Bachelor thesis
Computing Science

Radboud University Nijmegen

Searching IPFS

Author:
Jasper Haasdijk
jhaasdijk@protonmail.com

First supervisor/assessor:
prof. dr. ir. A.P. de Vries
a.devries@cs.ru.nl

Second assessor:
dr. ir. E. Poll
e.poll@cs.ru.nl

January 20, 2019

Abstract

This thesis describes the implementation of a prototype search engine based on IPFS. This prototype relies on the existing PubSub implementation of IPFS to create a shared channel on which nodes can publish queries and corresponding queryhits.

To increase runtime performance and decrease communication overhead, we introduce a summary file. This file contains a summary of the data that is available in the cluster.

Using this prototype we are able to locate assets in an IPFS cluster.

Contents

Chapter 1

Introduction

Juan Benet describes the InterPlanetary File System (IPFS) as "a peer-to-peer distributed file system that seeks to connect all computing devices with the same set of files" [1]. It is a protocol designed to create a network of peers which are all connected to the same (shared) filesystem. It is a digital network which allows sharing of network resources. However, IPFS lacks one key feature: search functionality. In this thesis we will address this absence and propose a prototype search engine based on IPFS. To achieve this, we define the following research question:

> How to devise a search engine in a decentralized file system to locate assets in an IPFS cluster?

1.1 Motivation

The Web today is moving more and more towards a centralized structure. Large server farms are under the control of a select few companies, which makes it possible to control the information flow, at least in principle. This enables digital monopolies, and governments can even use this to block specific content which they deem not suitable for their residents. One example of this can be seen in Turkey in 2017, when the Turkish government decided that Wikipedia is a "threat to national security" and subsequently restricted all access to the site [2].

Incidents like this fuel the existing concerns regarding the Web's evolution towards this centralized structure [3, 4, 5]. These incidents motivate a more decentralized setup [3]. Examples of such projects are Diaspora[1], Mastodon[2], ZeroNet[3] and IPFS.

[1] `https://diasporafoundation.org/`
[2] `https://joinmastodon.org/`
[3] `https://zeronet.io/`

IPFS allows for a Peer-to-Peer, decentralized network on which content can be served by anyone who possesses it. This makes the content incredibly hard to attack and allows users to view and serve this content, even in more restrictive areas. However, within IPFS there are no searching or indexing tools. This makes it difficult to determine which information can be found, and where. One solution is to share the hash of the content on publicly available forums. Since data is content-addressable, sharing the hash enables others to directly access the data. This is a rather cumbersome method and a search engine would make it much easier to search what content is accessible on the network.

The motivation to tackle this problem stems from the observation that searching is a vital part of any network. The ability to share network and computing resources is a key element of any network. If you are unable to share such resources, there is no point in being connected. IPFS aims to replace HTTP but without the means to search through its contents it will never live up to this goal. Therefore the addition of searching functionality to IPFS is an essential aspect in the adoption and future of the protocol.

The following chapters of this paper will walk you through our implementation of a prototype search engine based on IPFS. We will walk you through the design and creation using ample examples.

Chapter 2

Preliminaries

To set the scene we begin with a brief overview of the most important concepts discussed in this thesis.

2.1 IPFS

"IPFS is a peer-to-peer distributed file system that seeks to connect all computing devices with the same system of files" [1]. It is a protocol designed to create a network of peers which are all connected to the same (shared) filesystem.

While there is much to discover about IPFS, this Section focuses on the specific techniques and properties used in this thesis.

2.1.1 Content

IPFS employs content addressed storage. What this means is that when accessing data through IPFS, instead of using location based addressing such as an object's link, we use the contents of the data, specifically the hash. Adding a file `x` to the filesystem is done using the `ipfs add x` command. This will also print the resulting IPFS hash of the file that was just added. Getting a file from the filesystem is done using the `ipfs get <hash>` command, where `<hash>` is the IPFS hash of the file you want to fetch.

Data is stored using IPFS objects. Listing 2.1 shows the IPFS Object format [1]:

Listing 2.1: The IPFS Object format

```
type IPFSObject struct {
    links [] IPFSLink
    // array of links

    data [] byte
    // opaque content data
}
```

As we can see, an IPFS object contains a `links` and a `data` field. A single IPFS object can store up to 256 KB of data. When storing simple text files this is enough. When storing files that are $>$ 256 KB, the file is split up in parts of $\leq$ 256 KB. An empty parent object will then link all the pieces of the file together. This is shown in Figure 2.1.

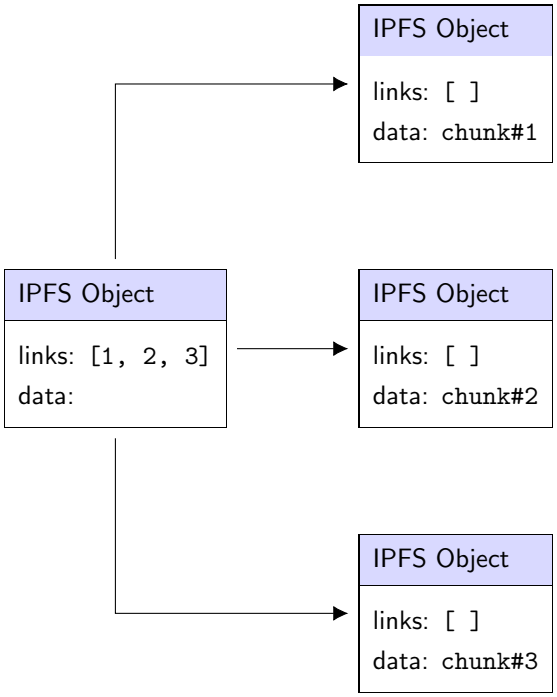

Figure 2.1: An empty parent object links the file together

2.1.2 Versioning

An updated file is different from the original file, which will result in a different IPFS hash. To support versioning, IPFS uses `commit` objects. A commit object contains a `parent` and an `object` field. When updating

an object, IPFS links the `parent` field to the IPFS object that has been updated. This allows us to track updates over files without losing older versions. Figure 2.2 shows how updating a file in IPFS using commit objects looks like. We have a text file with `"Hello World!"` and update this to `"Hello IPFS!"`.

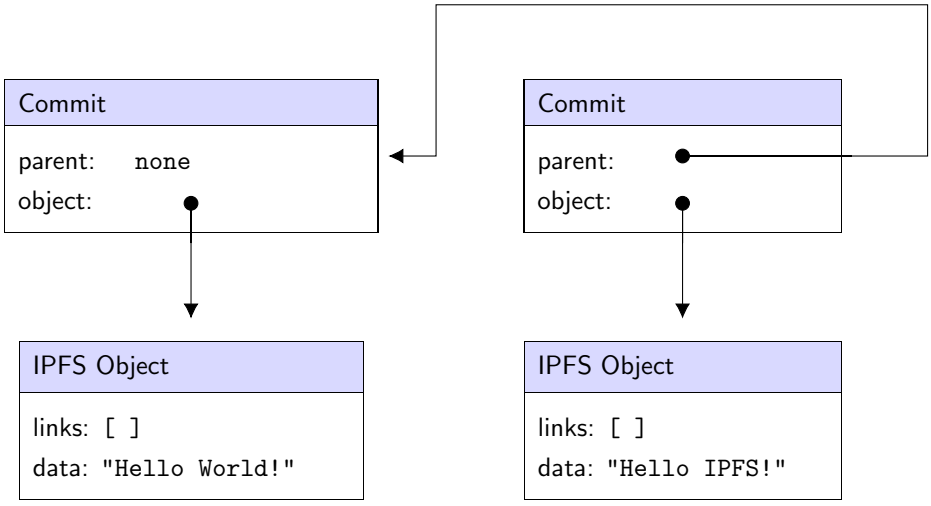

Figure 2.2: Updating an IPFS Object

2.1.3 PubSub

PubSub is short for the messaging pattern called Publish-Subscribe. IPFS employs a topic-based system. Publishers can post messages to a particular topic. Subscribers can subscribe to such a topic and using this approach only receive messages they are interested in. Subscribers do not necessarily know who is sending the messages. They simply express their interest in a particular topic. This offers great network scalability and flexibility.

The current implementation of PubSub in IPFS is called floodsub. This is due to the fact that messages are sent to every peer in the network. This is a very simple, easy to implement protocol. The problem however is that this creates a rather large communication overhead as there is no limit on the outbound degree when forwarding messages. This can sometimes lead to the same message being received multiple times. Therefore the IPFS team is working hard to improve this protocol [6]. This has lead to the specification of the gossipsub protocol, and a proposal for the episub protocol.

