Startrail - Adaptative Network Caching for Peer-to-peer File Systems

João Tiago
INESC-ID Lisboa
ULisboa / Instituto Superior Técnico
thesis@joaotiago.dev

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 Distribution 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.

ACM Reference Format:
João Tiago. 2019. Startrail - Adaptative Network Caching for Peer-to-peer File Systems. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 system, computers connected to the network played an equal role, each capable of contributing with as much as they utilised. 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. In this pattern, computers play either the role of a consumer - "client"- or producer - "server" - serving content to the network. Serving a big client base requires enormous amounts of server resources. As demand grows, performance deteriorates and the system becomes fragile. 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. Such innovations made the early client-server architecture a little more robust, but at considerable cost. Still, despite its inefficiency, the client-server model remaines dominant today and runs most of the web.

IPFS [5] seeks to revert the historic trend of a client-server only Web. It is a decentralized peer-to-peer content-addressed distributed file system that aims to connect all computers offering the same file system. 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[18][1], 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.

The goal of this work is to, taking advantage of the ecosystem build by IPFS, developing 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;

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- 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.

Startrail serves as a key enabling component for a future IPFS-based CDN.

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.

The rest of the document is as follows. In Section 2 we review the state of the art algorithms and architectures used by the relevant systems. Section 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 analyze Startrail's architecture, its algorithms and data structures. Section 4 describes the testbed platform used, all the relevant metrics and obtained results. Some concluding remarks and extension proposals are presented in Section 5.

2 Related Work

2.1 Distributed File System

Is a file system that supports the sharing of files over a set of network connected nodes. The challenge is keeping the system performant, secure and robust. In addition, 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.

Distributed follow different types of architectures [19]:

- **Client-Server Architecture** - the simplest; a directory tree to multiple clients (e.g. NFSv3 [8]). The capacity of the system is limited by the capacity of the server. The server is a single point of failure.
- **Object-based File Systems** - 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. This architecture allows for incremental scaling. E.g. GFS [7], MDFS [13].
- **Peer-to-Peer File Systems** - distributed systems consisting of interconnected nodes (peers). These can be:
  - **Unstructured**: the network imposes no constraints on the links between different nodes. The placement of content is completely unrelated to the overlay topology. Require little maintenance while peers enter and leave the system. These systems suffer from the lack a way to index data. Thus, resource lookup mechanisms consist of ineficient brute-force methods. Limited scalability.
  - **Structured**: fix the scalability issues of unstructured. The overlay topology is tightly controlled and data is placed at specific locations in the overlay [6]. These systems provide a mapping between the data identifier and location, so that queries can be efficiently routed to the node with the desired data. Chord [17], Pastry [16] and Tapestry [20]

2.2 Content Distribution

Content Delivery Networks (CDNs) are networks of geo-distributed machines that deliver web content to users based on their geographic location.[14] 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.

2.3 Web Distributed Technologies

A few technologies are kew for the success of design of decentralized systems in the Web. There are:

- The Browser: a very powerful tool to fuel adoption. Building a solution that runs on the browser means one is are able to reach a broad audience;
- Node.js: allows running JavaScript in the server. Bundles with the Node Package Manager (NPM) a very power library of reusable JavaScript modules;
- WebRTC: provides browsers and mobile applications with peer-to-peer Real-Time Communications capabilities accessible through a Javascript API. Using a suite of protocols, solves the problem of NAT traversal.

3 Architecture

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.

We shall now explore, in more detail, the intended behavior of the network. Figure 1 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*.

