invalidation message is sent to all surrogate caches when a document is changed at the origin server. The surrogate caches are blocked from accessing the documents when it is being changed. Each cache needs to fetch an updated version of the document individually later. The drawback of this approach is that it does not make full use of the distribution network for content delivery and belated fetching of content from the caches may lead to inefficiency of managing consistency among cached contents.

Performance Measurement

- **Cache hit ratio**: It is defined as the ratio of the number of cached documents versus total documents requested. A high hit rate reflects that a CDN is using an effective cache policy to manage its caches.
- **Reserved bandwidth**: It is the measure of the bandwidth used by the origin server. It is measured in bytes and is retrieved from the origin server.
- **Latency**: It refers to the user-perceived response time. Reduced latency signifies the decreases in bandwidth reserved by the origin server
- **Surrogate server utilization**: It refers to the fraction of time during which the surrogate servers remain busy. This metric is used by the administrators to calculate CPU load, the number of requests served and storage I/O usage.
- **Reliability**: Packet-loss measurements are used to determine the reliability of a CDN. High reliability indicates that a CDN incurs less packet loss and is always available to the clients.

2.3 Web Distributed Technologies

2.3.1 The Web platform

The Browser is a very powerful tool for fueling adoption. Building a solution that runs on the browser means one is are able to reach a broad audience without the need for, by itself, support many different operating systems. That is why any modern day network technology that wants to reach mass adoption is required to support the Web platform. Three main programming languages power the Web Browser: HTML, CSS and Javascript [29]. Since the first two focus on Web page's structure and appearance, respectively, we are only going to focus on the third. Javascript (or JS) is a lightweight, dynamic, interpreted and recently JIT-compiled language best known as the scripting language of the Web. As a multi-paradigm language, it supports the event-driven, imperative (including object-oriented and prototype-based) and functional programming styles. Introduced in 1995, by Brendan Eich at Netscape [6], it executes in the client's browser, with the intent of automating parts of a web page and make a web page more dynamic. Javascript is also used by many non-browser environments. Node.js [7], revised below, uses JS as the scripting language. MongoDB [8] accepts queries written in Javascript. Adobe's Acrobat and Adobe Reader support JavaScript in PDF files [9]. Google Apps Script accepts javascript as scripting language as a way for task automation [10].

Node.js - Prior to Node.js, other attempts at running Javascript in the server had been made [11], but none was able to successfully provide a true stand-alone runtime environment. Created by Ryan Dahl in 2009, Node.js is built on top of Google Javascript engine V8 [12]. It materializes Netscape's original vision

[6] https://web.archive.org/web/20080208124612/http://wp.netscape.com/comprod/columns/techvision/innovators_be.html
[7] https://nodejs.org/en/
[8] https://www.mongodb.com/
[9] https://www.adobe.com/devnet/acrobat/javascript.html
[10] https://www.google.com/script/start/
[11] Netscape's Lime-wire
[12] https://github.com/v8/v8/wiki

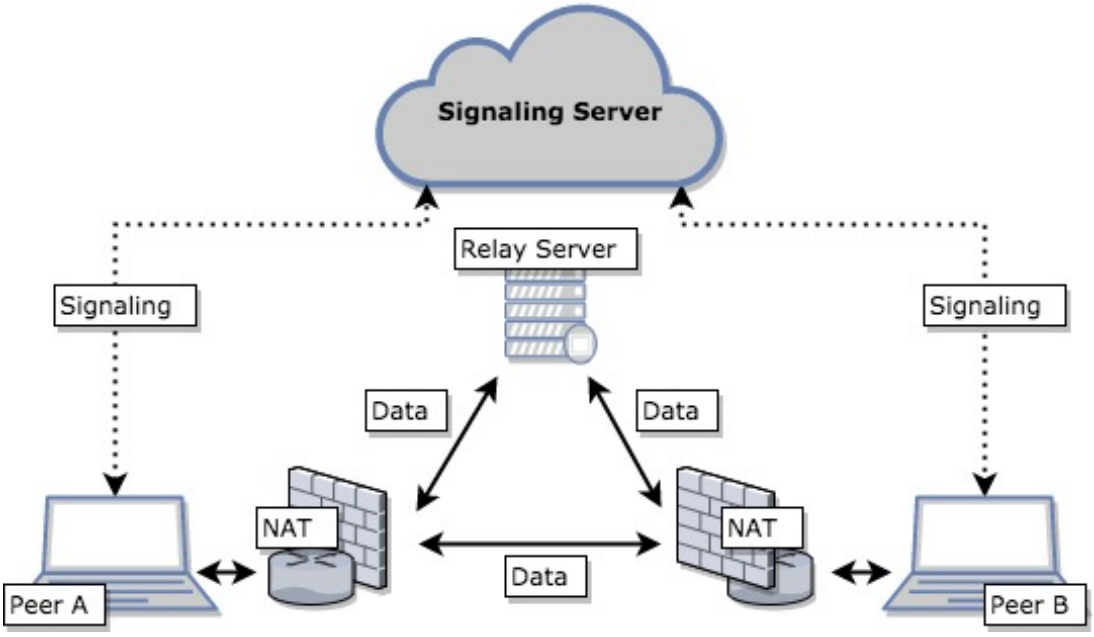

Figure 2.1: Web P2P Networking

of Javascript in the browser and in the server. Thus, providing minimal overhead when the developer switches between both environments. Node.js provides a low-level I/O API and, to cope with the lack of multi-threading, it uses an event loop which offloads operations to the system kernel whenever possible. This forces the developer to use non-blocking asynchronous operations so not to block the event loop. Node.js runs in most architectures and operating systems exposing a set of libraries, "modules" that handle various core functionality such as networking (HTTP, TCP...), binary data handling, cryptography functions, access to the filesystem amongst others. Crucial to Node's adoption was its package manager, NPM [13]. NPM incentivizes the minimalist, modular Unix philosophy of writing software that does one thing only and does it well. Each reusable small module should expose a clear interface, carry documentation and written tests which ease debugging a bigger composition of modules.

2.3.2 Peer-to-peer in the browser

WebRTC [14] is a project supported by Google, Mozzila and Opera, which provides browsers and mobile applications with peer-to-peer Real-Time Communications capabilities accessible through a Javascript API. This enables applications to transfer Audio, Video or arbitrary data, other users' browsers without the need of either internal or external plugins. To enable peer-to-peer connections, WebRTC uses a collection of communication protocols for NAT traversal and to provide a reliable transport layer. Namely, BitTorrent's uTP [15] or Google's QUIC [30]. In the following paragraphs we will address the key aspects that make up the system which are depicted in the figure 2.1.

The Problem With TCP The Internet was built over TCP largely because it is a reliable transmission protocol. But where TCP shines in reliability, it dims in the number of round trips required to establish a secure connection. A secure connection is needed before a browser can request a web page. Secure

[13] https://www.npmjs.com/
[14] http://w3c.github.io/webrtc-pc/
[15] `http://www.bittorrent.org/beps/bep_0029.html`

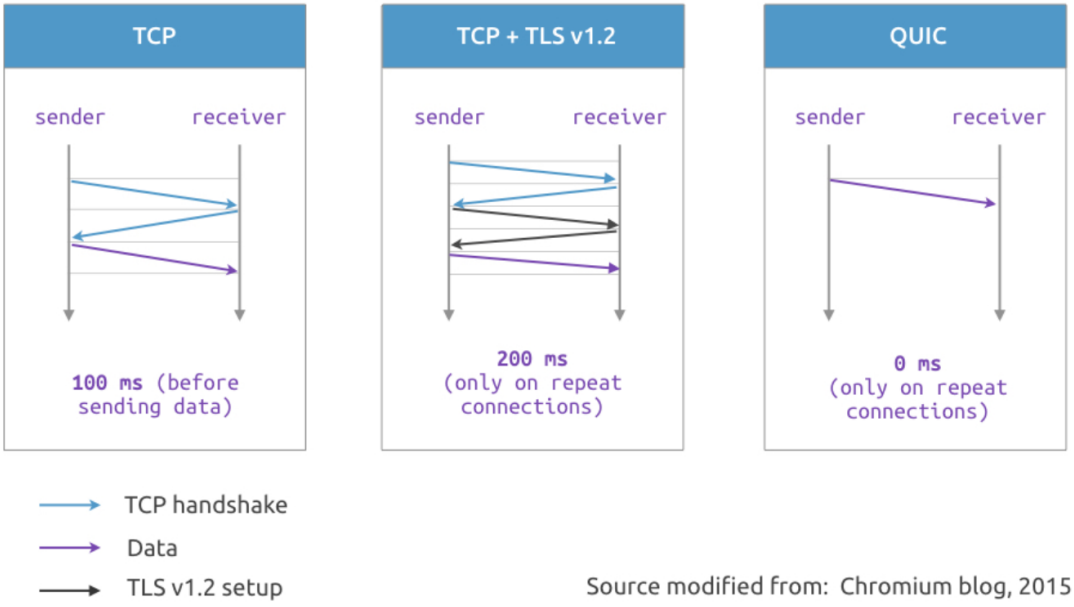

Figure 2.2: Handshake comparison between TCP-only, TCP-with-TLS and QUIC protocols.

web browsing usually involves communicating over TCP, plus negotiating TLS to create an encrypted https connection. This approach generally requires at least two to three round trips – packets being sent back and forth – with the server to establish a secure connection. Each round trip increases the latency to any new connection. Web browsers establish many parallel connection to the same serve. Another issue with TCP is the head-of-line-blocking (HOLB). HOLB happens when a TCP packet is lost enroute to the reciever, then all subsequent packets must be held in the receiver's TCP buffer until the lost packet is retransmitted and arrives at the receiver.

To solve this, Google designed QUIC [30]. QUIC reduces the amount of RTT to setup a connection down to 0 (the first RTT carries data already) as seen in the figure 2.2, it also supports a set of multiplexed connections between two endpoints over UDP, and was designed to provide security protection equivalent to TLS/SSL, along with reduced connection and transport latency, and bandwidth estimation in each direction to avoid congestion. To solve the head-of-line-blocing issue, QUIC allows applications to decide what to do with the incomplete stream by moving the congestion control algorithms into application space. Also to tackle these TCP issues BitTorrent developers designed the, UDP-based, Micro Transport Protocol (uTP) [16].

NAT In 1992, as in [31], the Routing and Addressing Group (ROAD) from the Internet Engineering Task Force (IETF) was tasked with tackling scalability issues on the IP architecture [31]. Hence, in 1994 [32], the Network Address Translation (NAT) devices where the solution found. NATs are devices that map scarce public IP addresses to private IP ones, only reachable from within the network. The NAT has a public IP address and machines connected to it will have private ones. Requests made from the inside of the network to the outside will be translated to the NAT's public IP and given an unique port. Like so, the rare public IP addresses are reused. Before NATs, it was possible to directly dial to another machine on the network without a problem. One just created a connection to the destination's IP address. With NATs that is not longer possible. Since there are potentially many private IP addresses behind a public IP, the router (NAT) is unnable to know to which one it should to route a specific packet.

[16] http://www.bittorrent.org/beps/bep_0029.html

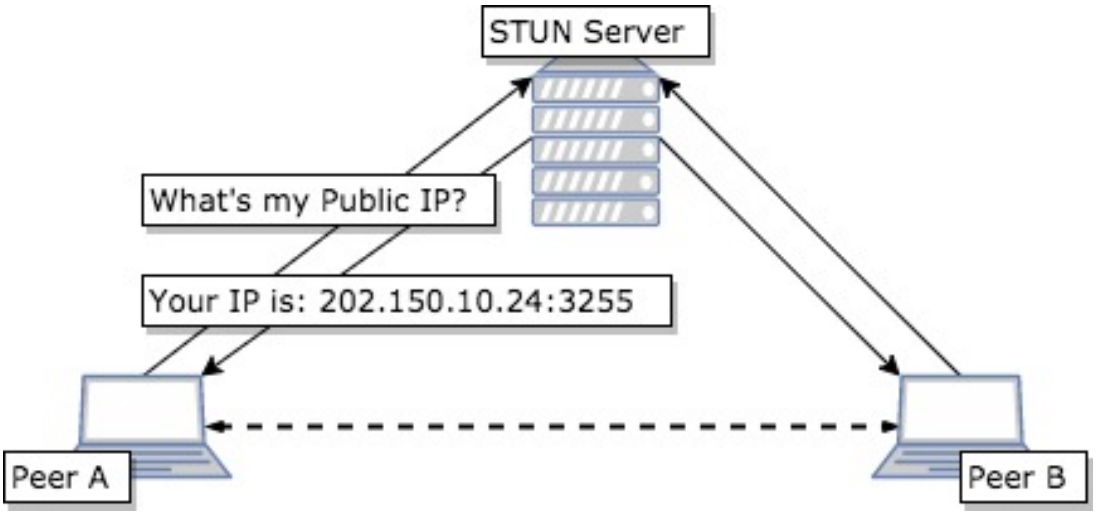

Figure 2.3: STUN Protocol operation

ICE Interactive Connectivity Establishment (ICE) is a framework to allow your web browser to connect with peers. ICE addresses the aforementioned problem of NATs. It has to bypass firewalls that prevent connections from being opened, give a machine a public IP so that it is reachable from the outside of the network (in case it does not have one), work around firewall prohibited protocols and, in case it is not possible to connect directly to other peers, relay data through a server. In order to do such NAT traversal, ICE uses STUN [33] and TURN [34] protocols. Bellow we address both this protocols in detail.