2.2 Search Engine

A search engine is a system which allows a user to search for information on a network. Before such a system can serve search queries, it needs to crawl and index the available data. When transitioning into a P2P, decentralized setup, these processes need to be adjusted to accommodate for the change in setup. For example, in a truly decentralized solution, we cannot employ a single, centralized index.

2.2.1 Indexing

Indexing is the term (related to search engines) used to describe the process in which raw data is parsed into a searchable format, the index. This provides a more efficient way of searching through the complete data collection.

2.2.2 Crawling

The most common method used by search engines to gather the data for the search index, is by employing a crawler. A Web crawler is a program that systematically gathers and processes the content of a webpage. This content is parsed and added into the search index. In a centralized environment this is a viable strategy. The company behind the search engine has a vested interest in gathering this data so it can serve its users relevant queryhits. There is no need to create an incentivised structure as the company is willing to do the work.

In a decentralized solution we cannot employ this method. There is no single entity we can rely on to provide us this function. Employing a Web crawler inevitably reduces the freshness of the search results [7]. Instead, we have to move to a completely new approach. An approach where content creators do the work for the data they want, so that there is no need to create an incentivised structure since anyone who *wants* something has to do it themselves. This is called the *no-crawling* approach.

Networks which do not employ the *no-crawling* approach are bound to create an incentivised structure. This is why the company behind IPFS, Protocol Labs, is working hard to create Filecoin [8]. Filecoin is a token that can be used to hire miners to store or distribute data for you. This is an alternative approach by paying nodes to do work they do not need themselves.

Chapter 3

Research

To define the scope of this thesis, we take a more concrete look at our problem definition. Our earlier stated research problem consists of the following question: "How to devise a search engine in a decentralized file system to locate assets in an IPFS cluster?". This is a rather broad question and can be interpreted in a variety of ways. Therefore we instead introduce a new, much more concrete problem definition.

> If we were to have a document dump consisting of Wikipedia pages, how do we arrange this collection of pages in a distributed IPFS cluster with 1, 2, ..., n clients so that we can support searching on the contents?

In Section 3.1 we present the main idea of this thesis. This Section contains the general approach behind our prototype search engine. Sections 3.2 and 3.3 explain the specific implementation of our prototype. This consists of two parts: a Listener and a Publisher. In Section 3.4 we will collect and process the document dump of Wikipedia pages, set up our cluster environment and show that using our search engine we are able to support searching on the contents of the cluster.

3.1 Design

The main idea presented in this thesis is a prototype search engine built on top of IPFS. This prototype supports searching by listening to the queries and answer messages that are published on the network. Whenever a node in our network wants to search for something, it publishes a query. Other nodes can then respond to this query by publishing an answer message.

When talking about nodes we refer to the individual peers in the network. Every node has their own content. A collection of documents that they are

willing to share with the rest of the network. Nodes can come and go as they please. Whenever a node decides to leave the network, so will their content. This choice stems from the observation that this is simply how the IPFS network works. If nobody wants to share specific content then that content will disappear.

The current implementation is not the most efficient one. It relies on a flooding approach as we simply publish every query we have to everyone on the network. This provides us with a high recall rate, at the expense of a lot of communication overhead. Many alternatives to this approach have been proposed, including overlay networks which try to organize "peers sharing common interest into clusters" with the goal to try and exploit this information when submitting queries [9].

In this thesis we aim to increase the speed and efficiency of our implementation by introducing a summary file. A summary file is a local list of {`document, score`} value pairs matched to a given query. This allows us to know beforehand which data is available in the network. For example our summary list could contain the following entry:

Listing 3.1: Summary file entry for `"World Wide Web"`

```
"World Wide Web": [
   {"ref":"QmbQTdn3swA7fHZjcEmZiC13shMi4mhND4YxT4iMGVXpSA",
    "score":1.812051130639889},
   {"ref":"QmXEg8dpxsDJ3dW8ZHa2M4Xri75EpQF9m6ypww2fe53bW5",
    "score":0.7558747218457033},
   {"ref":"QmWDjcyBFNVvUGcyBmr7FbarQ7dPEa5x8p1XLcvqgAR92H",
    "score":0.7469340688850746},
   {"ref":"QmSpC17ew83jjXrpLLh3eNweKGkNnUUJwqsDyHBHGkxTcb",
    "score":0.6883991979016595},
   {"ref":"QmZzMs5MdSU3nFfUWHa8sVJtkuXqAPyfacaY3uqjR2Fee2",
    "score":0.4483595602046584}
]
```

Now if we search for "World Wide Web", we can look into our local summary file and resolve this search without publishing the query.

Data entries that are in the summary file can originate from other nodes on the network that have responded to a query. You can compare this with a cache of previously seen answer messages. Using this file we create a summary of the data that is available to us in the network.

In the remaining part of this Chapter we will elaborate on this idea, and motivate how our solution solves the problem by providing a technical overview.

3.2 Implementation

This Section gives a technical overview of the implementation of our prototype. Our prototype uses the IPFS Javascript library[1] and consists of two parts: a Listener and a Publisher.

To be able to send and receive messages on the network we create a shared PubSub channel. This is done by letting every node in the network subscribe to the same topic. Whenever a node decides to publish something, PubSub will distribute this message to every other peer currently connected.

3.2.1 Listener

Our Listener is responsible for returning queryhits. The program flow can be divided in three parts; starting the Listener, handling network events, and shutting down the Listener.

Starting the Listener

When starting the Listener process, we check whether there exists a serialized local index file we can use. If there exists such a file, we do not have to rebuild the entire search index but can simply load it from disk. If this file does not exist, we generate a new search index and add our local files to it. Now that we have a local search index, we subscribe to the shared PubSub channel and start listening for events.

Handling network events

When subscribing to the shared channel, we have to supply an event handler. This event handler is used to parse received events. Two different types of events exist, `query` events and `answer` events. Figure 3.1 shows the design for these messages.

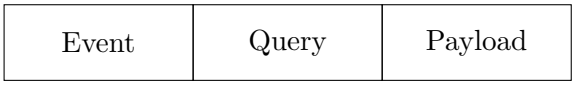

Figure 3.1: Pubsub message design

The `Event` field differentiates between a `query` and an `answer` event. The `Query` field is used to track which query we are dealing with. In the case of a `query` event this field indicates what we are querying for, in the case of an `answer` event this field indicates for which query the message is returning queryhits. The `Payload` field is used to transfer queryhits. In the case of a `query` event this field simply returns the empty string. In the case of an

[1] `https://github.com/ipfs/js-ipfs-http-client`

`answer` event this field returns a list of {`document, score`} value pairs. A document's score is calculated based on a similarity measurement with the query.

Figure 3.2 shows an example query message:

query	"Tomato"	""

Figure 3.2: Query event for `"Tomato"`

This message indicates that we are querying for the string `"Tomato"`. An example `answer` message for this query is shown in Figure 3.3:

answer	"Tomato"	<payload>

Figure 3.3: Answer event for `"Tomato"`

Where `<payload>` refers to a list of value pairs, for example:

Listing 3.2: Answer payload for a query for `"Tomato"`

```
1  [ { ref: 'QmTH5q7s7skYHaUwiKm4CASGGnkDy6zBtE5qmtDKuP8xit',
2    score: 0.7637125121768537 },
3  { ref: 'QmPm9ChQV4kJmV4vtPH3z6Nj2e7zZCn5AKjS9dkNbHArxp',
4    score: 0.3677390649567145 },
5  { ref: 'QmPxXH9xNbXxWvsF6zAYP8YJozZUfP49keWrWTNA47NFT6',
6    score: 0.35476848530507377 },
7  { ref: 'Qmc3UqTna5tAkE2gisyNyqxMt2c7mibaC168ofeT4kstj8',
8    score: 0.32045582549361074 },
9  { ref: 'QmTUCWEZTG2F6QcbMKNGNq3MhqcDg4tWoEswFPABPF4Sz7',
10   score: 0.2233432437517813 } ]
```

On receiving a `query` message, our Listener searches its local index and publishes the queryhits in a matching `answer` message.

Shutting down the Listener

When shutting down the Listener process we invoke its `exitHandler`. This is a function that gets called upon closing the program. This function checks if there already exists a serialized local index file. If there is no local index file yet, before shutting down our Listener, we generate a new one from our constructed search index. If there already exists an index file, we simply exit the process.

