# Building a Content-Addressed Filesharing System

Dissertation

**Student**

Nadav Rahimi

**Supervisor**

Professor Graham Hutton

School of Computer Science
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text: N.R.

## Abstract

Internet exchanges are consistently reaching new peaks of traffic flow due to surges in traffic. Specifically, this occurs when a group of clients all request data from a similar set of origin servers, which causes network congestion across many contention points in the request path.

Peer-to-peer networks boast resilience and performance whilst being trivial to implement and low-cost to operate. Some networks also use content-addressing, a method where data is uniquely identified via its computed fingerprint, this makes it trivial to archive, identify and propagate across peers.

We propose a new filesharing system where clients exchange data as part of a peer-to-peer content-addressed network in order to reduce network congestion. Critically, our implementation is scalable, performant, resistant to peer failures, and able to decrease network congestion.

## Acknowledgements

Thank you to my supervisor, Professor Graham Hutton, who always kept me grounded.

Thank you to Luke and Gary, who took time out of their day to provide me with their invaluable experience and advice.

Thank you to Abed and Dan, who put up with my rambling late into the night.

Thank you to my family who have always stood behind me.

And finally, thank you to Mark, who set me on this path; I'm grateful for your mentorship and guidance.

# Contents

# 1 Introduction

There are two types of networking traffic: north-south traffic and east-west traffic. North-south traffic originates at end users and arrives at origin servers, whereas east-west traffic travels between peers, (i.e. between users or between origins) [1]. Intuitively, client-server networking architectures promote north-south traffic and peer-to-peer architectures promote east-west traffic.

Recently, Internet exchanges are consistently setting records for peak north-south traffic flows [2] [3] [4], this occurs when a group of clients simultaneously request content from the same set of origin servers, subsequently congesting the network at different contention points in the request paths. In practical terms, this often occurs when a new update is released for a massively popular online game, (e.g. new Fortnite updates have consistently set peak traffic records [2]).

To reduce this load it's necessary to reduce north-south traffic; this can be done by instead promoting east-west traffic, which encounters fewer points of contention. One approach is to instruct the peers in a peer-to-peer network to exchange the content which they've already downloaded from a northbound server amongst themselves, prioritising for peers which are part of the same autonomous system; this reduces the number of contention points a request hits, thereby lowering the congestion in the network.

This system remains a non-trivial task to implement since since peer-to-peer networks are much less secure than their client-server counterparts [5] [6]. This is due to the lack of centralised authority and coordination in the network, which makes it vulnerable to malicious actors who are operating their own set of peers.

A possible technique to ensure security, specifically, to validate the integrity of data exchanged, is to use a Content Addressing (CA) scheme. CA is a method by which any arbitrary blob of data can be identified via a globally unique ID derived from its binary representation [7]. Importantly, CA is tamper-resistant, meaning it can be used to secure communications between peers in the network, (i.e. peers are able to validate the integrity of blobs they request from other peers through the use of CA).

## 1.1 Motivation

As the proliferation of Internet-connected devices increases, it becomes ever more important to reduce network congestion – a problem which plagues both end users and network operators. I hope to present a novel-yet-effective solution to this problem, which will also provide me with the ability to explore the concepts of networking and distributed systems more; a content-addressed filesharing system presents a clear self-contained idea which can be developed and evaluated.

## 1.2 Aims and Objectives

The aim of this project is to implement a content-addressed filesharing system with the following properties:

1. The system should utilise a peer-to-peer network for content discovery
2. The system should use content addressing to identify files in a secure manner
3. The system should optimise file exchange in a way which reduces network congestion
4. The system should be robust and adaptive, i.e. scalable, performant, reliable
5. The implementation should be maintainable and simple to understand

After this, the key aim of this project is to evaluate the performance and reliability of the system in a way which is consistent with past literature.

# 2 Related Work

This section reviews important properties of successful peer-to-peer networks in their purpose to share files. We contrast BitTorrent (a specialised filesharing system) with IPFS (a general-purpose content-addressed distributed file system).

## 2.1 BitTorrent

BitTorrent is a peer-to-peer filesharing protocol used to distribute files over the internet in a decentralised manner [8]. It is one of the most popular protocols for this purpose, having attracted over 150 million active users as of 2012 [9].

The protocol has four main components:

**Torrents** Torrents define a session of transfer of a file to a set of peers. They contain meta-information about the file to be downloaded. Peers involved in a torrent cooperate to replicate the file among each other using swarming techniques [10].