STUN Session Traversal Utilities for NAT (STUN) [17] is a protocol to discover a machine's public address and determine any restrictions in its router that would prevent a direct connection with another peer. STUN's behaviour can be observed in figure 2.3 bellow.

Clients contact the STUN server on the Internet who replies with the client's public (NATed) IP and source port. This information is then exchanged with other peers, for example through a SIP server (Session Initiation Protocol), enabling other nodes to use the just created mapping, thus traversing the NAT. Sometimes though, it is not possible to access the client behind the router's NAT. In the case the router is using a Symmetric NAT, like in most big corporate networks, it is not possible to use this sort of technique.

Symmetric NAT is a specific type of NAT that does not reuse the origin port of the machine. This means that even though the machine has a public IP address mapping found by the STUN server, this one is not available to other peers. In this case, for every new connection a new mapping is created. In this situation a TURN has to be used.

TURN In case the router employs Symmetric NAT, it will only accept connections from peers the client has previously connected to. Traversal Using Relays around NAT (TURN) was designed to bypass this restriction by opening a connection with a TURN server and relaying all information through that server. After the connection is opened the client may ask all other peers to contact him through the TURN server's public IP address which will then forward all packets to him. TURN's operation may be further observed in the figure 2.4

[17] acronym within an acronym

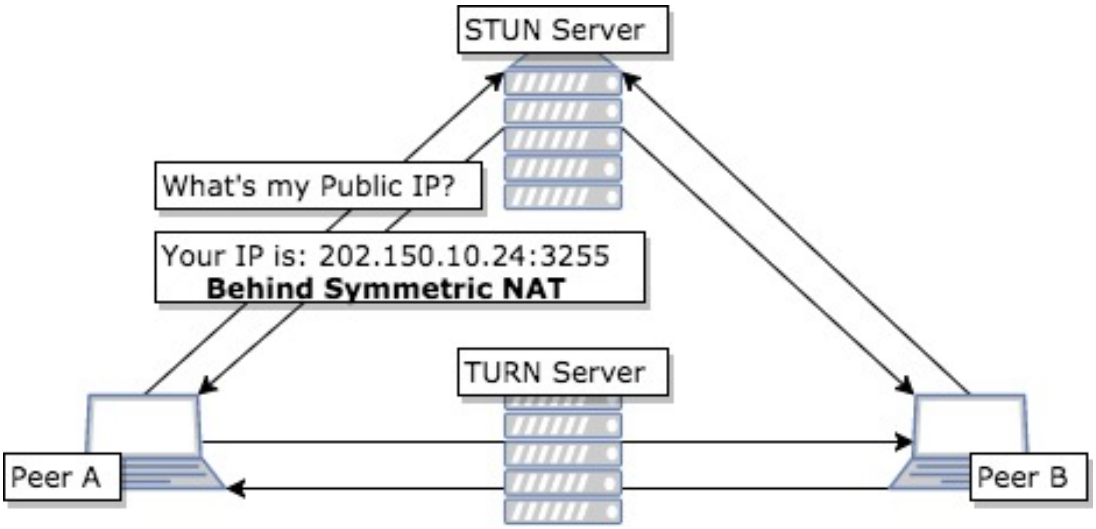

Figure 2.4: TURN Protocol operation

Although TURN almost always provides connectivity to other peers, relaying data through the intermediate server is a resource intensive operation. Therefore, ICE will only use this as a last resource.

Wrapping up, ICE always tries to connect peers directly, with the lowest latency possible, hence UDP is used. If UDP fails, ICE tries to use TCP. In order to connect peers, WebRTC uses the ICE protocol to quickly figure out the best way to connect them (i.e the best ICE candidate). It does so in parallel and settles on the cheapest path that actually works. WebRTC forces encryption in all for media and data. Because it is such a disruptive technology in the browser, WebRTC is enabling a completely new set of applications that before were not available for the Web platform. WebTorrent [18] makes use of this technology to enable browsers to connect to the BitTorrent network. PeerCDN [35] uses WebRTC to offload the burden of hosting content to website visitors. IPFS.js [19] is the javascript implementation of IPFS that runs on Node.js and in the browser, may also use WebRTC as a transport protocol.

2.4 Relevant Systems

2.4.1 Oceanstore

OceanStore [21] is a global-scale persistent data store. It provides a consistent, highly-available, and durable storage utility atop an infrastructure comprised of untrusted servers. Each object is addressed by the hash of the owners key plus a human readable name. Objects may be lookup up through two different alternatives: first, a fast, probabilistic algorithm attempts to find the object near the requesting machine using *Attenuated Bloom Filters*. If the probabilistic algorithm fails, location is left to a slower, deterministic algorithm. Every update in Oceanstore creates a new version, thus allowing “permanent” pointers. Because Oceanstore separates information from its physical location, data in this system is considered to be *nomadic*. Using this concept it introduces the concept of *Promiscuous Caching* - data may be cached anywhere, anytime. This notion differs from previous systems like NFS and AFS where data is confined to particular servers in particular regions of the network. Oceanstore presents the notion

[18] https://webtorrent.io/faq
[19] https://github.com/ipfs/js-ipfs

of *deep archival storage*. Archival versions of objects are read-only objects encoded with an erasure code and replicated over hundreds or thousands of servers. Since data can be reconstructed from any sufficiently large subset of fragments it would require a massive fraction of the network to be destroyed before content could considered to be removed. Oceanstore supports a large range consistency policies, ranging from extremely loose semantics to supporting the ACID semantics favoured in databases. To enhance performance, Oceanstore also supports prefetching mechanism they named *introspection*.

2.4.2 CoralCDN

CoralCDN [28] is a peer-to-peer content distribution network that allows a user to run a web site that offers high performance and meets huge demand. Volunteer sites that run CoralCDN automatically replicate content as a side effect of users accessing it. Publishing content through CoralCDN is done by simply changing the hostname in an object's URL. Coral uses a peer-to-peer DNS layer transparently that redirects browsers to nearby participating cache nodes, which in turn cooperate to minimize load on the origin web server. One of the system's key goals is to avoid creating hot spots that might dissuade volunteers and hurt performance. It achieves this through Coral, a latency-optimized hierarchical indexing infrastructure based on a novel abstraction called the distributed sloppy hash table, or DSHT. Wich relaxes the DHT API from `get_value(key)` to `get_any_values(key)`. Two properties make Coral particulary fit to CDNs. First, Coral allows nodes to locate nearby cached copies of web objects without querying more distant nodes. Second, Coral prevents hot spots in the infrastructure, even under degenerate loads. For instance, if every node repeatedly stores the same key, the rate of requests to the most heavily-loaded machine is still only logarithmic in the total number of nodes. Coral organizes a hierarchy of separate DSHTs called *clusters* depending on region and size, where nodes are able to query peers in their region first, and greatly reducing the latency of lookups.

2.4.3 PeerCDN

PeerCDN [35] proposes an hybrid peer-to-peer architecture to leverage the pros and cons of (centralised) CDNs and P2P networks for providing a scalable streaming media service. PeerCDN suggests the use of a two-layered architecture like see in the figure 2.5 bellow. The upper layer is a server layer which is composed of original CDN servers including origin servers and replica servers. The lower layer is composed of peer-to-peer network of clients who requested the streaming services (i.e downloaded the media file). The network has a topology-aware, Kademlia-like topology and a uses a DHT for data retrieval. For each network there is a coordinating node, the *strong node* that connects the network to the upper layer. Using the client peers participation, PeerCDN is able to achieve a higher capacity than traditional CDNs, extending the scale of the CDN in the edge-side. Using the topology-aware overlay network, PeerCDN restricts the unnecessary backbone bandwidth consuming during client peer sharing.

2.4.4 IPFS

InterPlanetary File System (IPFS) [4] is a peer-to-peer hypermedia protocol that seeks to create a network of persistent, content-addrassable objects as part of the same Merkle DAG. Merkle DAGs (Merkle Directed Acyclic Graph) are key to IPFS mechanics, they are directed acyclic graphs linked together via cryptographic hashes. The cryptographic integrity checking properties of this data structure allows

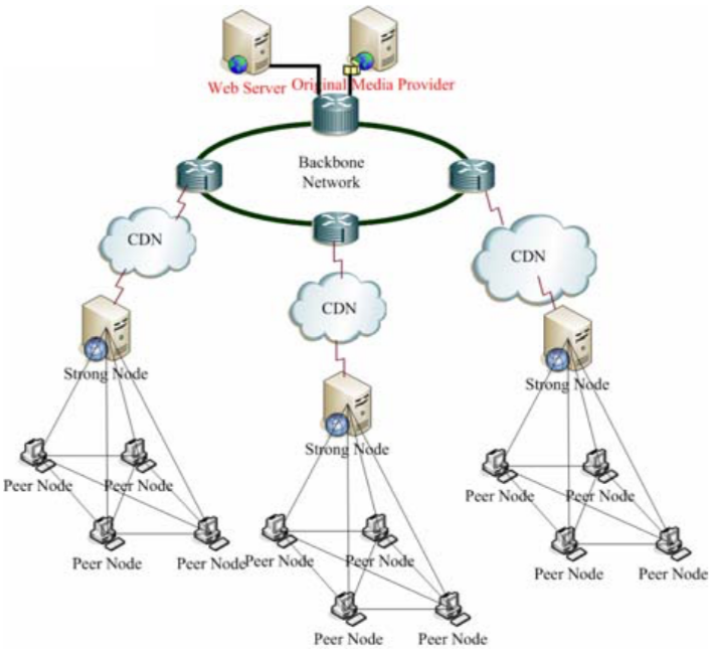

Figure 2.5: PeerCDN Hybrid Architecture

for an untrusted network of peers to assist in the distribution of content without the threat of content tampering, like in BitTorrent [16]. Hence, this data structures are immutable. Since files are identified by their hash, they are cache-friendly. IPFS exposes an API to interact with Merkle DAGs, allowing for reads and writes. This interface, the InterPlanetary Linked Data (IPLD) [20], focuses on bringing together all the hash-linked data structures (e.g. git, blockchains) under a unified JSON-based model. To prevent the system from being locked-in to a particular function or format, IPFS uses a Multiformats [21], self-describing formats that promote interoperability and protocol agility, multiformats are used for hashes, network addresses, cryptographic keys and more. IPFS creators, fully aware that to truly thrive, the protocol would have to be adaptable, have architectured the system so that this process would be facilitated. Thus, IPFS follows a modular approach to the software's development, enabling each small module to be replaced and adapted without impacting the overall system. Originally IPFS's networking stack, libp2p [22] is now an independent project that enables developers to build p2p applications through the libp2p's API. It enables applications to adapt to the heterogeneity of the web clients and their different requirements and resources, since not all of them have access to the same set of protocols. Libp2p enables peer look-up and dial and content discovery and transfer.

[20] https://ipld.io/
[21] https://github.com/multiformats/multiformats
[22] https://github.com/libp2p/libp2p

Chapter 3

Architecture

In this chapter we will describe the architectural elements of Startrail. Startrail is a pluggable and totally independent caching middleware that plugs into the IPFS Core, granting it with CDN-like capabilities.

IPFS offers the benefits of a structured peer-to-peer network, while also allowing for objects to be addressed by content. The all open-source IPFS stack is also fairly well documented and provides a good base to build up on. Because of this IPFS became a natural choice where to integrate Startrail.

The rest of the chapter is as follows, we'll start by describing the intendend use case in Section 3.1. Since the proposed solution is so intrinsically connected to IPFS, in Section 3.2 we'll outline the most relevant pieces of the system and data structures. On Section 3.3 the solution's architecture will be detailed, followed by an analysis of the caching algorithm on Section 3.4.

3.1 Use Case

We envisioned Startrail to be an adaptive network cache. One that continually moves content ever closer to a growing source of request. Hence, reducing, on average, the time it takes to access content on the network. It does so without requiring intermediate nodes to previously request such content. Thus, enabling smaller providers to serve bigger crowds. It should do so, in an interoperable manner. This means that nodes running Startrail should not depend on other nodes to be effective. Thus, it enables nodes to contribute to the network even when adoption is not absolute.