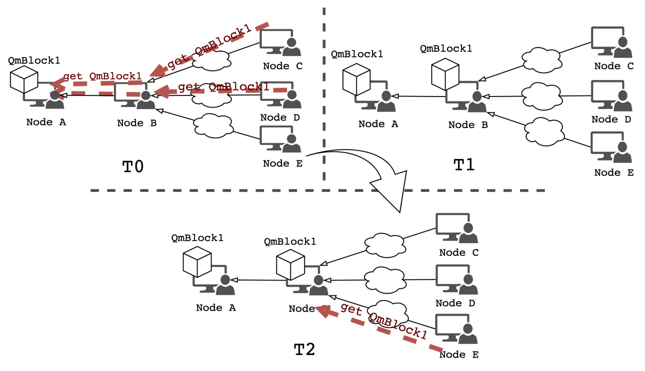

Figure 1. Illustration of the proposed Startrail flow

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. 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 [12]. 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. A high level overview of the architecture of the IPFS core is depicted in Figure 2. We shall delve into each of the illustrated components.

- **Core API** - The Application Programing 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.

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

- **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 [4]. It has tools for peer discovery and content or peer routing.

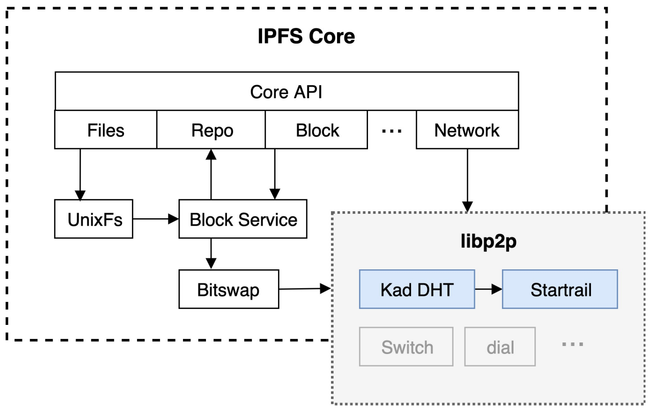

Figure 2. IPFS Core's Architecture

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

- `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.

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 query the local providers database for the know providers of a certain cid and if not enough are gathered, query the rest from the DHT 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.

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:

- On `Kad-DHT`, checking for the CID associated to `GET_PROVIDER` messages;

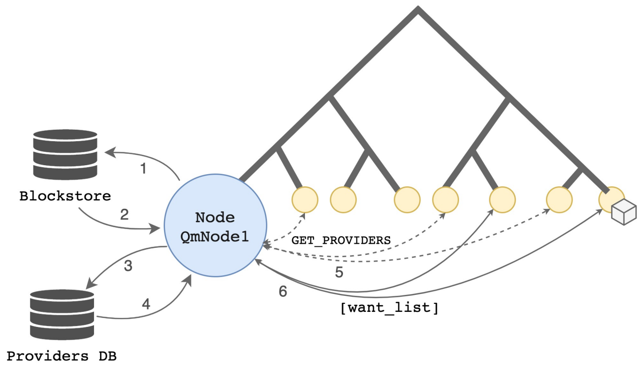

Figure 3. *Bitswap* and kad-dht interaction when fetching a block from the network

- On *Bitswap*, listening for new want_list messages and tracking each of the included CIDs;

Our solution implements the former.

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; Is responsible for integrating with the data trading module, Bitswap; With the BlockService to access data storage and libp2p for varios network utilities.
- The *Popularity Manager*. The component responsible for tracking 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 4

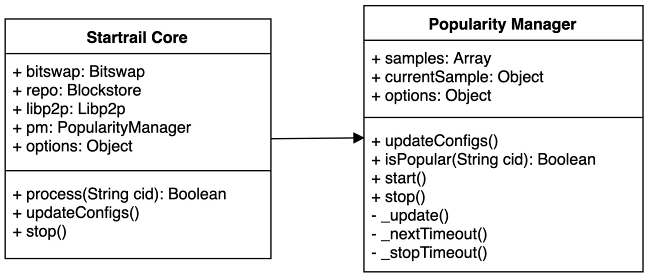

Figure 4. Class diagram of Startrail's Core and Popularity Manager

Further analysis of the Core's diagram reveals the afordmention integrations, including with the internal Popularity Manager. It also reveal 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; 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 hotrealoading the configuration when testing.

Additionally, for the Popularity Manager class we find: isPopular(cid), for updating and calculating the popularity for any CID passed as argument. updateConfigs(), serves the same purpose as the above mentioned one; 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.