**Pieces** Files are split into chunks called pieces, which are exchanged with multiple peers simultaneously: as each peer receives a piece, it becomes the source of that piece for other peers, enabling efficient dissemination of pieces throughout the network. Thus, the capacity of systems increases with the number of active peers enabling highly scalable file distribution [11]. Additionally, the torrent specifies the cryptographic hash of each piece, meaning the modification of a piece can be reliability detected and prevented [8].

**Trackers** A centralised tracker is used to enable coordination between peers. If the tracker fails or is unreachable, the system becomes unavailable to new peers, so they cannot obtain the file or contribute resources to the system [11].

**DHT** A (newer) alternative form of peer discovery is available in the form of a Distributed Hash Table (DHT), which provides a lookup service for piece-peer pairs [12], (i.e. you are able to look up which peers own a specific piece and then communicate with them directly in order to download the piece). The most popular BitTorrent DHT is known as Mainline, which, on average, has 15 to 27 million concurrent users online at once [13]. Albeit slower, the DHT reduces the reliance on a central tracker.

Even though BitTorrent is an effective means of filesharing, it is very rigid, which makes it unwieldy to use in certain contexts due to its reliance on torrent files; consider these issues:

- An external system must be used to track torrent files
- If multiple torrent files contain the same piece of content, it is not deduplicated
- It is not possible to be specific about content you are sharing, (i.e. sharing a subset of files within a torrent), without creating multiple torrent files

## 2.2 IPFS

The InterPlanetary File System (IPFS) is an open-source content-addressed peer-to-peer network which provides distributed data storage and delivery [14]. IPFS has millions of daily content retrievals and already underpins dozens of third-party applications [14]. IPFS has a stronger focus on being a distributed file system than just being a filesharing system, which is reflected in its general-purpose design.

We focus on the design of the IPFS white paper released in 2014 since it describes a simpler system to analyse and study in comparison to the complicated and convoluted IPFS implementation used today. Recently-added IPFS capabilities better improve its functionality as a global file system, but this is irrelevant to our study because we are looking to build a filesharing system instead.

**Merkle DAG** IPFS represents files using Merkle Directed Acyclic Graphs [14] for two reasons: (1) the root nodes of these DAGs can be used as CA identifiers, and (2) content which is shared between multiple files is deduplicated in addition to only storing one instance of each file. Also, they can be used to efficiently verify that blocks of data received from other peers in the peer-to-peer network are authentic. An important consideration is that it is non-trivial to update the contents of the tree [15], thus, they're unsuited for use in systems which address primarily dynamic content. We specify their design in Section 4.3.

**Distributed Hash Tables** IPFS uses the Kademlia DHT spec to provide an efficient lookup service for key-value pairs [16]. Kademlia [17] is a popular specification that provides (1) efficient lookup through massive networks, (2) low co-ordination overhead, (3) resistance to attacks by preferring long-lived nodes and (4) wide usage in peer-to-peer applications, including Gnutella, forming networks of over 20 million nodes [18] [13]. IPFS uses an extended version of the Kademlia DHT to facilitate a DAG-node lookup service – the clear benefit being no reliance on some centralised tracker. Crucially, IPFS stores the DAG nodes themselves on the DHT, whereas BitTorrent stores the peers which hold the DAG nodes.

Even though IPFS provides a versatile means of filesharing, it's inherent design makes it problematic in certain situations; consider these issues:

- It remains challenging to write and maintain an IPFS implementation due to its complex design: the reference Go implementation currently spans thousands of lines of code across multiple repositories [19] [20]
- The majority of IPFS and BitTorrent clients use a singular global DHT in addition to proactively sharing content around the network, (this sharing is required for correct system operation). Since the IPFS DHT stores file contents, instead of just file locations, users are susceptible to downloading illicit content, which represents a clear security risk.

## 2.3 Contributions

As seen in the previous two sections, both systems are subject to their own set of disadvantages: BitTorrent enjoys popular growth at the expense of rigidity, on the other hand, IPFS is able to overcome these disadvantages at the cost of being a much more complex system which is not able to reach the same efficacy in terms of content lookup and retrieval.

This project aims to find a middle ground between these two technologies by creating an BitTorrent-inspired system which is able to overcome the disadvantages of BitTorrent's rigidness. This system should be specialised for filesharing in a flexible manner, in addition to reducing network congestion through methods which BitTorrent does not currently consider.

More specifically, the system should:

1. Have no reliance on some external tracking mechanism
2. Provide more granular control over which files are shared
3. Deduplicate across all files shared using the system, (not just the contents of singular transfer)
4. Be simple to reason about and implement

# 3 Design

In this section, we specify the requirements of our system, and explain how they are addressed by the design of the system's architecture and operation.