Startrail was, to some degree, inspired by Kademlia's caching system, illustrated on Figure 3.1(a). Here nodes that request for an object (step 1) and receive it (step 2) will ask the node closer to the one that served them the content to cache it temporarily (step 3). The Time To Live (TTL) associated to the cached record is smaller, the further away from the provider node the caching peer is. This mechanism, requires all nodes to implement the same interface. In the protocol, contrary to the latter, there should be no need for interaction between nodes for the cache to work. Startrail was designed so that nodes are totally independent and may contribute to the network without the need to implement a specific interface to communicate with other Startrail (or regular IPFS nodes) for the cache to function. For comparison, Startrail's caching system is also illustrated on Figure 3.1(b). Here, it is not up to the requesting node to ask the caching node to cache the content. It is the caching node inspects content discovery messages

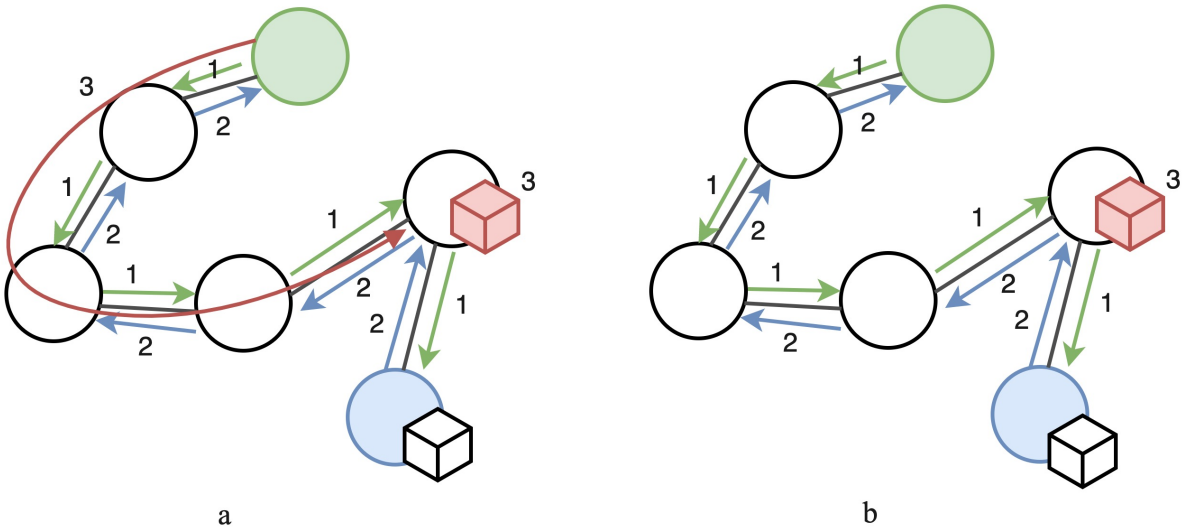

Figure 3.1: Comparison between the Kademlia and Startrail's caching systems

(in step 1) and continuously tracks the referenced object's popularity. If flagged popular he'll fetch and the serve the content himself.

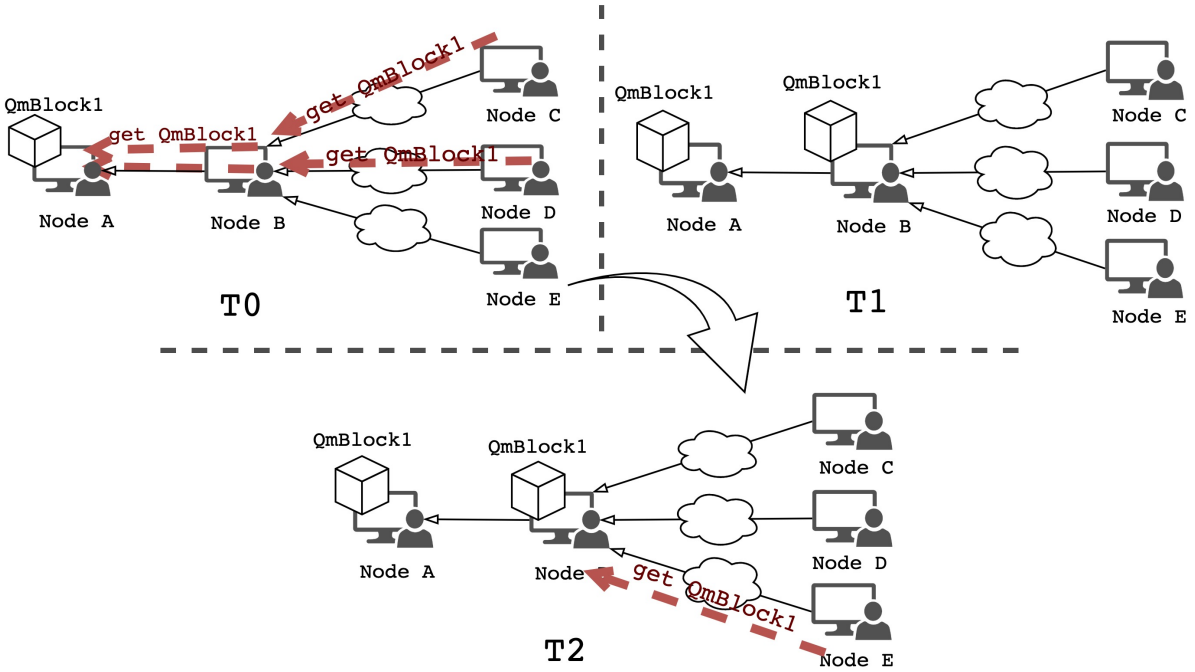

Figure 3.2: Illustration of the proposed Startrail flow

We shall now explore, in more detail, the intended behavior of the network. Figure 3.2 exposes the simplest scenario possible - a small portion of a network where all the nodes are running Startrail.

Here the content, in this case block *QmBlock1*, is stored on *Node A*. Nodes *C* and *D* request *QmBlock1* to the network. While doing so, *Node B* that is requested by both, detects that the Content Identifier (CID) is popular and flags it, fetching and caching the content itself. Later, when *Node E* request the for content, the response won't have to traverse the whole network, it may be fulfilled by *Node B*.

3.2 IPFS Architecture

Because Startrail is built on top of IPFS and integrates with some of its deep internals and mechanics it is then imperative that we thoroughly examine these.

The IPFS project follows the UNIX modular approach to software development. Hence, the codebase itself is segmented into smaller modules, each being responsible for a short and confined responsibility. This means that the project, as a whole, is made up of hundreds of small packages. In this Section we are going to select only the most relevant parts, the ones that our solution integrates with.

3.2.1 Objects

Objects on IPFS consist of Merkle DAGs of content-addressed immutable objects with links. With a construction similar but more general than a Merkle tree [7]. Deduplicated, these do not need to be balanced, and non-leaf nodes may contain data. Since they are addressed by content, Merkle DAGs grant tamper proof - changing the content, would change the address. They are used to represent arbitrary data-structures. These can represent, for example, hierarchies of files, or be used in communication systems. A visual representation of a Merkle DAG can be found on Figure 3.3.

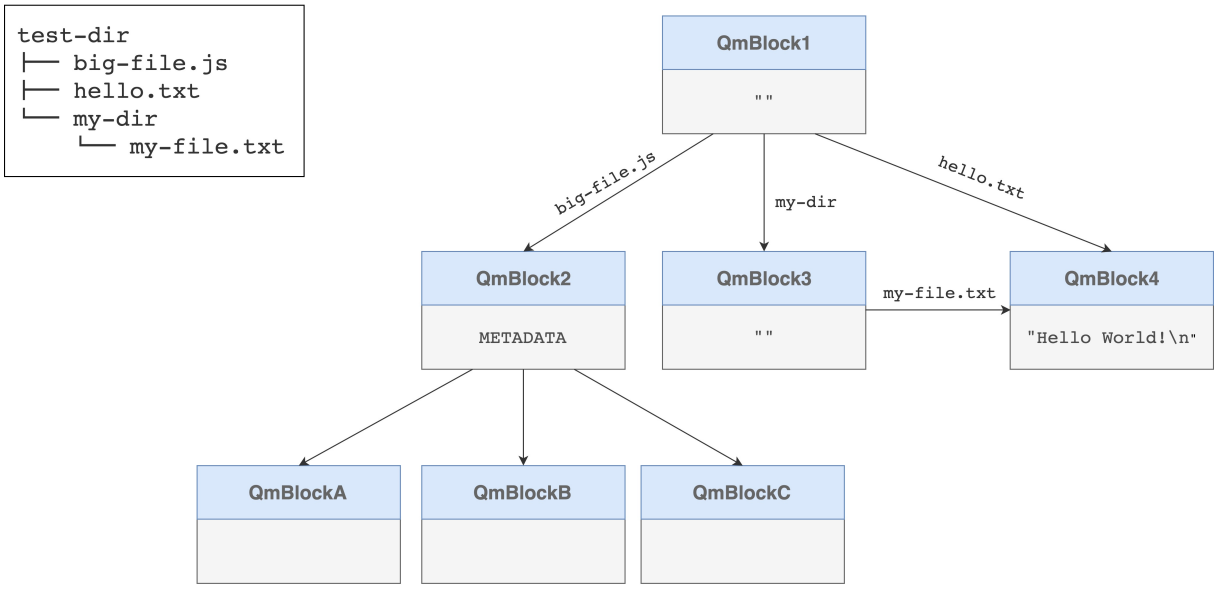

Figure 3.3: Merkle DAG representing a file structure with deduplication of data blocks

3.2.2 Core's Architecture

A high level overview of the architecture of the IPFS core is depicted in Figure 3.4. We shall delve into each of the illustrated components.

- **Core API** - The Application Programming Interface (API) exposed by the core, imported by both the Command-line Interface (CLI) and the HTTP API;
- **Repo** - The API responsible for abstracting the datastore or database technology (e.g. Memory,

Disk, S3[1]). It aims to enable datastore-agnostic development, allowing datastores to be swapped seamlessly;

- **Block** - The API used to manipulate raw IPFS blocks;
- **Files** - The API used for interacting with the File System;
  - **UnixFS** - The Unix Engine, implemented by the *Importer* and *Exporter* are responsible for the file layout and chunking mechanisms to import or export files from the network.
- **Bitswap** - Bitswap is the data trading module for IPFS. It manages requesting and sending blocks to and from other peers in the network. Bitswap has two main jobs:
  - to acquire blocks requested by the client from the network;
  - to judiciously send blocks in its possession to other peers who want them;
- **BlockService** - This is a content-addressable store for blocks, providing an API for adding, deleting, and retrieving blocks. This service is supported by the *Repo* and *Bitswap* APIs.
- **Libp2p** - This is a networking stack and modularized library that grew out of IPFS. It bundles a suite of tools that aim to support the development of large scale peer-to-peer systems. It addresses complicated p2p challenges like Discovery, Routing, Transport through many specifications, protocols and libraries. One of such libraries is the *kad-dht* module:
  - **Kad-DHT** - This is the module responsible for implementing the Kademlia DHT with the modifications proposed by S/Kademlia [36]. It has tools for peer discovery and content or peer routing.

3.2.3 Data Exchange

Data exchanges on IPFS are handled by *Bitswap*, a *BitTorrent* inspired protocol. *Bitswap* peers operate two data-structures:

- `want_list` - the set of blocks the node is looking to acquire;
- `have_list` - the set of blocks the node has to offer in exchange.

Bitswap is a message based protocol, as opposed to request-response. All messages contain `want_list` or blocks.

When the application wants to fetch new blocks through the *Bitswap* API, the `want_manager` will keep bundling new messages together; then sending them all together, reducing network congestion. Also, to propagate the `want_list` to newly discovered peers, every time a peer makes a new connection, *Bitswap* will send its `want_list`. Apart from this mechanism, peers will also periodically exchange their `want_list`s.