Message processing algorithm To recognize patterns in object accesses, Startrail examines the CIDs sent on discovery messages. Hence, the `process()` function is triggered every time the `GET_PROVIDER` message handler is called. Figure 5 unveils the execution flow starting when a peer requests a block from the network triggering the search for providers on the network. Upon receiving a `GET_PROVIDER` message, the `kad-dht` handler will execute Startrail's `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 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* 1.

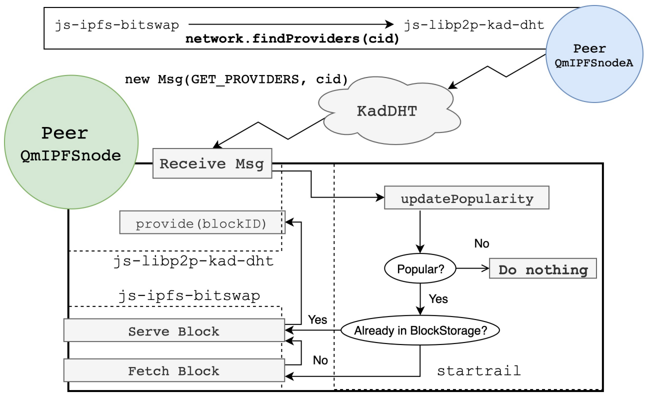

Figure 5. Execution flow inside the Startrail module

Listing 1. Startrail processing engine

```
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);
  }
```

Popularity Calculation Algorithm 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. 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 ??) that the popularity calculations are made. An illustration of the interaction between samples and block arrivals is represented on Figure 6.

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. 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.

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

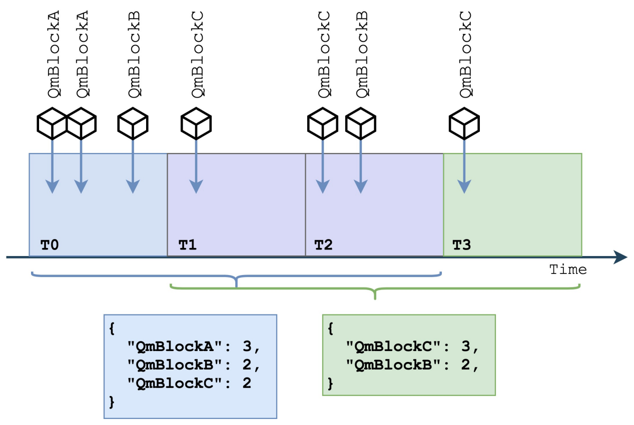
QmBlockA
QmBlockA
QmBlockB
QmBlockC
QmBlockC
QmBlockB
QmBlockC
T0
T1
T2
T3
Time
{ "QmBlockA": 3, "QmBlockB": 2, "QmBlockC": 2 }
{ "QmBlockC": 3, "QmBlockB": 2, }

Figure 6. Interaction between new block arrivals and sampling windows

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 [3], 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.

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. 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's 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 garbage collector. When the threshold of 90% of IPFS' storage is reached blocks are no longer pinned in order to leave room for new blocks.

4 Evaluation

4.1 The Testbed

Although not planned from the beginning, there was a considerable amount of effortput 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 [10], as this would require porting the whole protocol to the Java API.

Testbed Architecture

To allow for easy integration of new computing power into the testbed and management of deployments, we resorted to Kubernetes[9]. Kubernetes is a tool for deploying and managing containerized applications. It handles the discoverability and liveness of the deployed workloads.

The smallest computing unit on Kubernetes is the *Pod*. The *Pod* is a formation ofcontainers, in our case Docker [11] containers. 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* [2]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 7.