Figure 3.4 shows the flow of the Listener process schematically:

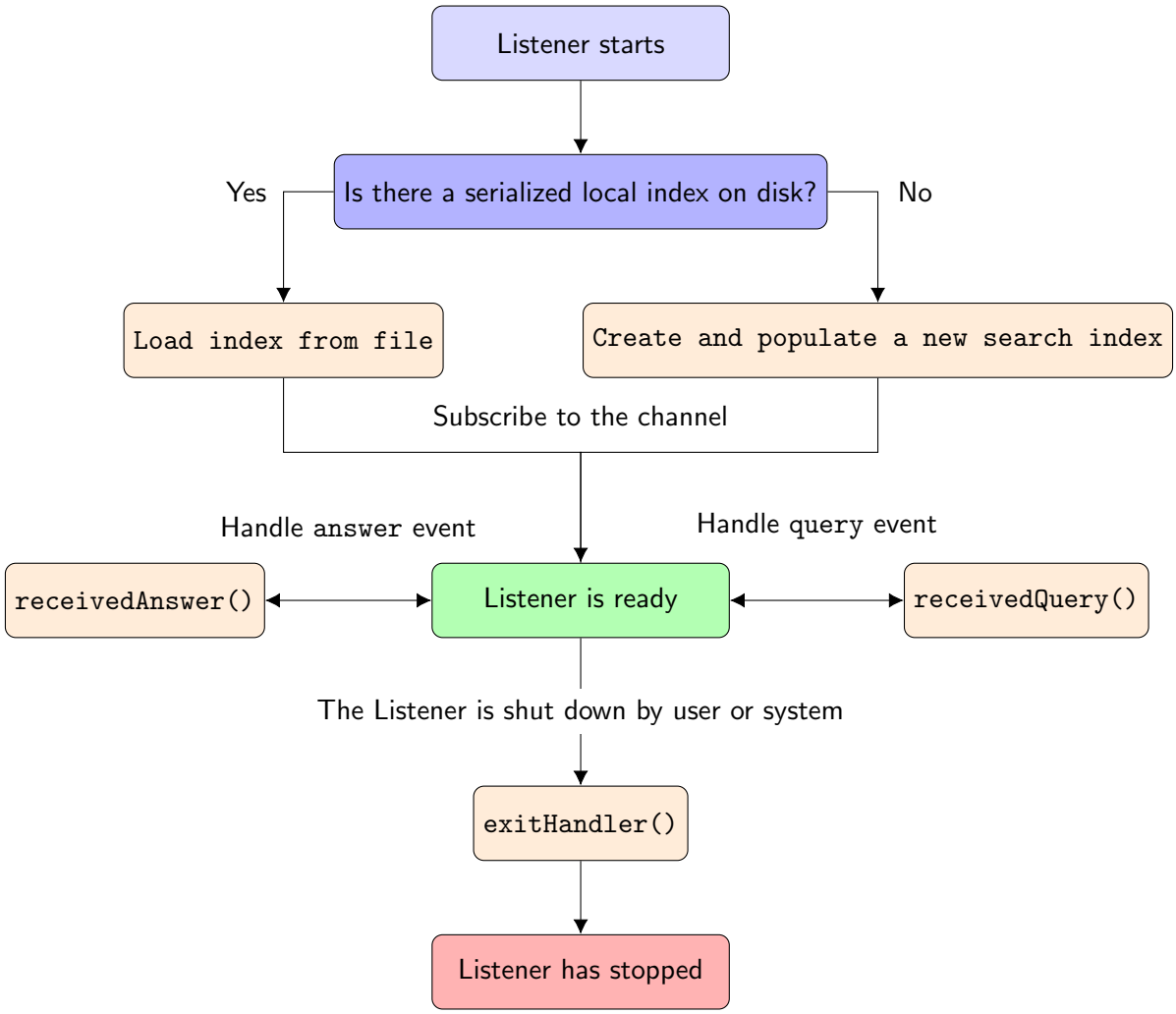

Figure 3.4: The Listener process

3.2.2 Publisher

Our Publisher is responsible for submitting queries. Although much simpler than our Listener, our Publisher's program flow can also be divided in three parts. Starting our Publisher, submitting queries, and shutting down the Publisher.

Starting our Publisher process is as simple as subscribing to the shared PubSub channel. This enables us to submit queries to the network. Figures 3.2 and 3.3 illustrate how this can be done. On receiving an `answer` event, our Publisher updates its local response list with the newly received queryhits.

We can shut down our Publisher by unsubscribing from the shared PubSub channel.

Figure 3.5 shows the flow of the Publisher process schematically:

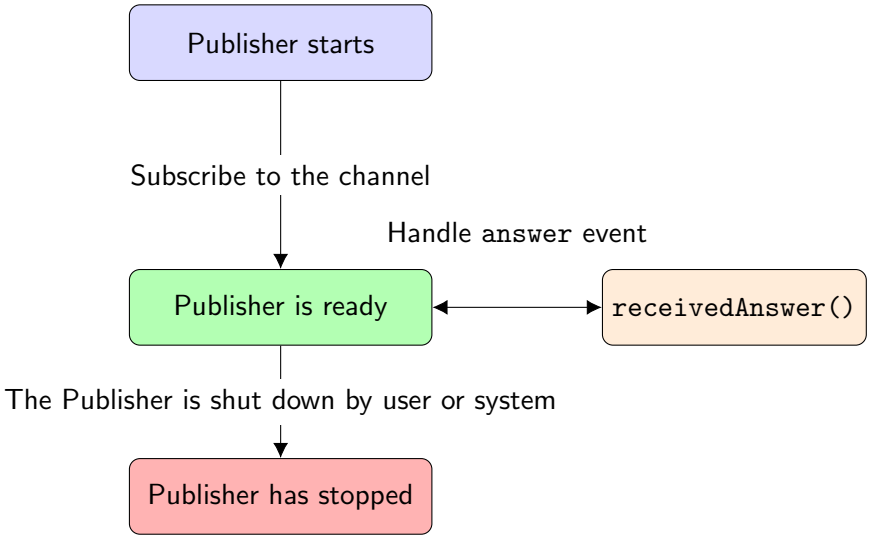

Figure 3.5: The Publisher process

3.3 Increasing efficiency

As mentioned in Section 3.1, we aim to increase the speed and efficiency of our prototype by introducing a summary file. In order for our implementation to handle this summary file, we introduce two new types of events: `sumQuery` and `sumAnswer` events. The design for these messages is the same as our original `query` and `answer` events as described in Figure 3.1.

An example `sumQuery` message looks like this:

`sumQuery`	`"Tomato"`	`""`

Figure 3.6: sumQuery event for `"Tomato"`

An example `sumAnswer` message looks like this:

`sumAnswer`	`"Tomato"`	`<payload>`

Figure 3.7: sumAnswer event for `"Tomato"`

Where `<payload>` refers to the returned list of value pairs, just as we have seen in Listing 3.2.

Notice that our summary messages are nearly identical to our `query` and `answer` messages. A `sumQuery` message indicates that we are looking to add an entry to our summary file. `sumAnswer` messages subsequently return the payload for our summary entry.

To be able to use this new feature, we need to adjust our Listener and Publisher process.

Listener needs to be able to handle `sumQuery` events. This is a rather small addition as it is similar to handling a `query` event. Only now we return a `sumAnswer` message instead of an `answer` message.

Figure 3.8 shows the updated Listener process. The changes have been highlighted.