When searching for a block, *Bitswap* will first search the local *BlockService* for it. If not found, it will resort to the content routing module, in our case, the `kad-dht` module. The latter will run the following tasks:

[1] Simple Storage Service - On demand persistent storage service hosted Amazon Web Services

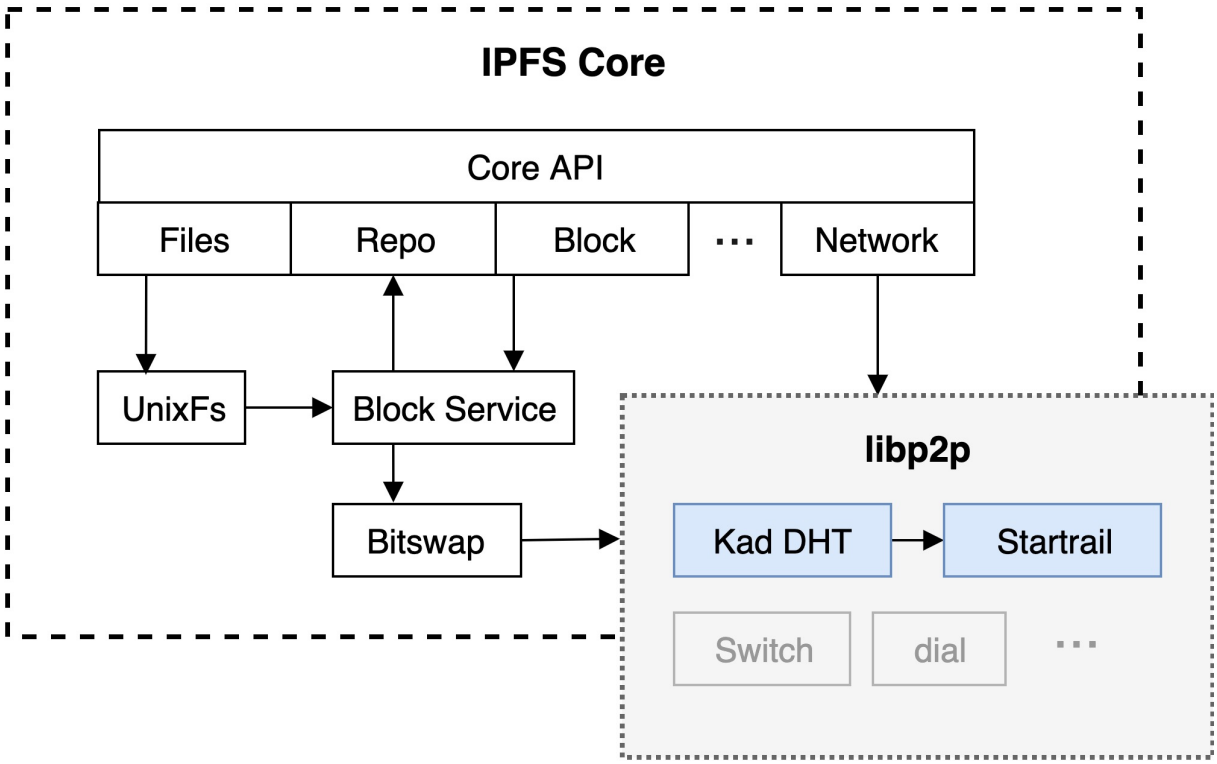

Figure 3.4: IPFS Core's Architecture

1. Query the local providers database for the know providers of a certain CID;
2. Query the DHT for providers of the target CID.

Following the acquisition of the group of potential providers, the node will connect and pass them its `want_list` containing the target CID.

The process can be further inspected on Figure 3.5.

When a node receives a `want_list` it should check which blocks it has from the list and consider sending the matching blocks to the requester.

When a node receives blocks that it asked for, the node should send out a notification called a `Cancel` to tell its peers that the node no longer wants those blocks.

3.3 Startrail's Architecture

Having a better grasp of the underlying system, we can proceed to understand how and where to integrate the Startrail cache. The first step is to identify where to tap into so that we are notified of new content requests. On IPFS we can do that through two different ways (at least, that we are aware):

- On `Kad-DHT`, checking for the CID associated to `GET_PROVIDER` messages;
- On *Bitswap*, listening for new `want_list` messages and tracking each of the included CIDs;

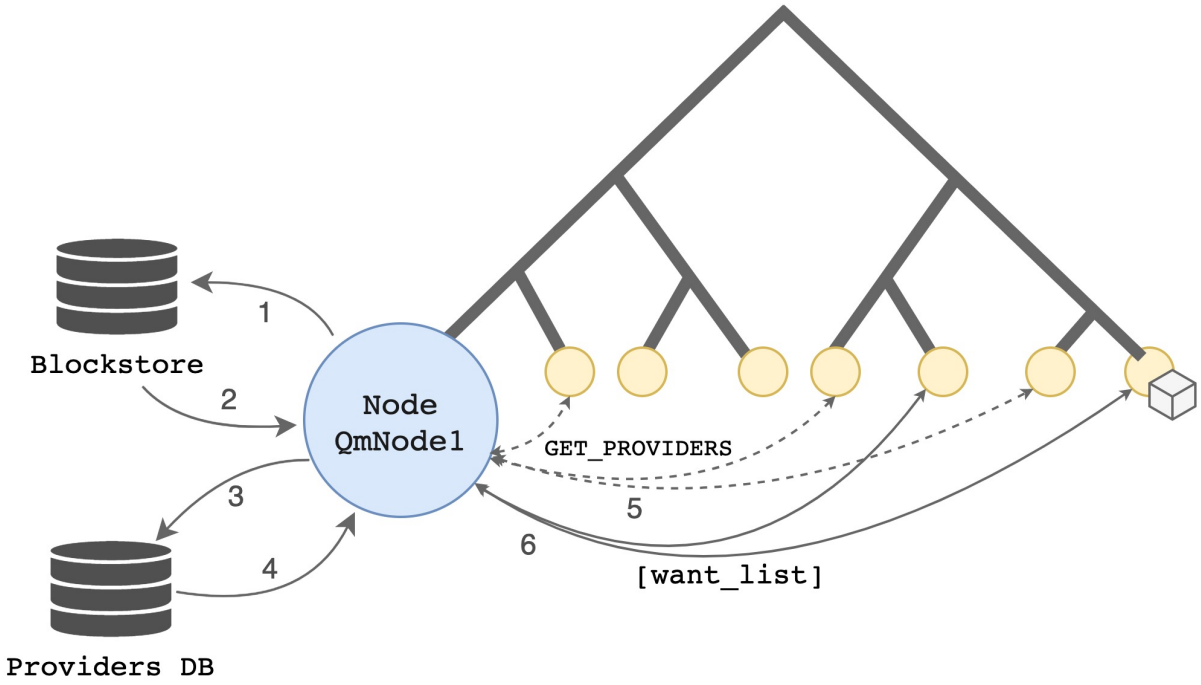

Figure 3.5: *Bitswap* and `kad-dht` interaction when fetching a block from the network

Our current implementation takes advantage of the former. Thus, our current implementation of Startrail does not take advantage of both methods and probing potential. We leave as future work implementation of the latter.

Startrail's main purpose is to recognize patterns in object accesses. To do so, it uses two separate components:

- The *Startrail Core*, that exposes the Startrail API. This is the interface other components will consume to work with the module;
- The *Popularity Manager*. The component responsible for tracking objects' popularity;

The *Startrail Core* integrates with the data trading module, *Bitswap*, the BlockService used to access the data storage and libp2p for several network utilities. The *Core*'s main responsibility is to orchestrate all these modules while integrating with data from the *Popularity Manager*.

The *Popularity Manager* tracks and updates objects popularity. It is totally configurable and it can operate with any specified caching strategy.

The class diagram of these components is defined on Figure 3.6

Further analysis of the *Core*'s 'diagram reveals the aforementioned integrations with external modules, including the internal popularity manager. It also reveals the two main exposed functions:

- `process(cid)` - responsible for triggering the orquestration and popularity calculation. Returns a `Boolean` for ease of integration with `kad-dht` module. The specific algorithm implemented can be seen in detail in Section 3.4
- `updateConfigs()` - used for configuration renovation. Once executed, it will fetch new configurations from the IPFS *Repo* and if changes are detected, will refresh the live ones. It is useful for

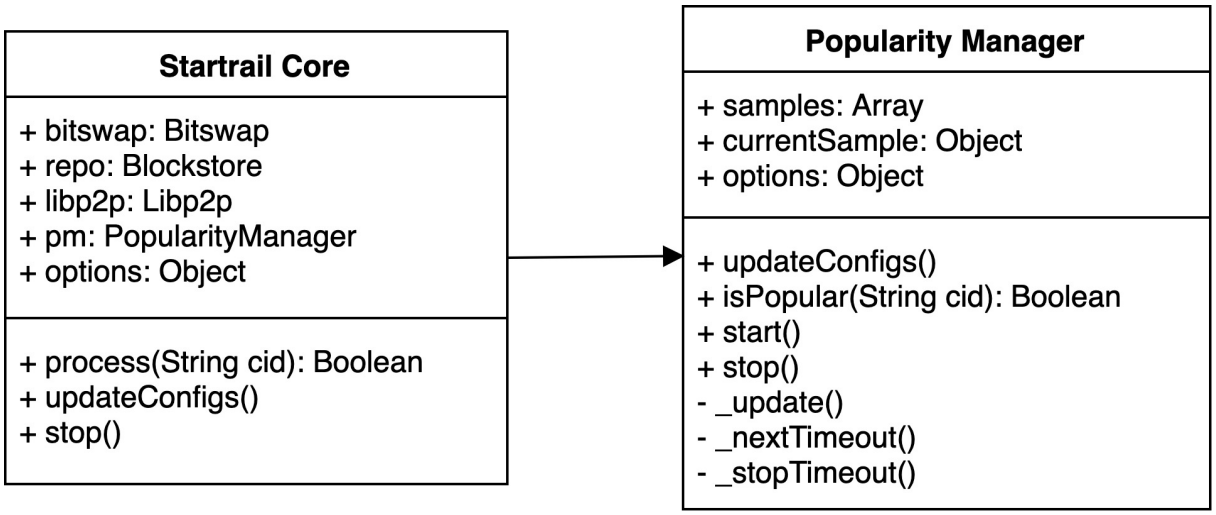

Figure 3.6: Class diagram of Startrail's Core and Popularity Manager

hotrealoading the configuration when testing.

Additionally, for the *Popularity Manager* class:

- `isPopular(cid)` - updates and calculates the updated popularity for any CID passed as argument. Returns a `Boolean` - true for popular objects, false otherwise. In Section 3.4 the specific calculation and tracking algorithm is explored in more detail;
- `updateConfigs()` - serves the same purpose as the above mentioned one;
- `start()` and `stop()` - methods for controlling the state of the sampling timer;
- `_nextTimeout()` - manages the sampling timer. Responsible for scheduling timeouts;
- `_update()` - runs every time the timeout pops. It pushes the current sample to the sampling history and a new one is created.

3.4 Algorithms

3.4.1 Message processing algorithm

In the previous section we've analyzed Startrail's architecture and pinpointed where it is ingesting data from. To recognize patterns in object accesses, Startrail examines the CIDs sent on `GET_PROVIDER` messages. Hence, we have to execute the `process()` function every time the `GET_PROVIDER` message handler is triggered. The complete execution flow, including the one inside the Startrail is illustrated on Figure 3.7.

Figure 3.7 unveils the execution flow starting when a peer requests a block from the network. This action triggers the search for providers on the network (as shown in 3.2.3). Upon receiving such message, the `kad-dht` handler will execute the Startrail `process` hook. Following the popularity update, either no further action is required, or the block is flagged popular and the peer will attempt to fetch it or

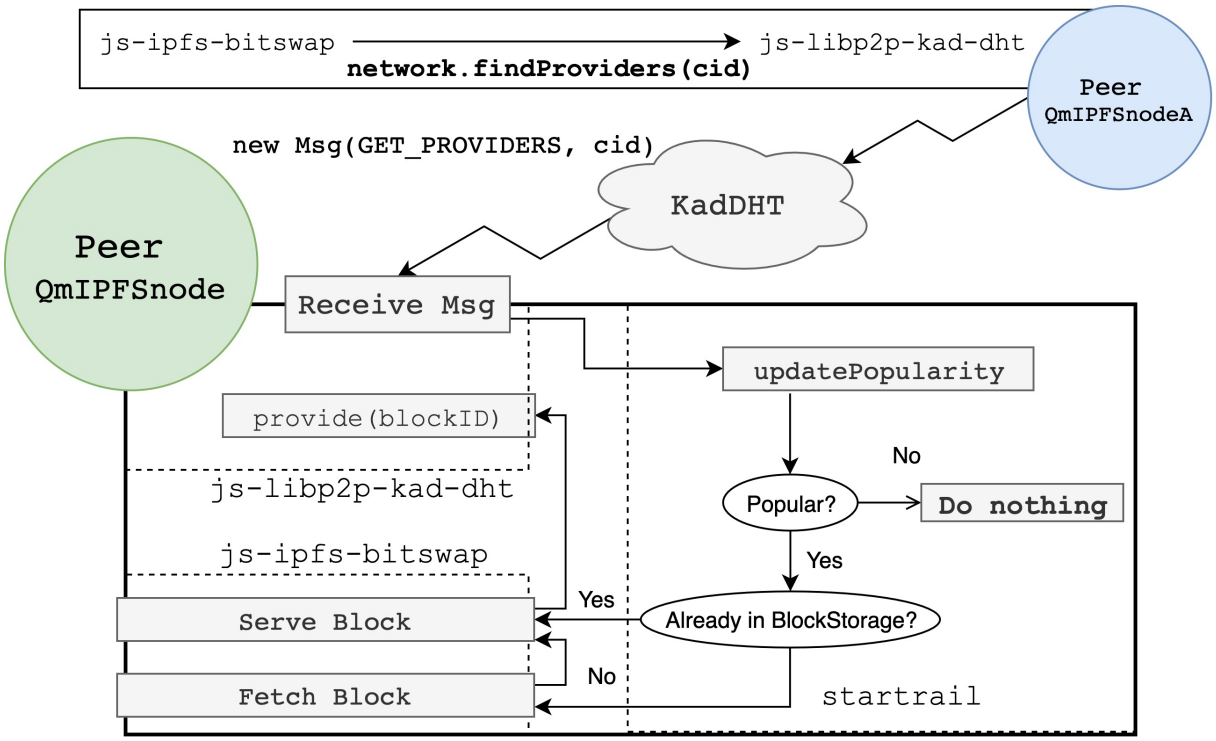

Figure 3.7: Execution flow inside the Startrail module

retrieve it from the BlockStorage. The block could potentially be found in the storage because, since we are using the IPFS BlockStorage, the block could have been previously fetched by either Startrail or the peer itself. Either way, subsequently to acquiring the block, the peer announces to the network that it is now providing it. The JavaScript pseudo-code can be further analyzed on *Code Listing* 3.1.