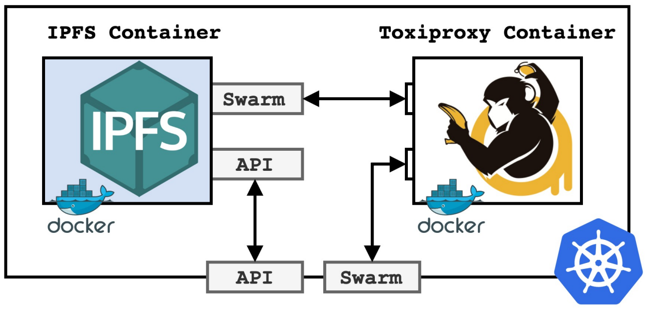

Figure 7. Composition of the Startrail Testbed Kubernetes Pod

We wanted to make the simulations easily reproducible, so we leveraged Helm [3], a tool that helps us release and manage Kubernetes applications. This enabled us to create different node configurations, *Charts*, and change them for every deployment.

To be able to gather data from different layers of the system during simulations implemented the ELK Stack [4]suite of tools. That provides tools for log indexing, searching, transformation, storage and visualisation.

The above described architecture is depicted on Figure 8.

Deploying a network We resorted to Unix's Makefiles to automate the network setup much as possible since we were aware that the network would have to be deployed many times. The deployment process, illustrated in Figure 9, 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.

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

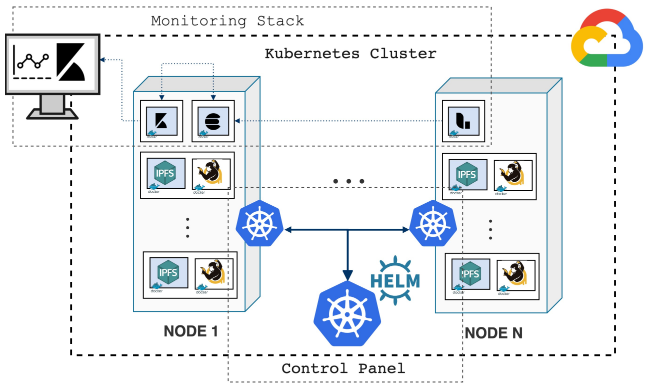

Figure 8. Architecture overview of the testbed network

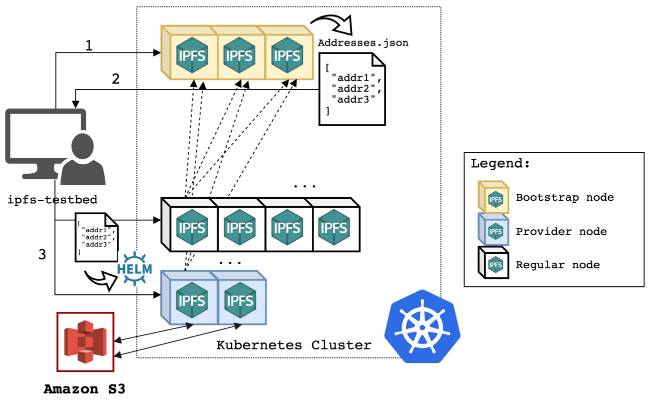

Figure 9. Deployment process on a new network on the Startrail testbed

Interacting with the network 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.

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 10

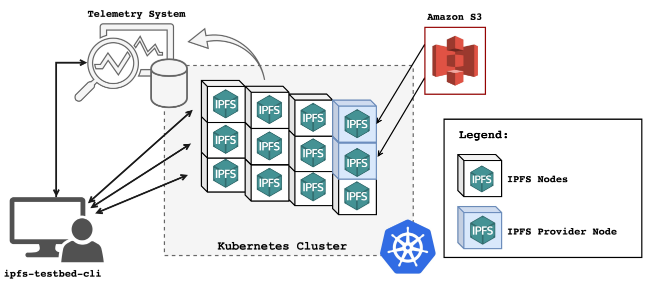

Figure 10. Testbed-cli orchestrating and monitoring Statrail testing cluster

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) that have the particularity of excluding the extreme values from the average calculation, the possible *outliers*.

4.3 Testing Setup

The implementation network used to execute the simulations was deployed in the aforementioned Kubernetes cluster running on three *n1-standard-4* Virtual Machines on Google Cloud. 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 tosimulate a network of 100 peers, 5 bootstrap and 2 providers preloaded with data.