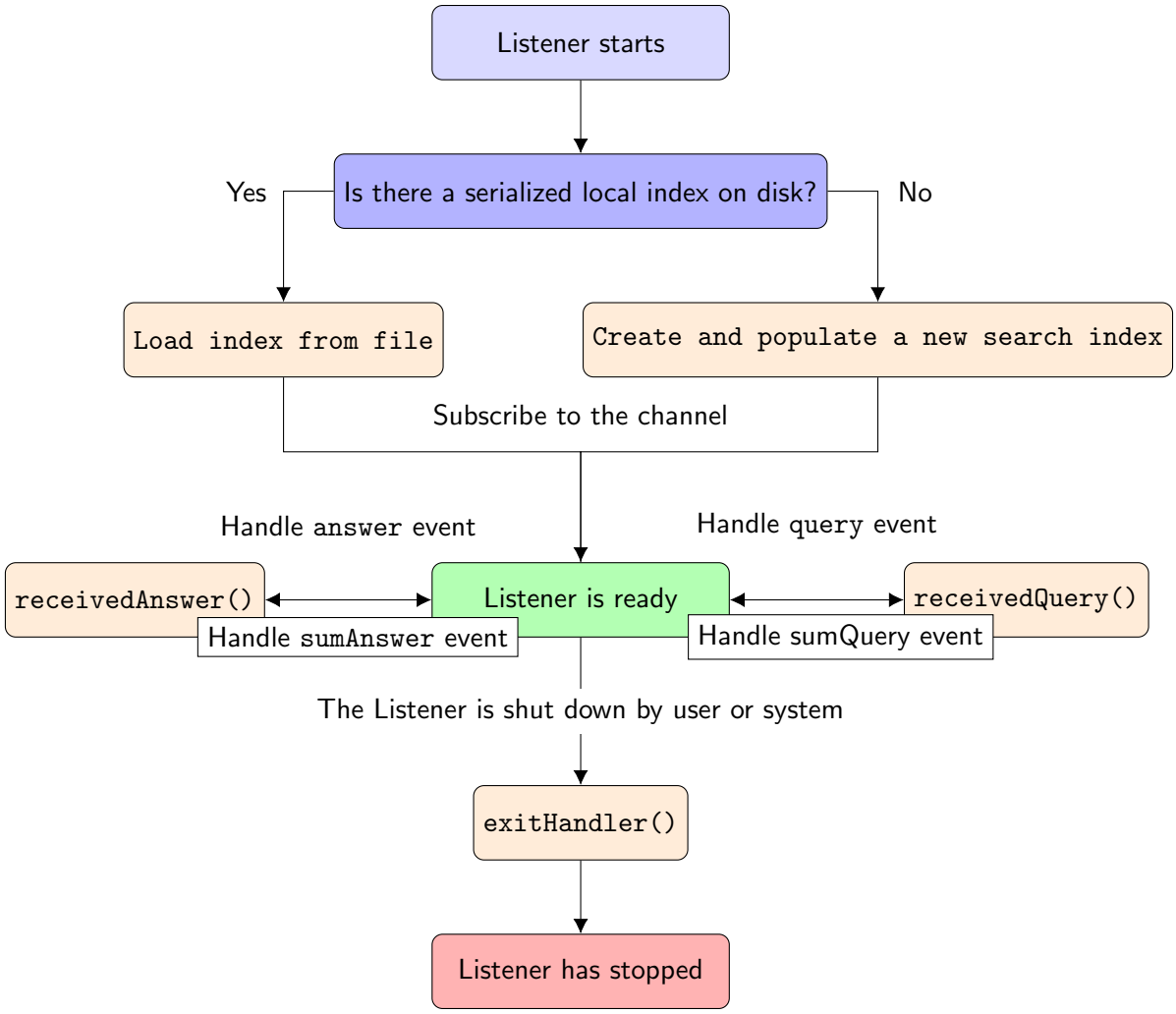

Figure 3.8: The updated Listener process

Our Publisher on the other hand undergoes quite the change. Instead of simply subscribing to the shared channel like we used to, we first need to check if there exists a local serialized summary file we can use. If there exists such a file, we don't have to generate a new one and instead load it from disk. If this file does not exist, we have to generate a new one.

We also need to update our search order. We now have 3 different sources to query when we want to search for something. Our local search index, our local summary file, and the cluster. If we are content with the scores returned from our local search index and summary file, we don't have to submit the query to the network. This saves us both in time and communication overhead.

We also add an `exitHandler` to the Publisher process, which we invoke when shutting down the process. This function checks if there already exists a serialized local summary file. If there is no local summary file yet, before shutting down our Publisher, we generate a new one from our constructed summary. If there already exists a summary file, we simply exit the process. Figure 3.9 shows the updated Publisher process.

3.4 Experiments and Evaluation

This Section provides a walk through and evaluation of how we use our prototype to support searching on the contents of a document collection consisting of Wikipedia pages.

3.4.1 Data

Searching for something only really makes sense once we have something to search for. So before we are able to do anything *with* data, we first need to *obtain* data. We will use a Wikipedia document dump, process this data into individual `.txt` files, and add them to our search index.

Data Collection

Wikipedia explains how an interested user can obtain a free copy of their available content on their "Wikipedia:Database download" page [10]. Following their instructions on where to get the data, we obtain and verify the data collection shown in Figure 3.10.

This is a collection of (a subset of) pages in their current version, that does not include metadata, category information, statistics, log events or edit histories. We are only interested in the page titles with their corresponding page text. Our desired end result is a list of Wikipedia articles where the filename is the article's title (using its `<title>` tag), and the file's content is the article's text (using its `<text>` tag).

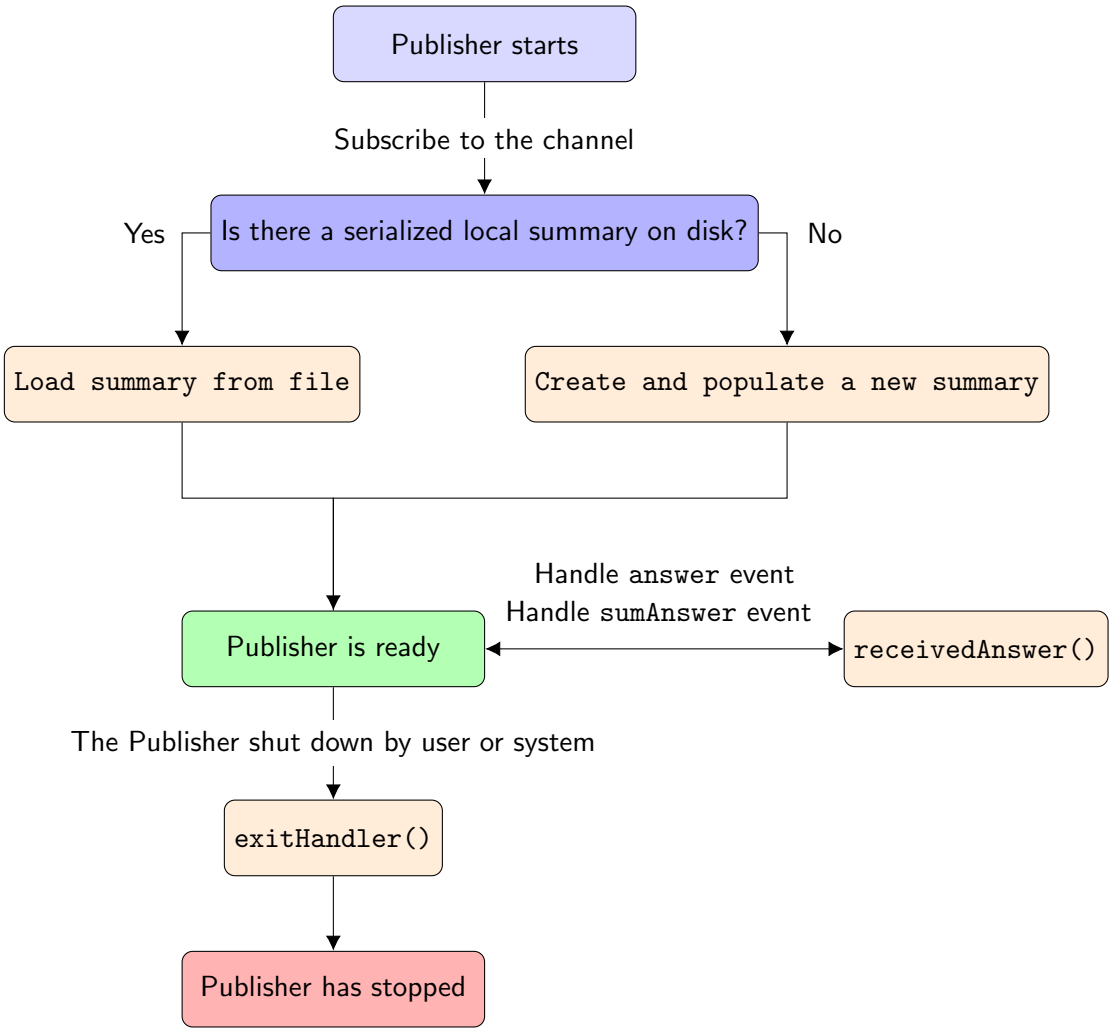

Figure 3.9: The updated Publisher process

Data Cleansing

Looking into this data, we see there is pre-processing required before we can actually do something with it. We received the article data in the form of an `.xml` dump. This file includes unwanted data entries such as redirects and disambiguation pages. We also notice that all the `<text>` fields are using the MediaWiki [11] format and that we are going to have to parse this to its plaintext equivalent. Some examples of unwanted data entries can be found in Appendix A.1.

File:	
from:	`https://dumps.wikimedia.org/enwiki/20181020/`
file:	`enwiki-20181020-pages-meta-current1.xml-p10p30303.bz2`
Hash:	
from:	`https://dumps.wikimedia.org/enwiki/20181020/enwiki-20181020-md5sums.txt`
hash:	`md5sum: cb5b3336b0bfbb3414aab61a54ecd40a`

Figure 3.10: Wikipedia data collection

Python XML processing

To ease the job of data cleaners, we can create a little `Python` program. We can break our task down into 2 subtasks.