3.4.2 Popularity Calculation Algorithm

"The best way to predict the future is to study the past." – *Robert Kiyosaki, American Author*

The best way to predict the near future is to look into the recent past. The notion of content popularity is related to the amount of requests the network makes trying to retrieve it.. By studying the current and past popularity of a certain CID, we are able to likely forecast content that is going to, at least likely, remain popular in the future. Caching this locally and serving it to other peers has the benefit of making other nodes' accesses faster.

For simplicity our forecast takes into consideration only a small subset of the node's *past*. This subset, or window, can be obtained through various techniques. The one implemented in our solution is a hopping window. Here, sampling windows may overlap. This is desirable in our solution as we want to maintain some notion of continuity between samples. Meaning that an object that was popular in the window before, still has high probability to remain popular in the current one, since a portion of the data remains the same.

Although the parameters are totally configurable the ones set by default are **30 seconds** for window duration, with hops of **10 seconds**. The Popularity Manager implements the hopping window by dividing it into hop-sized samples. In our case we divide the total 30 second sampling window into three 10 second samples. A sample consists of a simple JavaScript Object, generally known as a `Map`.

```
async function process(cid) {
  if ( !isPopular(cid) ) {
    return;  // DO NOTHING
  }
  if (await blockstorage.has(cid)) {
    // Block found in blockstorage, serve it
    bitswap.serveBlock(cid)
  }
  // Block not found in blockstorage, get it ourselves
  if (bitswap.wantlist.contains(cid)) {
    // Do not get a block already on the wantlist
    return;
  }
  await bitswap.get(cid)
  // Announce to the network we are serving the block
  await libp2p.provide(cid);
  if ( repo.size() < 9Gb ) // 90\% of IPFS max default storage
  await pin(cid);
}
```

Listing 3.1: Startrail processing engine

Every time a new message is processed, the Startrail Core checks the popularity of the referenced object by running the `isPopular()` function. The function will keep track of objects it has seen in the current 10 second window; incrementing a counter every time the CID processed. Every 10 seconds the current window, or `sample` expires and is pushed onto a list that holds the previous ones. It is on this latter list of samples (`samples` in the class diagram from 3.3) that the popularity calculations are made. An illustration of the interaction between samples and block arrivals is represented on Figure 3.8.

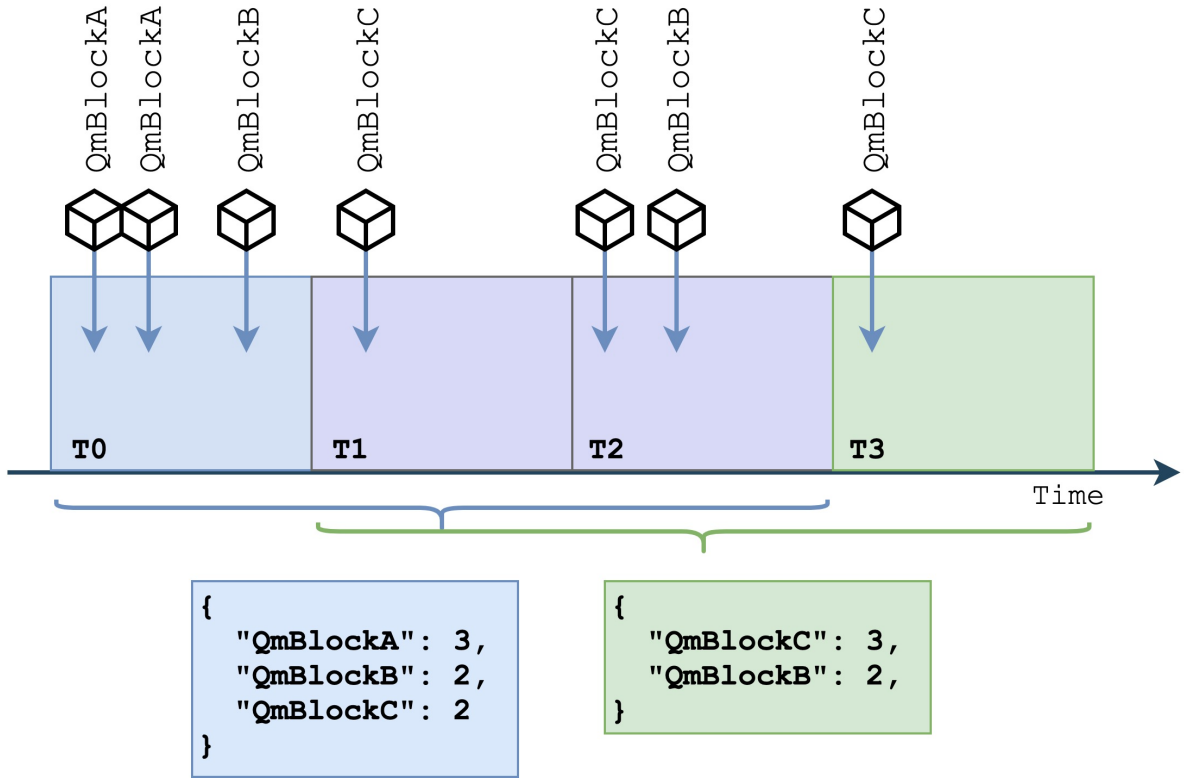

Figure 3.8: Interaction between new block arrivals and sampling windows

To calculate a block's popularity the Popularity Manager will first select the three most recent samples after concatenating the current one to this list. Next, will reduce the array outputting the total amount of times the object was observed. If bigger than a certain configurable threshold the object is considered popular. A more thorough inspection of the algorithm can be made on *Code Listing* 3.2.

```
const WINDOW_SIZE = 3
const CACHE_THRESHOLD = 2

class PopularityManager {
  constructor() {
    this.samples = [];
    this.currentSample = {};
  }
  isPopular(cid) {
    this.currentSample[cid] = this.currentSample[cid]
    ? this.currentSample[cid] + 1
    : 1;

    // Join last WINDOW_SIZE sample windows and calculate popularity
    const popularity = this.samples
    .concat([this.currentSample])
    .slice(-WINDOW_SIZE) // get last WINDOW_SIZE samples
    .reduce((popularity, sample) => {
      return (popularity += sample[cid]);
    }, 0);
    return popularity >= CACHE_THRESHOLD;
  }
}
```

Listing 3.2: Popularity Manager core

Caching Heuristic

In Startrail we employ our own heuristic algorithm for tracking objects popularity. Our algorithm sums the number of times an object was processed in the 30 second window and if bigger than the **default threshold of 2** it is flagged for caching.

This heuristic has the benefits of (i) being fairly simple to implement and compute; it also (ii) reacts quickly to changes in content access trends. It can be considered rather optimistic, since spotting the same object twice will consider it popular. However, we should take into account that we are not listening on the actual block requests, we are listening on the discovery requests. As analyzed on Section 3.2.3, the discovery messages are only sent when the amount of known possible providers for a CID is not big enough.

Due to its simplicity the heuristic has a limitation. Because the metric is statically defined it lacks the ability to adapt to different volumes of traffic and traffic conditions. Hence, a misconfigured threshold could lead the node into caching too little or too much data.

To further be able to adapt to the node's environment the metric can be dynamically calculated and factor in the node's variable conditions. Observing the same object twice will yield different popularity values for a node that processes 1 block a second and a completely different one for another that processes 100 blocks/second.

One alternative is to, instead of flagging objects based on the total amount of times they were processed, one could identify a block as popular based on the percentage of the overall traffic the node processed. In a mathematical expression:

$$\frac{\#\ times\ CID\ was\ processed}{\#\ total\ CIDs\ processed} \geq \%\ Cache\ Threshold$$

But in this case, even if we got the threshold heuristic close to optimal, that would, in practice, limit the amount of blocks the peer would be able to cache, i.e. considering the threshold was, for example, 1%, the node would be limited to caching 100 blocks when, in ideal conditions, each block is processed 1% of the times.

Our heuristic does not take into account the size of the content being cached. This was a conscious decision, the reasoning for it is that on IPFS most blocks have the maximum default size of 256Kb. Usually only the last one of the sequence that makes up a file is less than that. Hence, we despised the block size as parameter for caching heuristic.

Cache Maintenance

Contrary to most caching systems [37], where content is cached by default leaving the cache replacement algorithm responsible for releasing less relevant documents to create space for new ones. Startrail employs an heuristic to judge which objects should be cached in first place. Nevertheless the cache does not grow indefinitely as IPFS performs block replacement.

Startrail, for the most part, works by leveraging the internal IPFS mechanics. This ensures the component is lightweight and uses the same procedures as the rest of the system. Startrail also shares the same block datastore as the rest of IPFS. Thus, the blocks that the peer fetches through regular utilisation and the ones fetched by Startrail are all kept in the same storage. With no way of discriminating the origin that fetched the block it's impossible for a cache replacement algorithm to manage only the cached resources. Even if a list of the cached blocks CIDs was kept as a way to differentiate them, a block may be fetched by both the regular IPFS and Startrail, meaning that in this situation, evicting a block from cache would, potentially, remove used resources.

Hence on Startrail, we allow the node to utilise the full amount of allocated storage by IPFS which defaults to **10Gb**. Once the node fills up this space it's up to the IPFS Garbage Collector (GC) to discard unnecessary objects. The IPFS GC removes the non-pinned objects. Hence, to prevent popular blocks from being collected when the it executes, we pin the popular objects. When a block stops being popular it is unpinned, leaving it at the mercy of the GC. When the threshold of 90% of IPFS' storage is reached blocks are no longer pinned in order to leave room for new blocks.

3.5 Summary

In this Chapter we started by describing the way Startrail should approximate content from the source of requests. We then thoroughly analysed IPFS' base architecture, where Startrail is built on top of, and also its main mechanisms. We finalized this chapter with the extended description of the proposed solution's architecture and its relevant algorithms.

Chapter 4

Evaluation

This chapter describes in detail the evaluation of a realistic deployment of a Startrail network, accessing its performance and implications. We'll then analyze these results, in light of our initial expectations and to the results obtained by running an unmodified IPFS network through similar conditions.

We will start by describing the platform developed to execute the simulations in Section 4.1. Next, we analyze the metrics and results we are looking to assess in Section 4.2. In Section 4.3 we'll follow with an analysis of the testing setup and adjustable network conditions. Section 4.4 will outline the results obtained according to the described metrics. Additionally, we will also analyze, in Section 4.5, the impact different percentages of Startrail adoption can have in the overall network performance.

4.1 The Testbed

Although not planned from the beginning, there was a considerable amount of effort put into developing a testbed capable of simulating a realistic network. For our specific testbed we were looking for a solution that could fulfill the following requirements:

1. Enable us to seamlessly adjust and change network conditions, e.g. latency, jitter;
2. Provide a platform for gathering and monitoring a diverse array of metrics and logs;
3. Scale well as more computing power is added to the testbed;
4. Effortlessly enable us to orchestrate and coordinate peers in the networks, i.e execute commands;
5. Allow for effortless integration with the codebase. Excluding possible alternatives like PeerSim [38], as this would require porting the whole protocol to the Java API.