The amount of nodes simulated were limited by the resources available in the cluster as we were running on a limited budget.

To simulate different content access patterns we ran the following request functions which pick a block from a dataset of thousands and order the peer to fetch it:

- **Random Access** - The random access picks each block with equal probability. This pattern would serves as control;
- **Pareto Random** - The Pareto Distribution [15] serves as an adequate approximation for Internet objects popularity[2]. To achieve close to the desired distribution we used an alpha 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. We will also use a Pareto distribution to pick here.

Table 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.

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

Table 1. Different testing condition for running network tests

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.

4.4 Tests Results

Latency Analysis The graph on Figure 11 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.

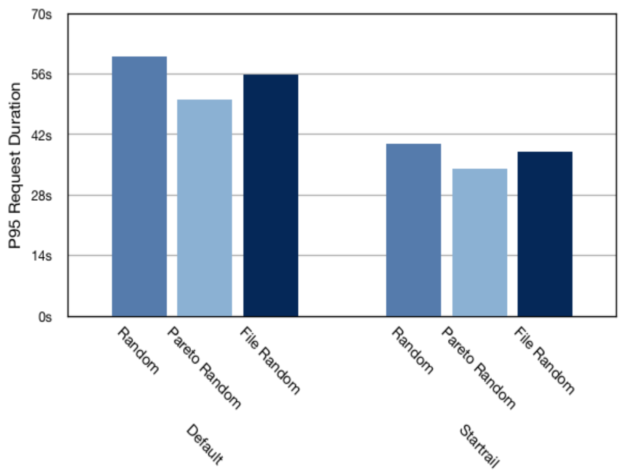

Figure 11. 95th percentile of request duration for the different testing scenarios

Memory Consumption Analysis The Graph on Figure 12 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. 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. 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 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.

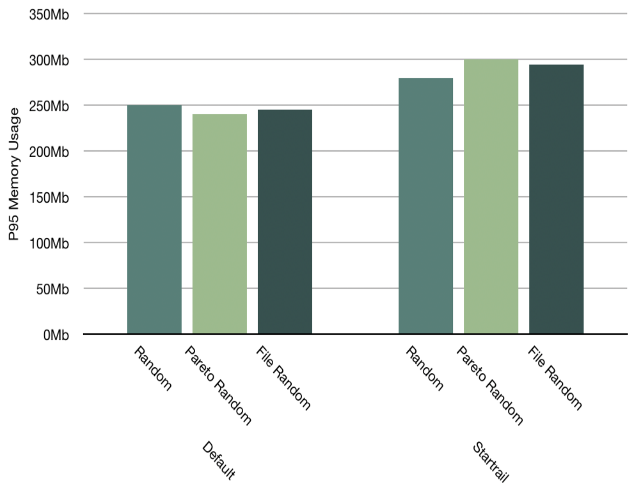

Figure 12. 95th Percentile of memory usage on the different testing scenarios

Network Consumption Analysis The Graph on Figure 13 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.

4.5 Variable Startrail percentages

Additionally, we also assessed 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 aditional simulations. The conditions of these were 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.

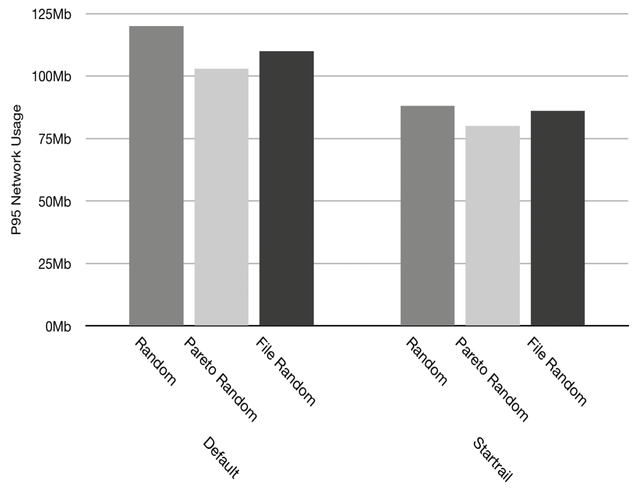

Figure 13. 95th Percentile of network usage on the different testing scenarios

The percentages were: 0%, 30%, 50%, 80% and finally 100% Startrail nodes.