1. Filter the unwanted data entries

We can filter any unwanted data entries using the `xml.etree.ElementTree` Python module. Note that this module comes accompanied with a warning that it is not secure against maliciously constructed data. Since we downloaded an 'official' Wikipedia data dump and verified its integrity with its hash, this should be fine for our use case. Our data parser processes the data along the following steps:

1. It loads the `xml` data using the `ElementTree.iterparse` method into a Python list of records. We construct a record for each data entry using its `<title>`, `<ns>`, `<rid>` and `<text>` tags.
2. We then filter this list based on a few simple rules.
   - We filter every data entry where the `<ns>` tag is not equal to 0. The other namespace keys are used to distinguish between categories we do not want to include at this stage, such as "Media" (key -2) or "Template" (keys 10 and 11) pages. The complete list of namespace keys can be found in Appendix A.2.
   - We filter out data entries consisting of disambiguation or redirect pages. We consider these to be metadata, and are in this project only interested in the default full-text retrieval use-case.

2. Parse the MediaWiki syntax

Now that we have eliminated the unwanted data entries, it is time to construct our article's text. This is something we can accomplish using the `mwparserfromhell` Python module, a powerful and easy to use MediaWiki parser. Using this module, we produce a clean version of the data collection which we can start processing.

Listing 3.3: Easily convert wikicode into plaintext

```
plaintext = mwparserfromhell.parse(source)
 return plaintext.strip_code()
```

File list

To improve the configurability and flexibility of our data set, let us split it up a bit further and create a set of `.txt` files where we use the record's title as filename and the record's text as the file's contents.

Since our records hold all the necessary information, this is something we can accomplish using Python's `open()`, `write()` and `close()` methods.

Listing 3.4: Easily create .txt file

```
file = open(f"{record.title}.txt", "w")
file.write(f"{record.text}")
file.close()
```

Now that we have completed the preparation of our `.txt` files, it is time to assess our finished data collection. As the figure below illustrates, we started with `22299` entries and filtered this down to `14978` files, a reduction of `32.83` %.

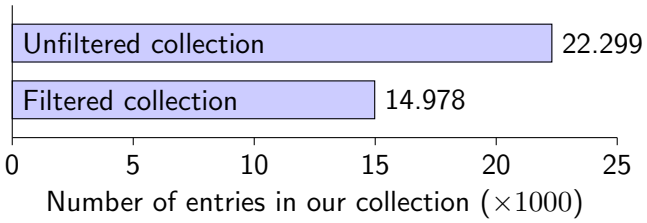

Figure 3.11: Filtering our data collection

Data Indexing

The final step in the preprocessing phase of the data is creating the search index and adding the files to it.

Creating our search index

Since we are using the Javascript client library for the IPFS HTTP API, we have chosen to use the Javascript module `elasticlunr`[2] to provide us with offline lightweight full-text search. This choice stems from the observation that this module supports a scoring mechanism and the option to save a serialized version of the index to disk. We use this module to create our search index.

Listing 3.5: Create a search index

```
1 var index = elasticlunr(function() {
2     this.addField("file");
3     this.addField("data");
4     this.setRef("hash");
5     this.saveDocument(false);
6     });
```

The option `this.saveDocument(false)` is used to configure the index not to store the original JSON documents. This reduces the index size without losing the option to return the original file. We can refer to the document by its hash as IPFS is content addressed.

Adding documents to our index

We add documents to our search index by reading the file from our file directory, adding the file to IPFS, and creating a Javascript object with the IPFS hash, filename and file's contents. This looks like this:

Listing 3.6: Add document to the search index

```
1 async function addToIndex(index, filepath, filename) {
2     var filedata = await readFilePromise(filepath, "utf8");
3     let content = ipfs.types.Buffer.from(filedata);
4     let results = await ipfs.add(content);
5     let hash = results[0].hash;
6
7     var obj = new Obj(hash, filename, filedata);
8     index.addDoc(obj);
9 }
```

Here `readFilePromise(path, encoding)` is a library function we use to get the file's contents located at `path`.

[2] `http://elasticlunr.com/`

After every new addition to the search index, we call `elasticlunr.clearStopWords()` to remove default English stop words.

Now that we have created our search index and added the files from our file directory to the index, our data is ready to be searched on.

3.4.2 Cluster Setup

Our data is processed and ready for use. It is time to move on to the cluster setup. With our setup we are building an n node, *internal* IPFS cluster. 'Internal' has been emphasized as we are not interested in connecting our cluster to the global IPFS network. We will simply emulate a cluster environment, with a layout introduced next.

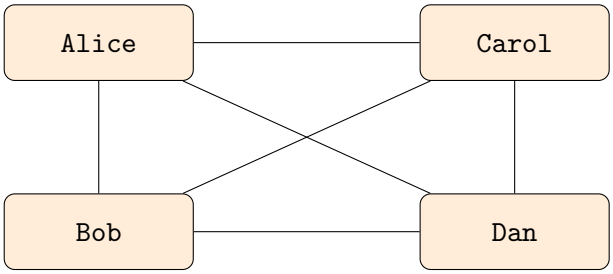

Figure 3.12: Cluster layout after completing docker setup

Prerequisites

Before we are able to launch a cluster, we need to be able to run and manage cluster peers. For this, we install 2 additional tools: `ipfs-cluster-service` and `ipfs-cluster-ctl`. Both binaries can be downloaded from `https://dist.ipfs.io/`. `ipfs-cluster-service` is used to run a cluster peer and `ipfs-cluster-ctl` is the command-line interface used to manage a cluster peer.

In order to be able to store the configurations and states of our nodes permanently, our cluster expects a 'compose' subfolder. Its directory tree looks like this:

```
compose/
|-- cluster0
|-- cluster1
|-- cluster2
|-- cluster3
|-- ipfs0
|-- ipfs1
|-- ipfs2
|-- ipfs3
```

During the first start, default configurations are created for all peers.

The Cluster

We create our cluster environment using Docker[3]. The Docker documentation page on the IPFS cluster website [12] provides us with templates for our `Dockerfile` and `docker-compose.yml` configuration files.

Dockerfile

A Dockerfile is a configuration file which can be used to tell Docker how to build a specific image. It includes all the commands a user would otherwise have to manually execute. We have modified the template to include the required Javascript libraries, e.g., the client library for the IPFS HTTP API and Elasticlunr, and execute the daemon subcommand to enable PubSub. We use the Dockerfile to build a custom Docker image and we include the resulting image ID in our `docker-compose.yml` to specify what image to base our containers on.

Docker Compose

"Compose is a tool for defining and running multi-container Docker applications.". We use this to define our cluster setup by specifying how our nodes look like. The configuration for a single node looks like:

Listing 3.7: Node configuration

```
1  ipfs0:
2      container_name: ipfs0
3      image: 5537dbaf28d9
4      ports:
5          - "4001:4001" # ipfs swarm
6      volumes:
7          - ./compose/ipfs0:/data/ipfs
8
9  cluster0:
10     container_name: cluster0
11     image: ipfs/ipfs-cluster:latest
12     depends_on:
13         - ipfs0
14     environment:
15         CLUSTER_SECRET: ${CLUSTER_SECRET} # shell variable
16         IPFS_API: /dns4/ipfs0/tcp/5001
17     ports:
```

[3]Docker is a software tool which provides us with lightweight virtualization through the use of containers. We can define what our containers look like using a Dockerfile configuration file.

```
18          - "127.0.0.1:9094:9094" # api
19      volumes:
20          - ./compose/cluster0:/data/ipfs-cluster
```

Here `5537dbaf28d9` is the image ID of our custom Docker image. We extend the provided template to a 4 node setup to run four peers (cluster0, ..., cluster3) attached to four IPFS daemons (ipfs0, ..., ipfs3).

Now that both files are complete, to bootstrap our cluster we simply execute `docker-compose up`. This launches the appropriate docker containers and realizes the cluster setup as sketched in Figure 3.12.