4.1.1 Testbed Architecture

To develop such a solution we resorted to our professional experience in deploying and monitoring large scale micro-services as source of inspiration.

To allow for easy integration of new computing power into the testbed and management of deployments, we resorted to Kubernetes (K8s) [39]. Kubernetes is a tool for deploying and managing containerized applications. It handles the discoverability and liveness of the deployed workloads. Our Kubernetes cluster was deployed on Google Cloud, on the managed Google Kubernetes Engine, for convenience.

The smallest computing unit on K8s is the *Pod*. The *Pod* is a formation of containers, in our case Docker [40] containers. This enabled us to build automated Continuous Integration (CI) pipelines that would build the containers, testing them afterwards. This granted us that only working artifacts, would be deployed to the cluster.

Since Kubernetes *Pods* allows us to create any arbitrary composition of containers, we took advantage of this and made sure to inject, alongside every IPFS node, a *Toxiproxy* [1]sidecar - a small proxy - from which we channel all IPFS traffic through, coming in and out from the node. This allows us to inject network variability into IPFS connections and simulate diverse network conditions. Such *Pod* architecture is illustrated in Figure 4.1.

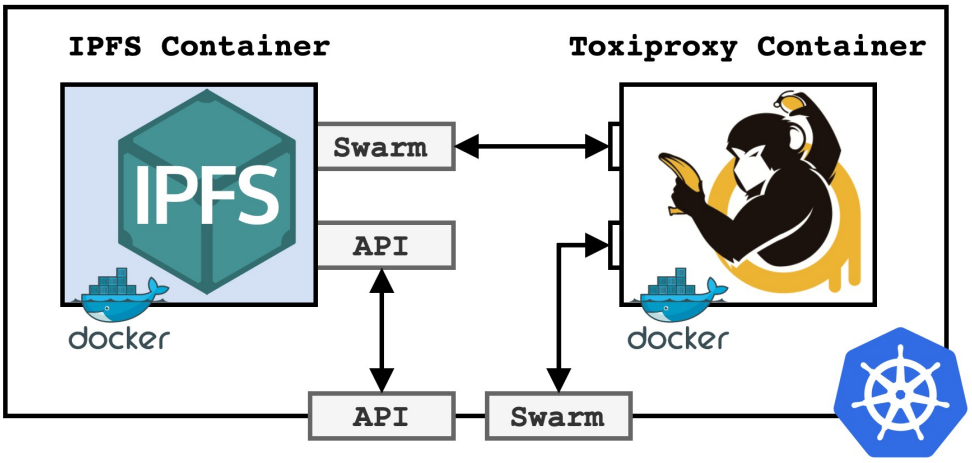

Figure 4.1: Composition of the Startrail Testbed Kubernetes Pod

Further examination of Figure 4.1 reveals the piping of the IPFS service port, the *swarm* endpoint, through *Toxiproxy*. The peer will then announce the *Toxiproxy* endpoint as its dial address. The IPFS API connection is not proxied since it doesn't require being modified.

We wanted to make the simulations easily reproducible and preferably software defined, as configuration files. Hence, we leveraged Helm [2], a tool that helps us release and manage Kubernetes applications. This enabled us to create different node configurations, *Charts*, e.g. for provider nodes and consumer nodes. Helm also made easy dynamically configuring IPFS containers, including setting test dependent Startrail options.

One critical requirement of the testbed is to be able to gather data from different layers of the system during simulations. We accomplished such requirement in our solution by implementing the ELK Stack [3]suite of tools. The ELK stack is comprised of *Elasticsearch* for log indexing and searching engine, *Logstash* for transformation and *Kibana* for log and metric visualization. This way, Docker containers will log to standard out and have their logs sent to *Logstash*, which will filter them and send them to *Elasticsearch* to be stored. *Kibana* would then consume this data on demand.

[1]https://github.com/Shopify/toxiproxy
[2]https://helm.sh
[3]https://www.elastic.co/what-is/elk-stack

The above described architecture is depicted on Figure 4.2.

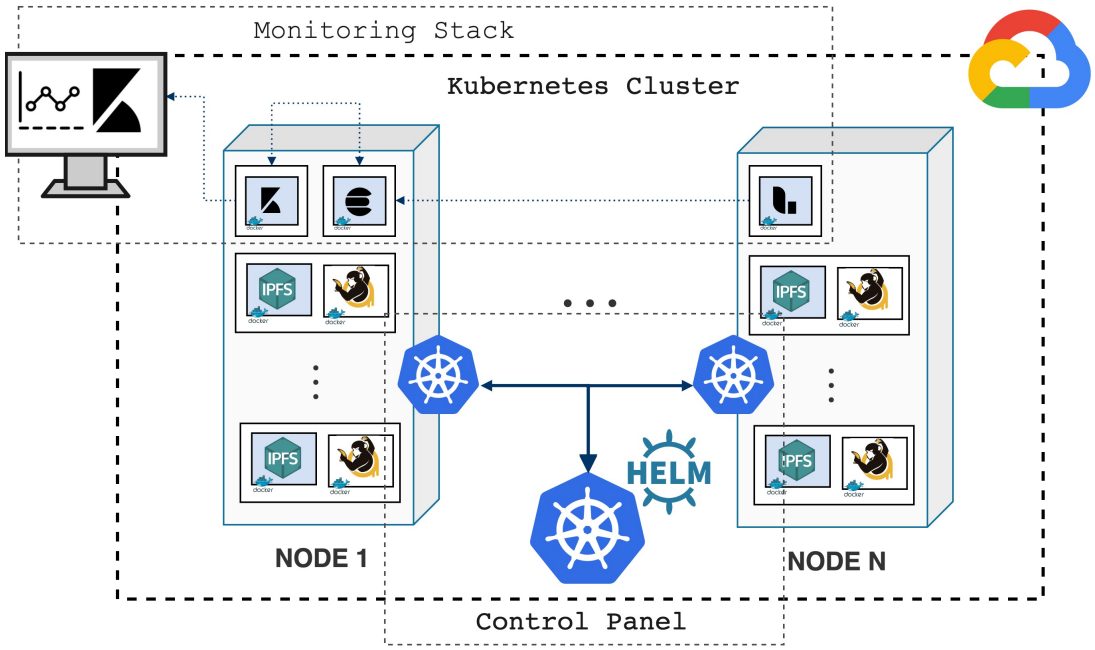

Figure 4.2: Architecture overview of the testbed network

4.1.2 Deploying a network

We tried to make the deploy process the simplest and automated as much as possible since we were aware that the network would have to be deployed many times to simulate different network conditions. Hence, we resorted to Unix's Makefiles to automate the network setup.

The deployment process, illustrated in Figure 4.3, is as follows:

1. Setup Bootstrap nodes. On IPFS to setup a network we first need to setup the bootstrap peers. These are used by other nodes as *Rendezvous Point* to join the network.

2. Create rest of nodes. Additionally, using Helm's ability to dynamically configure releases, we need to point these new nodes to the already setup Bootstrap ones.

3. Deploy Provider nodes. These are nodes preloaded with data. For these, as datasets were sometimes of considerate dimensions, datasets were downloaded onto the *Pod* from an S3 Bucket before the starting the container.

4.1.3 Interacting with the network

Having the network setup and the nodes able to connect and fetch data from each other and also having the ability to shutdown and bootstrap everything back together in minutes, provided us with solid ground from where to ran test from.

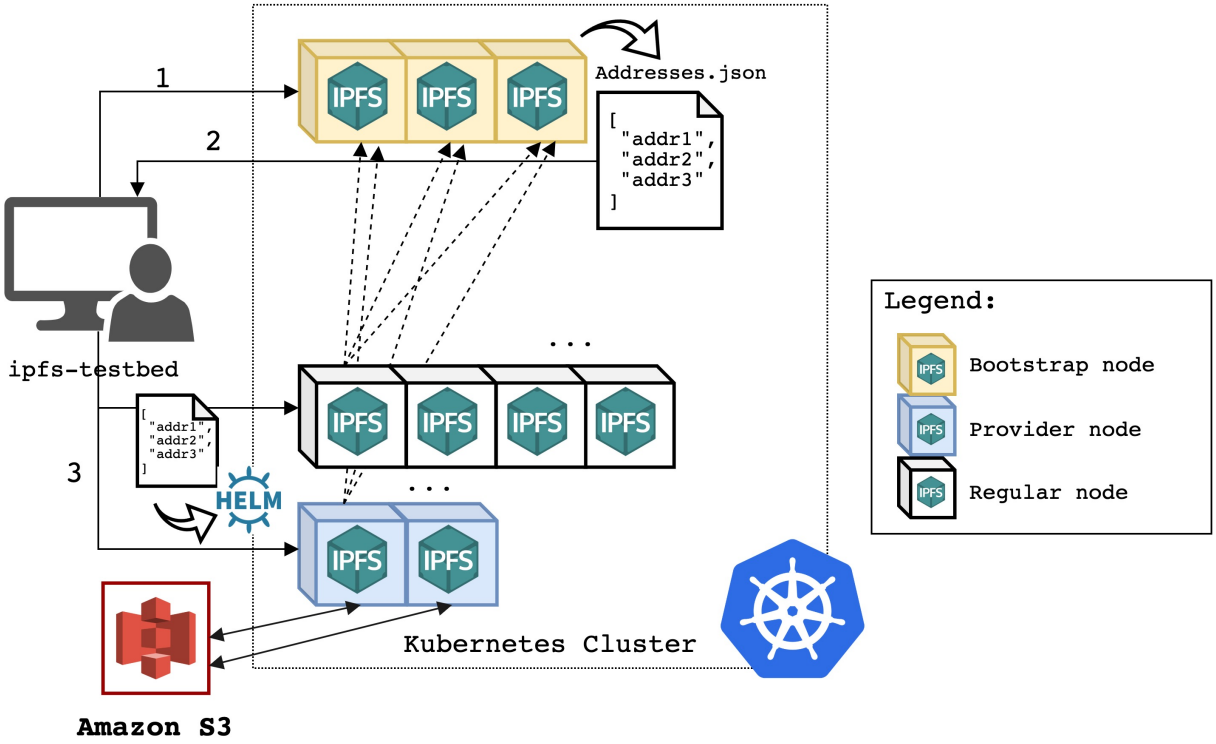

Figure 4.3: Deployment process on a new network on the Startrail testbed

To execute our tests, we developed our own solution that utilizes the IPFS and *Toxiproxy* APIs to orchestrate the peers and change the conditions of the network. The classes diagram of the CLI tool designed to interact with the network is depicted on Figure 4.4.

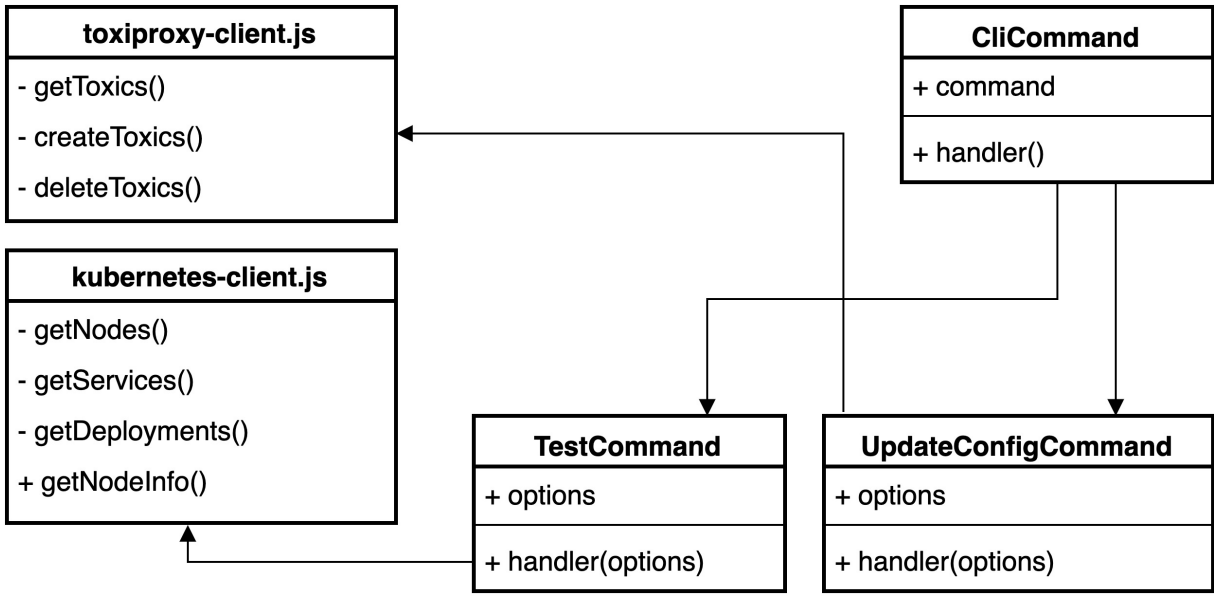

Figure 4.4: Classes diagram of the testbed CLI tool

The CLI tool implements a client for interacting with the Kubernetes API. This is crucial for fetching all the IPFS pods and the addresses they are running at, it is through this API that we are able to get the endpoints for the IPFS Swarm and API, as well as *Toxiproxy* API, with which we'd interact through its own client. These clients were then brought together in the implementation of the test and configuration commands.