## 3.1 Requirements

We define the specific functional and non-functional requirements that our system should have. For these, we considered the initial objectives defined in sections 1.2 and 2.3, in addition to characteristics of successful distributed systems described in past literature.

Key literature included:

- Amazon Dynamo [21] – a seminal work in the field of distributed systems
- Kademlia [17] – the DHT used by both BitTorrent and IPFS
- DS: Principles and Paradigms [16] – an influential textbook on the topic

Table 3.1: Functional Requirements

| ID | Description |
|---|---|
| 1 | Our system should be composed of a peer-to-peer network |
| 2 | The system should use content addressing to identify files |
| 3 | There should be granular control over which files are shared |
| 4 | There should be no additional tracking mechanism for sharing files |
| 5 | Deduplication should occur across all files shared |

Table 3.2: Non-Functional Requirements

| ID | Description |
|---|---|
| 6 | Simplicity - The system is simple to reason about and implement |
| 7 | Scalability - The system should be able to scale out one peer at a time with minimal impact to its operation |
| 8 | Transparency - The underlying complexity of the system should be hidden from the peers who are exchanging content |
| 9 | Heterogeneity - The components in the system can work together even if their technology varies |
| 10 | Fault Tolerance - The content exchange process can cope with partial failure of the system |
| 11 | Security - The system is secure from threats, especially malicious peers who attempt to disrupt the content exchange process |
| 12 | Congestion Reduction - The content exchange process optimises for the reduction of network congestion |

## 3.2 Architecture

We can create a simple BitTorrent-like filesharing system by considering three components: (1) peers which exchange content, (2) a peer lookup service to discover nearby peers, and (3) a communication protocol.

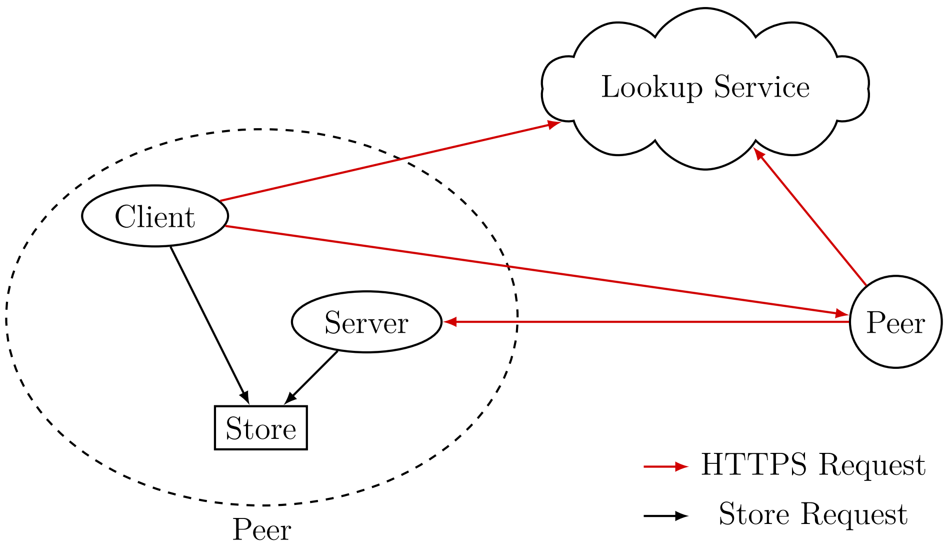

Figure 3.1: System Architecture

**Lookup Service** To find nearby peers which offer a sought piece of content we use the Kademlia DHT previously mentioned in Chapter 2. Kademlia has proven to be a scalable system which supports millions of nodes whilst still being performant. Its ease-of-use lies in the fact that it is both access and location transparent: users of Kademlia do not need to know (1) which nodes are responsible for hosting which files, and (2) the physical locations of other nodes. Kademlia is also fault tolerant since it replicates data across nodes for redundancy, meaning that if a node fails, its data can still be served by its neighbours. Finally, due to its preference for long-lived nodes, Kademlia is resistant to different types of attacks including denial-of-service attacks. Extending the Kademlia implementation permits resistance to more types of attacks including Sybil attacks [22].

**Protocol** Our system uses the HTTPS protocol to communicate between peers and DHT nodes. HTTPS is one of the most popular standards for communication protocols: it is currently implemented in over 30 different browsers and programming languages [23] [24]; thus, using HTTPS means the system can operate in spite of heterogeneous components. Additionally, communication is guaranteed to be reliable and secure due to the use of both TCP/IP and TLS, which is an important quality of service metric for this system. HTTPS is also compatible with many modern HTTP load balancers and caches, which are able to increase the scalability, performance and reliability of the system for a low cost by utilising existing CDN infrastructure. Finally, HTTPS is a stateless protocol, making it simpler to implement in comparison to a stateful protocol such as TCP which require complex managing of session state.