We can verify that this cluster has been set up correctly using the command `ipfs-cluster-ctl peers ls`, a command to check which nodes are participating in the IPFS Cluster. The output of this command looks as follows. (We have truncated the hash length to 10 characters to improve readability):

Listing 3.8: Output of `ipfs-cluster-ctl peers ls` label

```
1  QmP1bWtmYN | 8d38dbcb5eba | Sees 3 other peers
2      > Addresses:
3          - /ip4/127.0.0.1/tcp/9096/ipfs/QmP1bWtmYN
4          - /ip4/172.22.0.8/tcp/9096/ipfs/QmP1bWtmYN
5          - /p2p-circuit/ipfs/QmP1bWtmYN
6      > IPFS: QmZBgfQS3q
7          - /ip4/127.0.0.1/tcp/4001/ipfs/QmZBgfQS3q
8          - /ip4/172.22.0.2/tcp/4001/ipfs/QmZBgfQS3q
9  QmSdS2dWcv | 4935a2c554ad | Sees 3 other peers
10     > Addresses:
11         - /ip4/127.0.0.1/tcp/9096/ipfs/QmSdS2dWcv
12         - /ip4/172.22.0.6/tcp/9096/ipfs/QmSdS2dWcv
13         - /p2p-circuit/ipfs/QmSdS2dWcv
14     > IPFS: QmYnECUVMw
15         - /ip4/127.0.0.1/tcp/4001/ipfs/QmYnECUVMw
16         - /ip4/172.22.0.4/tcp/4001/ipfs/QmYnECUVMw
17 QmVSAiDDaB | d3c0d2906dc6 | Sees 3 other peers
18     > Addresses:
19         - /ip4/127.0.0.1/tcp/9096/ipfs/QmVSAiDDaB
20         - /ip4/172.22.0.9/tcp/9096/ipfs/QmVSAiDDaB
21         - /p2p-circuit/ipfs/QmVSAiDDaB
22     > IPFS: QmRcnWHaVc
23         - /ip4/127.0.0.1/tcp/4001/ipfs/QmRcnWHaVc
24         - /ip4/172.22.0.5/tcp/4001/ipfs/QmRcnWHaVc
25 QmbZ8AaKFZ | 2e941bbfe324 | Sees 3 other peers
26     > Addresses:
27         - /ip4/127.0.0.1/tcp/9096/ipfs/QmbZ8AaKFZ
28         - /ip4/172.22.0.7/tcp/9096/ipfs/QmbZ8AaKFZ
29         - /p2p-circuit/ipfs/QmbZ8AaKFZ
30     > IPFS: Qmbs6GDGkK
31         - /ip4/127.0.0.1/tcp/4001/ipfs/Qmbs6GDGkK
```

```
32          - /ip4/172.22.0.3/tcp/4001/ipfs/Qmbs6GDGkK
```

Summary

When generating a new summary file, the main question we have to ask ourselves is:

> How do we decide which entries are included in the summary file?

How do we share what we have, without sharing *all* that we have? This is called Resource selection: "the goal of resource selection is to select a small set of resources that contain a lot of relevant documents." [13]. We are looking to extract a small set of documents to share our most relevant data.

Since we generate this summary file when starting up the Publisher, we have no idea of knowing what the nodes in the cluster will be querying for. For this reason we have decided to use the statistics provided by `stats.wikimedia.org` on 'Top viewed articles'. This gives us a list of the top 1000 most viewed articles of the last month. We filter this list (e.g. by removing `Help:` pages) and submit `sumQuery` messages for the remaining entries in the list. We use the returned `sumAnswer` messages to update our local summary file.

Please take a look at what happens when we publish the following two search queries. The first search query searches for `"Ant-Man and the Wasp"`. The second search query searches for `"Potatoes"`. Both queries are looking to retrieve documents with a score of at least 2.0.

Listing 3.9: Searching IPFS

```
1  done loading summary
2
3  Starting search for 'Ant-Man and the Wasp' ...
4  Local results scored too low
5  Published a query for 'Ant-Man and the Wasp' on the network
6
7  Starting search for 'Potatoes' ...
8  Local results were sufficient
9  [ { ref: 'Qme6nrkgqvL1sAwhgeFC9aGFx1FU25tGx8hUpKesTn9KXW',
10   score: 6.754024083362458 },
11   { ref: 'QmPxXH9xNbXxWvsF6zAYP8YJozZUfP49keWrWTNA47NFT6',
12   score: 0.5051386479740516 },
13   { ref: 'QmcPK5u2iiXn3YPHRGVuGj2GY4vnAhnmt2iyffZpCWs99R',
14   score: 0.389280527600815 },
15   { ref: 'QmSEtTcxsbCoa91xjKCGaBuwNgyz9w3XYzVuxaXrCMCnVD',
16   score: 0.31295709261084725 },
17   { ref: 'QmZcgTzafnTZd1FM7mf2v9wc8sQ5tbPcPHKfvB3cP9Mowz',
```

```
  score: 0.2249596581756685 } ]

I received answer for: 'Ant-Man and the Wasp'
Updating my response list ...
Our response list for 'Ant-Man and the Wasp' now looks like:
[ { ref: 'QmPZrjeZrSBcofPWUDL8sPkJod4CCTJ7NYe92a4U7JLszf',
    score: 1.6673388780404585 },
  { ref: 'QmeFJRGoK6YeYg4T84jzFAM8kJmAouKU7VdxnXC1YSfggS',
    score: 0.2597188628938538 },
  { ref: 'QmUiRfbuAw8ayxyghGnixYoqZsY4D1GDDyHciwHb7EwLtB',
    score: 0.2490572926268008 },
  { ref: 'QmRT1Jyb3CGHmQUgepq3vQDt6VMf6LfAe8ZCRbkMiHyU3m',
    score: 0.22425568575833124 },
  { ref: 'QmUPzRgcqFQLi4U5y2Md8PmqCvYtmwEFNPQtg9mDqfrfWP',
    score: 0.2218710340306346 } ]
```

We see that our local results for `"Ant-Man and the Wasp"` scored too low. Only *after* we observe this, do we publish the query on the network. On the other hand we are able to retrieve a document with a very high score for `"Potatoes"` locally. This saves us the overhead of publishing the query on the network. Also notice that after a while we receive an answer for `"Ant-Man and the Wasp"` with which we immediately update our response list. We log the output of this to `stdout` so the user can see what is happening.

Getting a file

Now that we are able to generate queryhits for our queries, we might also be interested in actually getting a file, instead of only seeing its rank. Since all the files that have been added to the search index have also been added to IPFS, and queryhits are referenced by the hash of the document, getting a file is a rather easy process. All we need to do is issue a `get` operation on the hash of the document we want to retrieve. This looks like this:

Listing 3.10: Retrieve a file over IPFS

```
ipfs get Qm...
```

This saves the data contained in the IPFS object referenced by the hash to our local disk.

Or if we want to implement something like this in our Javascript process:

Listing 3.11: Retrieve a file over IPFS, using Javascript

```
ipfs.get("Qm...", function(err, files) {
    files.forEach(file => {
        console.log(file.path);
        console.log(file.content.toString("utf8"));
    });
});
```

This logs the file's path and its contents to `stdout`.

Where `Qm...` is the hash of the document you are looking to retrieve.

As we can see, using our implementation we are able to support searching on the contents of a document dump consisting of Wikipedia pages, arranged in a distributed IPFS cluster. This concludes the implementation of our prototype search engine built on top of IPFS.

3.4.3 Evaluation

To be able to evaluate how our prototype performs, we constructed a list of requirements on which we have focused our implementation. We have sorted this list in descending order of importance:

- **Recall rate**
  If a document is available we want to be able to retrieve it. A recall rate of 100 % means that we are able to retrieve every available document.
- **Scalability**
  Since one of the goals of the IPFS project is to replace HTTP [14], the network needs to be able to scale. For this reason, our implementation also needs to be able to scale. We don't want to be waiting 5 minutes for a query-hit.
- **Communication overhead**
  We want to minimize the load on the network.
- **Disk size**
  Since it is a P2P network we want everyone to be able to join. We don't want our implementation to be too big for the available disk space on some devices.
- **Ideology**
  We want to keep close to the idealogy of P2P systems. We want to avoid centralization, master-slave relations and single point of failures. We also want to keep close to the idealogy of IPFS. This means that popular searches are cached and non-searched items will disappear.

Recall rate

By default, our implementation adds every file we add to our search index, to IPFS as well. We do this so we can use its hash as a reference in our queryhits. This in turn makes it easier to retrieve a file as we have discussed in Chapter 3.4.2. Since every file is made available through IPFS, as long as a node stays connected to the network, its documents are available.

Scalability

Unfortunately our cluster setup does not include scalability testing. This is something that should be included in future research. We do however employ a similar technique to ISM [15] which shows very promising results. More on this in Section 6.2.

Communication overhead

Using our summary file we try to resolve queries locally as much as we can. This saves the overhead of having to publish to the network for *every* query.

Disk size

Looking at our current cluster nodes, the average file size for our index file is 206.25 MB. The average file size for our summary file is 244.825 KB. This is small enough such that this should not be a limiting factor.

Note however that the size of the index file heavily correlates with the size of the data you are sharing with the network. On devices with a smaller data source the index will naturally be smaller as well.

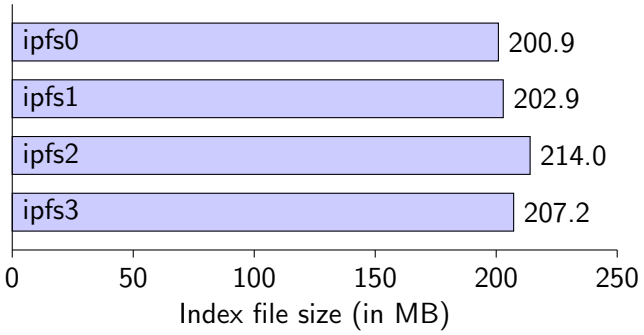

Figure 3.13: Measuring the file size of the local index

A future improvement to be able to limit the size of the summary file could be made by implementing a check in our `exitHandler` to only write as many entries to disk as allowed.