With the network in place and a tool with which to execute tests from, it was possible to execute the multiple Startrail tests. We would execute the tests from our local machine that would orchestrate commands to each individual peer according to the scripted test file. Using the monitoring platform we would then gather live data from the nodes.

An illustration of the interaction can be examined in Figure 4.5

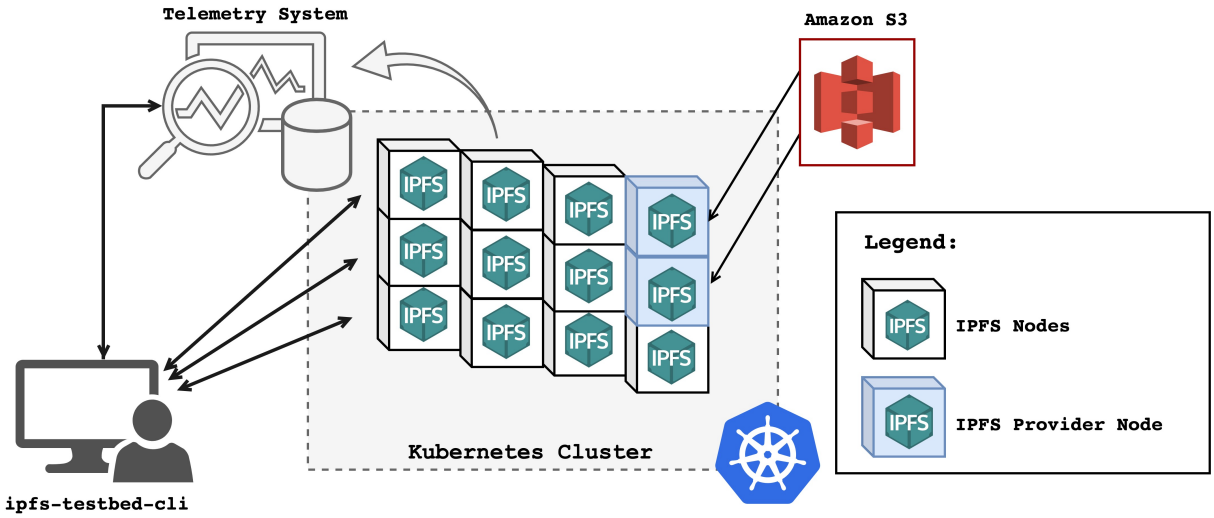

Figure 4.5: Testbed-cli orchestrating and monitoring Statrail testing cluster

4.2 Relevant Metrics

The metrics considered relevant for evaluating Startrail's network performance are:

- **Request Duration** - The duration of network request is inherently the most important metric to analyze Startrail's impact on the system. It will assess if caching mechanism is working and how effective it is. Hence, to measure this we should analyze the 95th Percentile request duration.
- **Memory usage** - Considering that Startrail Nodes are caching content we want to analyze how much additional data each node has to store.
- **Network usage** - We want to assess how much each node has to resort to the network in order to fetch content. Hence, we measure the volume of traffic each peer sends to the network.

For these metrics we will calculate the 95th Percentile (95P) and use for our analysis. The percentiles have the particularity of excluding the extreme values from the average calculation, the possible *outliers*, and the 95th provides a relevant assessment of the global performance of a system not subject to stringent SLAs.

4.3 Testing Setup

The implementation network used to execute the simulations was deployed in the aforementioned Kubernetes cluster running on Google Cloud. This cluster comprised of three *n1-standard-4* nodes. These are a general-purpose type of machine with 4 vCPUs, 15 GB of memory and 128 GB of storage. With this setup we were able to simulate a network of 100 peers. Of these 100, 5 were bootstrap nodes and 2 were provider nodes, that were preloaded with data. The dataset used was 200Mb. The size of the dataset was limited to such size due to the restrictions in memory each container could utilize without starving the whole cluster.

The amount of nodes simulated were limited by the resources available in the cluster as we were running on a limited budget. This is particularly important because although we are simulating network congestion, if the simulation becomes CPU bounded, we will have the peers struggling to get CPU time.

The simulations performed aim to test different access and network conditions. Each dataset is comprised of a known list of blocks (in the order of thousand blocks). For the duration of the test, we would from time to time, select a block from the list according to the specified distribution function and order the node to fetch it from the network. This block selection function was responsible for simulating the different access patterns, as we varied the function that selected the blocks. The different accesses we tried to simulate are:

- **Random Access** - The random access picks each block with equal probability. This pattern would serves as control, a basis for comparison, as it is not realistic;
- **Pareto Random** - The Pareto Distribution [41] is a power-law probability distribution. It serves as an adequate approximation for various observable phenomena. Internet objects popularity can also be approximated through such distribution [42]. To achieve close to the desired distribution we used an α equal to 0.3, meaning that 20% of the blocks generate 80% of the overall network traffic;
- **File Random** - Here we divided the total list of blocks into smaller lists, each amounting to 3Mb in block data. Selecting on of these lists means that the peer would fetch all the nodes in the list, thus simulating file access. To pick a random file from the list, we will also use a Pareto distribution.

Table 4.1 encompasses all the test scenarios simulated. Each of these had an induced latency of 100ms, ran for 10 minutes and each node requested a new block every 30seconds.

The parameter columns are:

- **Startrail** - defines if Startrail was running on all the nodes in the network;
- **Access Type** - indicates which of the previously mentioned access patterns we are simulating;
- **Latency** - expresses the amount of latency introduced by *Toxiproxy*;
- **Req. Freq.** - specifies the frequency at which each requests for blocks - or array of blocks, in the case of File Random - are made by the individual nodes;
- **Duration** - indicates for how long we ran the simulation;
- **Window** and **Threshold** - specify the used Startrail configuration, if applicable. *Window* being the amount of samples and the duration of each, in seconds; and *Threshold* the cache threshold at which Startrail will cache content.

Test Name	Startrail	Access Type	Window Size	Threshold
Random no Startrail	False	Random	N.A.	N.A.
Random w/ Startrail	True	Random	3*10sec	2
Pareto no Startrail	False	Pareto Random	N.A.	N.A.
Pareto w/ Startrail	True	Pareto Random	3*10sec	2
File Random no Startrail	False	File Random	N.A.	N.A.
File Random w/ Startrail	True	File Random	3*10sec	2

Table 4.1: Different testing condition for running network tests

4.4 Tests Results

After running the above-mentioned simulations we compiled the most relevant of obtained results into a set of graphs which we are going to analyze next.

Latency Analysis

The graph on Figure 4.6 exposes the calculated 95P Request duration for each simulation. The simulation of the random access running on a regular IPFS nodes' network, on the far left, yielded a P95 latency of 60 seconds. The same simulation running on the Startrail network only did 40 seconds. This is a considerable reduction of one third. Similar gains in speed can be observed for the tests with Pareto distribution access pattern. While at first one would think that this would be the test where the impact of Startrail would be the most evident, because the blocks would reach the cache threshold easier, in this case, however, since the same blocks are being requested more often, different peers on the network also serve the content because they previously downloaded it. Hence, the difference remains the same. For the file access type the proportion of gains remains similar, with the overall latency going up since now we are requesting a lot more of different blocks.

Memory Consumption Analysis

The Graph on Figure 4.7 compares the memory cost of running the simulations on a network with and without Startrail. Analysis of the graph reveals that running the simulations without Startrail costs generally the same, with only slight variations proportional to the amount of different blocks requested.

For the simulations ran on the Startrail network we observe a growth in memory utilization. This is expected since nodes are now storing more content, the cached blocks. Here, one interesting result stands out. For the Pareto access pattern we notice an increase in memory consumption relative to the all random one. This was not expected as the smaller diversity of blocks requested would mean less content being cached. This was not the case. One possible explanation we find that supports such results is that in this simulation a smaller subset of the dataset is now being constantly requested, meaning that, although smaller, we are guaranteeing that this subset will reach the cache threshold and

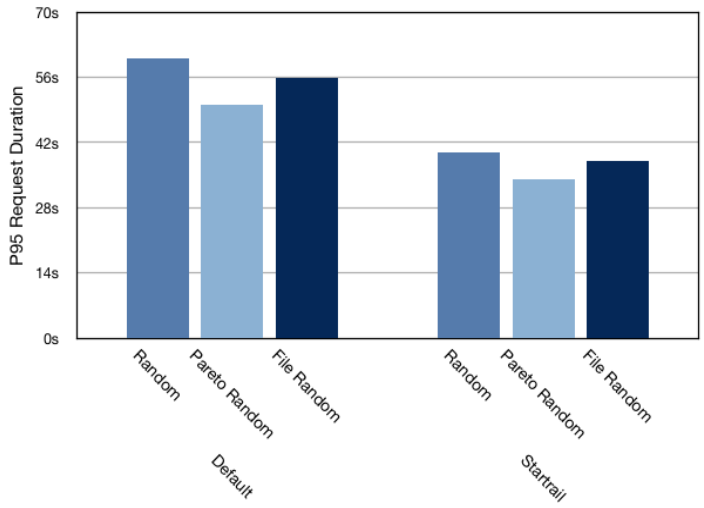

Figure 4.6: 95th percentile of request duration for the different testing scenarios

be stored. Because it doesn't grow significantly and past the IPFS default storage of 10Gb, the garbage collector never triggers and thus the cached blocks are kept for the whole duration of the test. This does not happen with the random access pattern because, since it follows a uniform distribution, requests for the same block may be scattered in time and consequently some never reach the threshold.

We can analyze Startrail's impact on the memory consumption of the node using the following expression:

$$\frac{Total\ observed\ memory\ consumption}{Regular\ Nodes\ memory\ consumption} \times 100$$

This expression yields the percentage increase in memory consumption when comparing the values obtained from running the simulation on the Startrail network with the ones obtained from running on the regular IPFS network. By computing the earlier expression for each of the simulations running on different networks, we can observe that the increases in memory consumption varies between 15% and 25%.

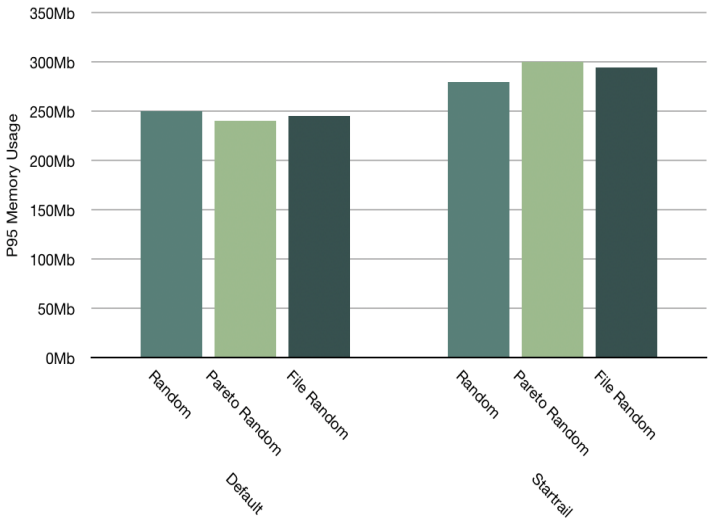

Figure 4.7: 95th Percentile of memory usage on the different testing scenarios

Network Consumption Analysis

The Graph on Figure 4.8 illustrates the network impact of running network simulations with and without Startrail. Further comparison of the obtained results reveals that the savings in network traffic (Mbs) are proportional to the speed ups in request latency. This happens because requests are being served closer to their source by caching nodes and thus fewer messages have to transit the network. This result wasn't totally expected, since while Startrail nodes serve content closer to their source, which reduces traffic, in order to do so they also need to fetch the content first for themselves, which generates traffic as well.