**Peer** Peers are comprised of three internal components: (1) a HTTPS client to communicate with the lookup service and other peers, (2) a HTTPS server through which other peers can request files, and (3) a store for content addressed data. The storage format chosen was a Merkle DAG inspired by IPFS which would also provide total deduplication across all files shared; this would provide granular control over which files are shared in a content exchange.

We further address how the requirements are met in Chapter 4 where we discuss the implementation of the system.

## 3.3 Operation

The system relies on two public APIs, one provided by the lookup service to find nearby peers, and another provided by peers to facilitate content exchange. We describe the APIs and the content exchange process below.

Table 3.3: Lookup Service API

| Operation | Description |
|---|---|
| `Notify` | Notify the lookup service that we own a DAG node and the port by which we are accessible on |
| `FindPeers` | Ask for nearby peers which own a given DAG node |

Table 3.4: Peer API

| Operation | Description |
|---|---|
| `Node` | Request a DAG node from another peer given the node's content addressed ID |

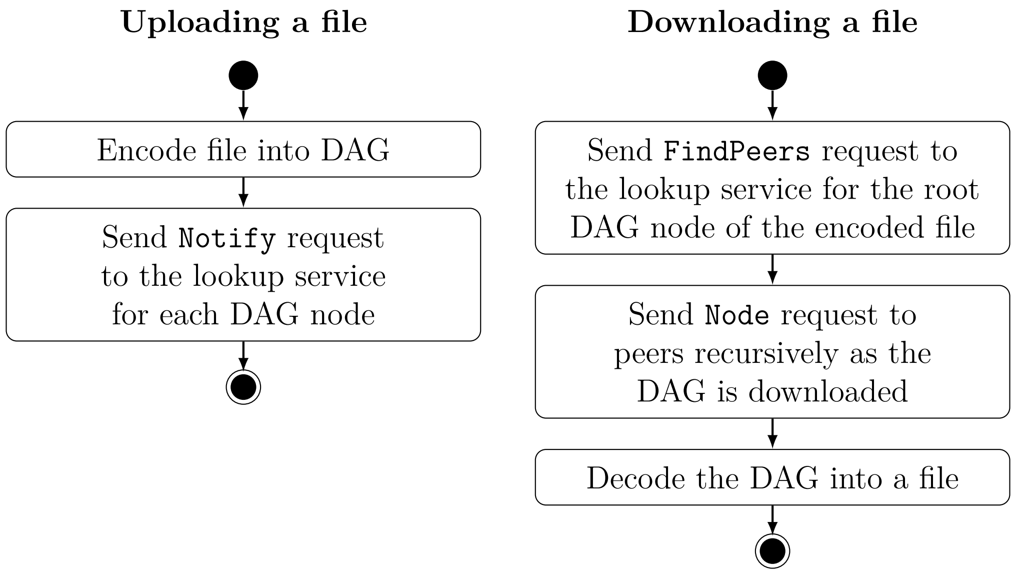
Uploading a file
Encode file into DAG
Send Notify request to the lookup service for each DAG node
Downloading a file
Send FindPeers request to the lookup service for the root DAG node of the encoded file
Send Node request to peers recursively as the DAG is downloaded
Decode the DAG into a file

Figure 3.2: Process for uploading and downloading files

## 3.4 Discussion

We briefly discuss certain choices which influenced the architecture and operation of the system, in addition to certain drawbacks which the design contains including how they can be addressed.

**Existing HTTP-based Architecture** The peer API is non-volatile compared to the lookup service API (i.e. the DAG encoding of a file is unlikely to change whereas the nearby peers which are available to share a file could change often). Due to this, we anticipate that placing peers behind CDNs will improve the performance of the system by caching DAG nodes on existing CDN infrastructure in addition to the peer-to-peer network.

**End-to-End Connectivity** Our scheme relies on complete end-to-end connectivity throughout the network, however, a significant portion of peers who would participate in this network, (specifically home users), reside behind routers which perform Network Address Translation (NAT), making them unavailable to participate in the peer-to-peer network. This can be solved using some kind of NAT traversal technique such as Interactive Connectivity Establishment (ICE) [25]; this task is left as a future extension for the system.

**Spoofing** We only allow peers to specify which port they are accessible on, and not which IP they are accessible by. The reasoning behind this was to prevent IP spoofing, which could lead to some form of denial of service attack.