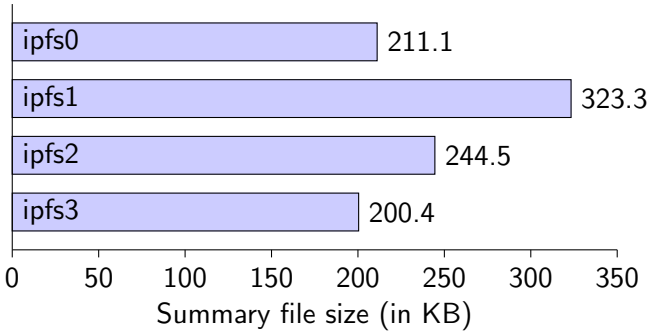

Figure 3.14: Measuring the file size of the local summary

This should allow devices with limited hardware to participate in searching over the contents of the cluster.

Ideology

In the current version of our implementation every node has equal rights. We have employed a true decentralized solution without introducing single point of failures, centralization or master-slave relations.

A future addition that would move even closer to the idealogy of IPFS would be to incorporate a resource selection algorithm that uses the published `query` and `answer` messages as input, as proposed in Section 6.1.2. As a result of this, popular searches will be cached in the summary file and non-searched items will disappear from the network.

Chapter 4

Related Work

This Chapter elaborates on a few similar projects to demonstrate where our research fits into the bigger picture. Instead of providing an exhaustive list we try to focus on the more interesting projects or projects that are otherwise worth mentioning.

4.1 YaCy

YaCy[1] is a free, open and decentralized search engine. The software is developed by the YaCy community on Github. They employ a crawling based method where peers parse, index and store a local search index. This local search index is merged with other peers' indices on the same network to provide search results. Index fragments are exchanged using a distributed hash table. This project is probably the most advanced decentralized search project at the moment.

The main differences compared to our approach, is that instead of swapping index fragments, we swap query results. We also allow for a higher peer autonomy. Since we do not employ a crawling method, peers are free to join and leave the network whenever they want.

4.2 Intelligent Search Mechanism (ISM)

ISM [15] is a P2P information retrieval technique which aims to improve speed and efficiency by constructing a *profile mechanism* and a *relevance rank*. A profile mechanism is a list which "maintains T most recent queries and the corresponding `QueryHits`, along with the number of results.". The relevance rank uses this profile to determine which neighbouring node to

[1] `https://yacy.net/`

forward a query to. It shows to be a very promising technique addressing scaling issues which flooding techniques have to deal with. The biggest disadvantage of this technique is that it has quite a long startup time before it is able to attain a high recall rate [15].

This is something our implementation tries to address by supplying an initial summary file before the first query has even been published on the network. After incorporating a resource selection algorithm as proposed in Section 6.2 it would be very interesting to see how our implementation compares to ISM as both techniques are quite similar. Unfortunately no public implementation is available.

4.3 Presearch

Presearch[2] is a decentralized search engine currently in early access, providing curious users the option to join the beta program. It is a promising project with ambitious goals. Apart from their whitepaper, technical details are unfortunately scarce. They seem to be using a browser based crawling method based on the user's history. To incentivize and reward participation they have created the PRE (Presearch Token). These tokens are issued to members as they search, and can be used by sponsors to place advertisements. For further reading, take a look at the presearch whitepaper[3].

4.4 PeARSearch

PeARSearch[4] is the decentralized version of PeARS[5]. It is a prototype decentralized Web search system under construction. It is a search engine that you can query without ever publishing your query on the network. The main idea is that users collect their favorite web pages and create a local 'pod', a collection of indexed web pages that share a topic. Pods can then be exported and shared through any medium. Users can import pods and add them to their local search index collection. When searching for something, results from a user's local pods are merged and returned. This way your actual search query never leaves your computer.

This is a fantastic idea. It however requires an active community collecting webpages and sharing and maintaining pods. Unfortunately this project has not yet gained any traction. Currently there are only a select few pods available which makes it not something someone would use.

[2] `https://www.presearch.io/`
[3] `https://www.presearch.io/uploads/WhitePaper.pdf`
[4] `https://github.com/PeARSearch/PeARS-orchard`
[5] `http://pearsearch.org/`

4.5 ipfs-search

ipfs-search[6][7] is a project which aims to provide a search engine for IPFS by sniffing the DHT gossip and indexing the file and directory hashes. When listening to the DHT gossip, discovered hashes are added to a queue. A crawler uses this queue to list the type of the IPFS object. If it is a file, its metadata is extracted and added to the index. If it is a directory, its referred items will be added to the queue. Searching is implemented using Elasticsearch[8].

Since it is listening to the DHT gossip between its own node, and is using the crawling approach, this project will have to face a significant challenge as IPFS grows. It is not a decentralized solution nor is it scalable.

[6] `http://ipfs-search.com/`
[7] `https://github.com/ipfs-search`
[8] `https://www.elastic.co/products/elasticsearch`

Chapter 5

Conclusions

IPFS is a P2P hypermedia network protocol with very ambitious goals. It aims to replace HTTP and make the Web faster, safer and more open. Before it can realize any of these goals, it will need to be adopted and used. We observe that the addition of searching functionality to IPFS is an essential aspect in the project's adoption and future. Therefore, the goal of this thesis has been to present a prototype search engine built on top of IPFS and provide an answer to the question:

> If we were to have a document dump consisting of Wikipedia pages, how do we arrange this collection of pages in a distributed IPFS cluster with 1, 2, ..., n clients so that we can support searching on the contents?

Our prototype relies on the existing PubSub implementation to create a shared channel on which nodes can publish events to talk to each other. Using these events, queries and their corresponding queryhits can be exchanged over the network.

We also propose the use of a summary. This file contains a summary of the data that is available in the cluster. Using a summary file we can increase the speed, and decrease the communication overhead with which queries can be resolved.

As explained in Chapter 3, using our implementation we are able to support searching on the contents of a document dump consisting of Wikipedia pages, and locate assets arranged in a distributed IPFS cluster. This concludes the implementation of our prototype.

Chapter 6

Discussion and Future Work

This Chapter discusses our assumptions and the strengths and weaknesses of the current implementation.

6.1 Discussion

In this Section we will talk about some of our remarks with regards to our data, our cluster and our prototype.

6.1.1 Data

When adding documents to our search index, the plaintext files are processed using Elasticlunr's tokenizer. This tokenizer splits strings into tokens. By default this is configured to split strings on whitespace and hyphens. When observing our cleaned text files we see that there is still some WikiMedia syntax appearing here and there. This could skew the inverted index positions of certain words a little. Fortunately most (almost all) of our files are processed properly and therefore the effect this will have on the overall search quality will be minimal.

In the current version of our implementation we assume that our corpus is static. We do not check for updates in our text nor do we check to see if there are new files. We could however enable having dynamically changing data by checking if files have been updated, and adding the new versions to our search index. Since IPFS supports versioning using commit objects, whenever we have a new version of a file, we can simply add it to IPFS and the network will take care of the rest.

6.1.2 Prototype

The current version of our summary file only uses `sumAnswer` and `sumQuery` events to kickstart the summary file. This has to do with how we generate 'interesting' queries to populate this file. Remember our resource selection question:

> How do we share what we have, without sharing *all* that we have?

How do we know which of our documents are relevant without knowing what the nodes in the cluster are searching for? Resource selection algorithms need to have this knowledge to determine what the most relevant documents are. As a future improvement we could use the published `query` and `answer` messages as input for a resource selection algorithm to determine a node's most relevant local set of documents. Using this knowledge we could use the `sumAnswer` and `sumQuery` events to push summary updates to the network. This could possibly increase the efficiency of using a summary file.

6.2 Future Work

As mentioned in Section 3.3, using our summary file we try to save as much of the overhead of publishing to the network when trying to resolve a query. However the decrease in communication overhead is only as much as the quality of the summary file. Therefore it would be a very interesting research topic to incorporate different resource selection algorithms in our prototype to try and increase the relevance of the documents in the summary file, such that we can resolve as much locally as possible.

Bibliography

[1] Benet J. IPFS-content addressed, versioned, P2P file system. arXiv preprint arXiv:14073561. 2014;.

[2] Reuters in Ankara. Turkey blocks Wikipedia under law designed to protect national security; 2018. "[Online; accessed 23-November-2018]". https://www.theguardian.com/world/2017/apr/29/turkey-blocks-wikipedia-under-law-designed-to-protect-national-security.