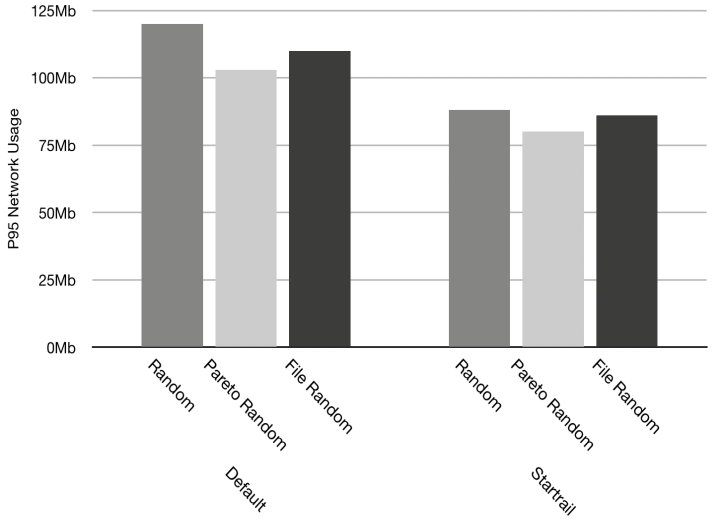

Figure 4.8: 95th Percentile of network usage on the different testing scenarios

4.5 Variable Startrail percentages

Additionally, we also assess the impact of the percentage of Startrail nodes in the network, to confirm that the benefits of Startrail are relevant even if a smaller percentage of nodes are contributing to caching or if, on the other hand, there was an percentage of Startrail participation in IPFS that would yield the optimal performance.

In order to evaluate which of the above mentioned propositions were true we ran the simulations described on Table 4.2. The conditions are similar to the ones presented earlier, in each of the tests was induced 100ms of latency, the tests ran for 10 minutes and each node requested a new block every 30seconds. The distinction between the ones described before and these is that in the latter we alternated the percentage of nodes running Startrail that were deployed on the network.

The results obtained from running the simulations were compiled into the graph in Figure 4.9. The graph's samples start on the far left with higher values of request latency for no Startrail participation on the network, and decreases nearly linearly as the percentage of Startrail nodes increases.

The impact of Startrail nodes' percentage on the network can be approximated through the linear regression drawn on the dotted line in the graph. This supports our initial proposition that the performance improves as the percentage of Startrail nodes increases. Nevertheless there is a slightly higher slope in

Test Name	Startrail Percentage	Access Type	Latency	Req Freq	Duration	Window Size	Threshold
Random No Startrail	**0%**	Random	100ms	30sec	10min	3*10sec	2
Random 30% Startrail	**30%**	Random	100ms	30sec	10min	3*10sec	2
Random 50% Startrail	**50%**	Random	100ms	30sec	10min	3*10sec	2
Random 80% Startrail	**80%**	Random	100ms	30sec	10min	3*10sec	2
Random 100% Startrail	**100%**	Random	100ms	30sec	10min	3*10sec	2

Table 4.2: Testing conditions for the different percentage of Startrail nodes in the network

the 30% to 50% straight, that however does not allow any conclusions of an optimal point to be taken due to experimental noise. This finding would have to be further assessed with more testing.

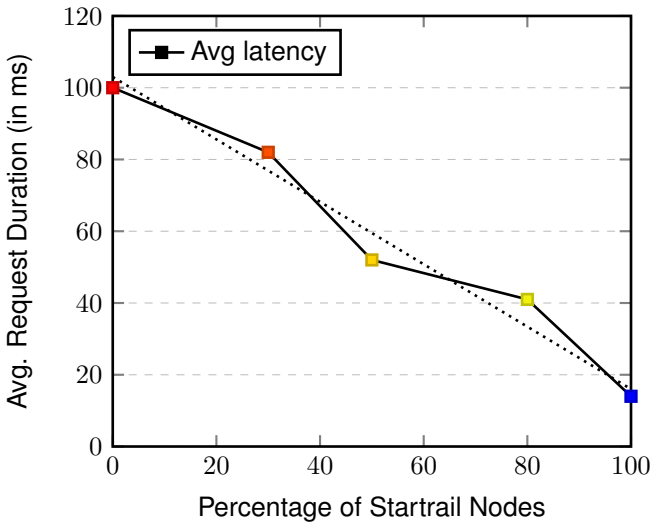

Figure 4.9: Avg request duration vs. Startrail nodes percentage

4.6 Summary

In this Chapter we started by thoroughly analysing the Cloud Deployment Infrastructure that allowed us to quickly setup the test network and served as testbed for the execution of the Startrail tests. Next, we described the setup used to run the simulations, the tests executed and also the specifications of the cluster and datasets. We followed that section with an analysis of the results obtained while running the simulations. We finished with the additional analysis of the impact that different percentages of Startrail nodes have on the overall network performance.

Chapter 5

Conclusion

In this thesis we proposed a solution that extends the existing IPFS and improves it as a content sharing system and its ability to distribute such content using a novel technique in the peer-to-peer file sharing systems' realm.

5.1 Concluding remarks

We started this dissertation by analyzing the relevant Content Sharing systems and by describing their compositions. We followed that by classifying the significant Content Distribution Systems' features. Next we then explored the technologies that enable peer-to-peer systems to exist inside a web browser and how to leverage them.

Having inspected the relevant components towards a design enabling a distributed, peer-to-peer CDN, we described the architecture of the InterPlanetary File System and its key mechanics. After identifying the key architectural elements and functionality of IPFS, and its shortcomings, we proposed Startrail, an extension caching component and described it thoroughly. We defined the solution's requirements and documented its implementation, data structures used and processes, as well as integration points with IPFS. We then analyzed the implementation of the system for containerized network deployments along with its architecture.

The platform used for simulating the network, including the setup conditions were then described, along with the test executed. The obtained results show that a network running Startrail nodes is able to perform better than one running only IPFS nodes. Startrail reduces the request latency by 30%, at the cost of small increase in total memory consumption of 20% while also reducing bandwidth utilization by around 25%.

Additionally, we assessed the impact that different percentages of Startrail nodes have in the overall network performance, as they can seamless coexist with non-Startrail plain IPFS nodes within the same network. The results, confirm the expectation that there is an inverse relation between Startrail nodes percentage and network latency. When one increases, then other is reduced.

5.2 Future work

Although the results are positive and bring improvements to IPFS's operation, there are potential further advances to be implemented. Startrail enables the development of additional enhancement features. Bellow we enumerate and describe some of the aspects that could be further explored:

1. **Broader probing potential** - Startrail takes advantage of a single request popularity probe, the discovery messages. In order to achieve a broader probing potential an improvement could be made allowing the system to further analyze requests on the received *want_list*s;
2. **Design a dynamic caching heuristic** - Caching thresholds on Startrail are static which can lead to caching imbalances when under high stress. One very interesting research topic would develop an heuristic that would dynamically adapt to the amount of requests the node processes;
3. **Prefetching** is the process of requesting content before it is actually necessary with with the expectation that it will be eventually requested, and thus being able to mask its retrieval time significantly.. Prefetching on IPFS can be implemented at system level or at application level. Startrail is a key crucial enabler for its operation. Startrail makes it possible for the prefetching mechanism to have a reduced impact on the network by leveraging the caching, which, instead of stressing the network, has the effect of setting up the network caches for traffic to come.

Bibliography

[1] G. Schneider, J. Evans, and K. Pinard, "The Internet - Illustrated," in *The Internet - Illustrated*, 2009.

[2] A. Bhushan, "File transfer protocol," RFC 114, RFC Editor, April 1971.

[3] J. Melvin and R. Watson, "First cut at a proposed telnet protocol," RFC 97, RFC Editor, February 1971.

[4] J. Benet, "IPFS -Content Addressed, Versioned, P2P File System," *arXiv preprint arXiv:1407.3561*, 2014.

[5] P. Teixeira, H. Sanjuan, and P. Samuli, "Merkle-CRDTs (DRAFT)," pp. 1–26, 2019.

[6] "Merkle-dags," tech. rep., Protocol Labs.

[7] R. C. Merkle, "A digital signature based on a conventional encryption function," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 293 LNCS, pp. 369–378, 1988.

[8] T. D. Thanh, S. Mohan, E. Choi, K. SangBum, and P. Kim, "A taxonomy and survey on distributed file systems," *Proceedings - 4th International Conference on Networked Computing and Advanced Information Management, NCM 2008*, vol. 1, pp. 144–149, 2008.

[9] Goldberg, Sandberg, Kleiman, Walsh, and B. Lyon, "Design and Implementation of the SUN Network Filesystem," *Usenix*, 1985.

[10] "The Andrew File System (AFS),"

[11] Microsoft, "Distributed File System (DFS): Referral Protocol," 1996.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, p. 29, 2003.

[13] R. Rodrigues and P. Druschel, "Peer-to-peer systems Survey," *Communications of the ACM*, vol. 53, no. 10, p. 72, 2010.

[14] H. Schulze and K. Mochalski, "Internet Study 2008/2009," *Africa*, pp. 1–13, 2009.

[15] J. Liang, R. Kumar, and K. Ross, "Understanding kazaa," *Manuscript, Polytechnic University Brooklyn*, 2004.

[16] Bittorrent, *The BitTorrent Protocol Specification*, 2008.

[17] S. El-ansary and S. Haridi, "An Overview of Structured P2P Overlay Networks," 2004.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A Scalable Peer-to-peer Pookup Service for Internet Applications,” *Sigcomm*, pp. 1–14, 2001.

[19] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 329–350, Springer, 2001.

[20] B. Y. Zhao, J. Kubiatowicz, A. D. Joseph, *et al.*, “Tapestry: An infrastructure for fault-tolerant wide-area location and routing,” 2001.

[21] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, “OceanStore: An Architecture for Global-Scale Persistent Storage,” *Asplos*, 2000.

[22] D. Eastlake and P. Jones, “Us secure hash algorithm 1 (sha1),” RFC 3174, RFC Editor, September 2001. `http://www.rfc-editor.org/rfc/rfc3174.txt`.

[23] D. Mazieres and P. Maymounkov, “Kademlia: A Peer-to-peer Information System Based on XOR Metric,” 2002.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*, vol. 31. ACM, 2001.

[25] A.-m. K. Pathan and R. Buyya, “A Taxonomy and Survey of Content Delivery Networks,” *Grid Computing and Distributed Systems GRIDS Laboratory University of Melbourne Parkville Australia*, vol. 148, pp. 1–44, 2006.

[26] E. Nygren, R. K. Sitaraman, and J. Sun, “The Akamai Network: A Platform for High-Performance Internet Applications.”

[27] E. Berlekamp, “Bit-serial reed-solomon encoders,” *IEEE Transactions on Information Theory*, vol. 28, no. 6, pp. 869–874, 1982.

[28] M. J. Freedman, F. Eric, and D. Mazieres, “Democratizing content publication with coral,” *NSDI’04 Proceedings of the 1st conference on Symposium on Networked Systems*, vol. 17, no. 3, p. 365, 2004.

[29] ECMAScript, *Standard ECMA-262 Language Specification*, 2017.

[30] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, “The QUIC Transport Protocol: Design and Internet-Scale Deployment,” *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 183–196, 2017.

[31] P. Gross and P. Almquist, “Iesg deliberations on routing and addressing,” RFC 1380, RFC Editor, November 1992.

[32] K. B. Egevang and P. Francis, “The ip network address translator (nat),” RFC 1631, RFC Editor, May 1994. `http://www.rfc-editor.org/rfc/rfc1631.txt`.

[33] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, “Session traversal utilities for nat (stun),” RFC 5389, RFC Editor, October 2008. `http://www.rfc-editor.org/rfc/rfc5389.txt`.

[34] R. Mahy, P. Matthews, and J. Rosenberg, “Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun),” RFC 5766, RFC Editor, April 2010. `http://www.rfc-editor.org/rfc/rfc5766.txt`.

[35] J. Wu, Z. Lu, B. Liu, and S. Zhang, “PeerCDN: A novel P2P network assisted streaming content delivery network scheme,” *Proceedings - 2008 IEEE 8th International Conference on Computer and Information Technology, CIT 2008*, pp. 601–606, 2008.

[36] I. Baumgart and S. Mies, “S/Kademlia: A practicable approach towards secure key-based routing,” *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 2, 2007.

[37] W. Ali, S. M. Shamsuddin, and A. S. Ismail, “A survey of web caching and prefetching,” *International Journal of Advances in Soft Computing and its Applications*, vol. 3, no. 1, pp. 18–44, 2011.

[38] M. Jelasity, A. Montresor, and O. Babaoglu, “T-Man: Gossip-based fast overlay topology construction,” *Computer Networks*, vol. 53, no. 13, pp. 2321–2339, 2009.

[39] B. J. Hightower K, Burns B, *Kubernetes: Up and Running*. O’REILLY, 2017.

[40] D. Merkel, “Docker: lightweight Linux containers for consistent development and deployment,” *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[41] K. T. Rosen and M. Resnick, “The size distribution of cities: An examination of the Pareto law and primacy,” *Journal of Urban Economics*, vol. 8, no. 2, pp. 165–186, 1980.

[42] L. Adamic, “Zipf, Power-laws, and Pareto - a ranking tutorial,” 2000.