The results obtained from running the simulations were compiled into the graph in Figure 14. 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 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.

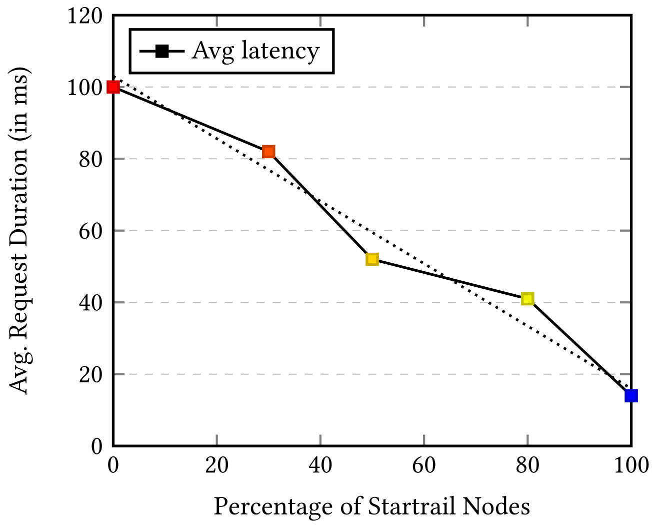

Figure 14. Avg request duration vs. Startrail nodes percentage

5 Conclusions

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 it 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 design enabling a distributed, peer-to-peer CDN, we described 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 describing 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. 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_lists*;

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 hopes that it will be eventually necessary. 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.

References

[1] Merkle-dags. Technical report, Protocol Labs.

[2] Lada Adamic. Zipf, Power-laws, and Pareto - a ranking tutorial. 2000.

[3] Waleed Ali, Siti Mariyam Shamsuddin, and Abdul Samad Ismail. A survey of web caching and prefetching. *International Journal of Advances in Soft Computing and its Applications*, 3(1):18–44, 2011.

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

[5] Juan Benet. IPFS -Content Addressed, Versioned, P2P File System. *arXiv preprint arXiv:1407.3561*, 2014.

[6] Sameh El-ansary and Seif Haridi. An Overview of Structured P2P Overlay Networks. 2004.

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, page 29, 2003.

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

[9] Beda J Hightower K, Burns B. *Kubernetes: Up and Running*. O'REILLY, 2017.

[10] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009.

[11] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[12] Ralph 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)*, 293 LNCS:369–378, 1988.

[13] Microsoft. Distributed File System (DFS): Referral Protocol. 1996.

[14] Al-mukaddim Khan Pathan and Rajkumar Buyya. A Taxonomy and Survey of Content Delivery Networks. *Grid Computing and Distributed Systems GRIDS Laboratory University of Melbourne Parkville Australia*, 148:1–44, 2006.

[15] Kenneth T. Rosen and Mitchel Resnick. The size distribution of cities: An examination of the Pareto law and primacy. *Journal of Urban Economics*, 8(2):165–186, 1980.

[16] Antony Rowstron and Peter 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*, pages 329–350. Springer, 2001.

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

[18] Pedro Teixeira, Hector Sanjuan, and Poyhtari Samuli. Merkle-CRDTs (DRAFT). pages 1–26, 2019.

[19] Tran Doan Thanh, Subaji Mohan, Eunmi Choi, Kim SangBum, and Pilsung Kim. A taxonomy and survey on distributed file systems. *Proceedings - 4th International Conference on Networked Computing and Advanced Information Management, NCM 2008*, 1:144–149, 2008.

[20] Ben Yanbin Zhao, John Kubiatowicz, Anthony D Joseph, et al. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. 2001.