[3] Verborgh R. Decentralizing the Semantic Web through incentivized collaboration. In: van Erp M, Atre M, Lopez V, Srinivas K, Fortuna C, editors. Proceedings of the 17th International Semantic Web Conference: Blue Sky Track. vol. 2189 of CEUR Workshop Proceedings; 2018. [Online; accessed 8-October-2018].

[4] Berners-Lee ST. Three challenges for the web, according to its inventor; 2017. [Online; accessed 5-October-2018]. https://webfoundation.org/2017/03/web-turns-28-letter/.

[5] Simonite T. The decentralized internet is here, with some glitches; 2018. "[Online; accessed 5-October-2018]". https://www.wired.com/story/the-decentralized-internet-is-here-with-some-glitches/.

[6] libp2p. gossipsub: An extensible baseline PubSub protocol;. Available from: https://github.com/libp2p/specs/tree/master/pubsub/gossipsub#controlling-the-flood.

[7] Lai Z, Liu C, Lo E, Kao B, Yiu SM. Decentralized Search on Decentralized Web. arXiv preprint arXiv:180900939. 2018;.

[8] Protocol Labs. Filecoin: A Decentralized Storage Network; 2018. [Online; accessed 18-October-2018]. https://filecoin.io/filecoin.pdf.

[9] Raftopoulou P, Petrakis EGM. iCluster: A Self-organizing Overlay Network for P2P Information Retrieval. In: Macdonald C, Ounis I, Plachouras V, Ruthven I, White RW, editors. Advances in Information

Retrieval. Berlin, Heidelberg: Springer Berlin Heidelberg; 2008. p. 65–76.

[10] Wikipedia, Wikimedia Foundation. Database Download;. Available from: https://en.wikipedia.org/wiki/Wikipedia:Database_download.

[11] Wikipedia, Wikimedia Foundation. Help:Wikitext;. Available from: https://en.wikipedia.org/wiki/Help:Wikitext.

[12] IPFS Cluster. Docker;. Available from: https://cluster.ipfs.io/documentation/deployment/docker/.

[13] Si L, Callan J. The Effect of Database Size Distribution on Resource Selection Algorithms. In: Callan J, Crestani F, Sanderson M, editors. Distributed Multimedia Information Retrieval. Berlin, Heidelberg: Springer Berlin Heidelberg; 2004. p. 31–42.

[14] Protocol Labs. IPFS is the Distributed Web;. Available from: https://ipfs.io/.

[15] Zeinalipour-Yazti D, Kalogeraki V, Gunopulos D. Information retrieval techniques for peer-to-peer networks. Computing in Science & Engineering. 2004;6(4):20–26.

Appendix A

Appendix

A.1 Data Cleaning

Here are some common examples of data entries we have removed from the data set or which we have filtered into a cleaned version.

A.1.1 WikiMedia Format

These are examples of the WikiMedia format which we have tranformed to its plaintext equivalent.

```
=See also==
&lt;!-- Please keep entries in alphabetical order &amp;
     add a short description [[WP:SEEALSO]] --&gt;
{{Div col|colwidth=20em|small=yes}}
* [[Cool roof]]
* [[Daisyworld]]
* [[Emissivity]]
* [[Exitance]]
* [[Global dimming]]
* [[Irradiance]]
* [[Kirchhoff's law of thermal radiation]]
* [[Opposition surge]]
* [[Polar see-saw]]
* [[Solar radiation management]]
{{div col end}}
&lt;!-- please keep entries in alphabetical order --&gt;
```

```
=References==
{{Reflist|refs=
&lt;ref name=&quot;Goode&quot;&gt;{{Cite journal
    |last=Goode |first=P. R. |date=2001 |title=Earthshine
    Observations of the Earth's Reflectance
    |journal=[[Geophysical Research Letters]] |volume=28
    |issue=9 |pages=1671-1674
    |url=http://www.agu.org/journals/ABS/2001/2000GL01
2580.shtml |doi=10.1029/2000GL012580 |bibcode =
    2001GeoRL..28.1671G |display-authors=etal}}&lt;/ref&gt;
```

A.1.2 Redirect pages

These pages include nothing but a redirect to another wikipedia article.

Listing 1: Redirect page for "Computer accessibility"

```
<page>
    <title>AccessibleComputing</title>
    <ns>0</ns>
    <id>10</id>
    <redirect title="Computer accessibility" />
    <revision>
        <id>854851586</id>
        <parentid>834079434</parentid>
        <timestamp>2018-08-14T06:47:24Z</timestamp>
        <contributor>
            <username>Godsy</username>
            <id>23257138</id>
        </contributor>
        <comment>remove from category for seeking
            instructions on rcats</comment>
        <model>wikitext</model>
        <format>text/x-wiki</format>
        <text xml:space="preserve">#REDIRECT [[Computer
            accessibility]]

{{R from move}}
{{R from CamelCase}}
{{R unprintworthy}}</text>
        <sha1>42l0cvblwtb4nnupxm6wo000d27t6kf</sha1>
    </revision>
</page>
```

A.1.3 Disambiguation pages

These pages include a definitions list. They do not contain actual text regarding the page title and therefore have been excluded in the filtered collection.

Listing 2: Disambiguation page for "Austin"

```
<page>
    <title>Austin (disambiguation)</title>
    <ns>0</ns>
    <id>590</id>
    <revision>
        <id>842727923</id>
        <parentid>841549643</parentid>
        <timestamp>2018-05-24T08:36:07Z</timestamp>
        <contributor>
            <username>Crouch, Swale</username>
            <id>11009441</id>
        </contributor>
        <minor />
        <comment>per [[MOS:DABPRIMARY]]</comment>
        <model>wikitext</model>
        <format>text/x-wiki</format>
        <text xml:space="preserve">{{wiktionary|Austin}}
'''[[Austin]]''' is the capital of Texas in the United
    States.

'''Austin''' may also refer to:

{{TOC right}}

==People names ==
* [[Austin (name)]] - a short form of Augustin, or
    Augustine
** [[Augustin (disambiguation)]]
** [[Augustine (disambiguation)]]
** [[August (disambiguation)]]

==Geographical locations==

===Australia===
* [[Austin, Western Australia]]

...
```

A.2 Namespaces

The following list dictates which namespace key corresponds to which category. It consists of the `<siteinfo>` data from the `enwiki-20181020-pages-meta-current1.xml-p10p30303.bz2` file.

Listing 1: Namespace key list

```
<siteinfo>
  <sitename>Wikipedia</sitename>
  <dbname>enwiki</dbname>
  <base>https://en.wikipedia.org/wiki/Main_Page</base>
  <generator>MediaWiki 1.32.0-wmf.26</generator>
  <case>first-letter</case>
  <namespaces>
    <namespace key="-2" case="first-letter">Media</namespace>
    <namespace key="-1" case="first-letter">Special</namespace>
    <namespace key="0" case="first-letter" />
    <namespace key="1" case="first-letter">Talk</namespace>
    <namespace key="2" case="first-letter">User</namespace>
    <namespace key="3" case="first-letter">User talk</namespace>
    <namespace key="4" case="first-letter">Wikipedia</namespace>
    <namespace key="5" case="first-letter">Wikipedia talk</namespace>
    <namespace key="6" case="first-letter">File</namespace>
    <namespace key="7" case="first-letter">File talk</namespace>
    <namespace key="8" case="first-letter">MediaWiki</namespace>
    <namespace key="9" case="first-letter">MediaWiki talk</namespace>
    <namespace key="10" case="first-letter">Template</namespace>
    <namespace key="11" case="first-letter">Template talk</namespace>
    <namespace key="12" case="first-letter">Help</namespace>
    <namespace key="13" case="first-letter">Help talk</namespace>
    <namespace key="14" case="first-letter">Category</namespace>
    <namespace key="15" case="first-letter">Category talk</namespace>
    <namespace key="100" case="first-letter">Portal</namespace>
    <namespace key="101" case="first-letter">Portal talk</namespace>
    <namespace key="108" case="first-letter">Book</namespace>
    <namespace key="109" case="first-letter">Book talk</namespace>
    <namespace key="118" case="first-letter">Draft</namespace>
    <namespace key="119" case="first-letter">Draft talk</namespace>
    <namespace key="710" case="first-letter">TimedText</namespace>
    <namespace key="711" case="first-letter">TimedText talk</namespace>
    <namespace key="828" case="first-letter">Module</namespace>
    <namespace key="829" case="first-letter">Module talk</namespace>
    <namespace key="2300" case="first-letter">Gadget</namespace>
    <namespace key="2301" case="first-letter">Gadget talk</namespace>
    <namespace key="2302" case="case-sensitive">Gadget definition</namespace>
    <namespace key="2303" case="case-sensitive">Gadget definition talk</namespace>
  </namespaces>
</siteinfo>
```