**Peer-DHT Separation** Classical filesharing implementations such as BitTorrent specify that each peer should participate in the lookup service by operating its own DHT node. However, in our specification there is a clear dichotomy between peers (who are end users), and the lookup service (which we anticipate to be operated by network providers). This prevents malicious actors from participating in the DHT and propagating illicit content which network providers would be accountable for. Some mutual authentication scheme, (such as mTLS [26]), is needed so that each DHT node can verify that it's communicating with a valid peer also operated by the network provider; this task is left as a future extension for the system.

# 4 Implementation

The Go programming language was chosen to implement the system since it, (a) has a proven track record for large-scale distributed systems, and, (b) is the language of choice for the main IPFS implementation [19] and some popular BitTorrent DHT implementations [27].

Throughout this section we describe the implementation for each low-level component of the system; note that the name of the implementation is `inu` which you may see referenced throughout subsequent chapters. At the end we discuss the development process, in addition to Go's ability to build maintainable, simple, and correct software.

## 4.1 Content Identifiers

We define a content addressable identifier, which we call a CID. We use this to identify blobs of content which are propagated across the system.

```
type CID string
```

Listing 4.1: Definition of a CID

We apply the SHA256 hash function to some blob to generate its respective CID. SHA256 is suitable because it's (relatively) fast and cryptgraphically secure [28]; a requirement for CA systems. The hash digest is represented using Base32, (in lieu of more compact encodings such as Base64), because Base32 is case-insensitive, making it simpler and more readable.

```
> NewCID("Shiba Inu")
PUBFTB7DJM2PH4WOJKL5635AAZWZHRKQUEBDURMUZ5YPWAFK74GQ
```

Listing 4.2: Example CID

## 4.2 Block Store

Blocks group together blobs of bytes together with their CID so that the implementation doesn't waste time re-computing a blob's CID on demand. The store provides a standard `Get/Put/Delete` API for blocks.

```
type Block struct {
  CID  inu.CID
  Data []byte
}
```

Listing 4.3: Definition of a block

Blocks are stored together in an SQLite store. SQLite was chosen because it is fast, reliable, and requires no configuration or maintenance [29].

## 4.3 Merkle DAG

Merkle DAGs are $n$-ary trees with no balance requirement [14]. Each leaf's value is the SHA256 hash of some binary data $x_i$ which is part of an ordered list of elements $x_1, x_2, ..., x_n$. Ascending the tree, internal nodes correspond to the hash of the concatenation of their children, until eventually a root is formed [15]. Critically, after a file has been DAG-encoded, the root node can be used as its CA identifier. DAG nodes are serialised into blocks before being stored in the block store.

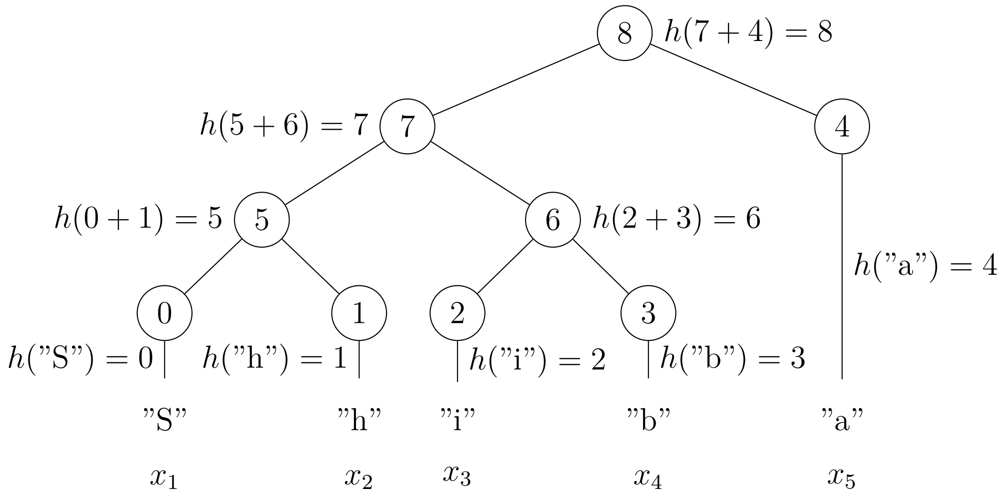

Figure 4.1: Merkle DAG for the word "Shiba" using hash function $h$

### 4.3.1 Edges

Links are used to represent the edges of the Merkle DAG. Each link has an optional `name` attribute which may be used to encode more complex structures into the DAG.

```
type Link struct {
    // Optional name
    Name string

    // CID of the target node's binary representation
    CID inu.CID
}
```

Listing 4.4: Definition of a link

### 4.3.2 Vertices

A `Node` represents vertex in the Merkle DAG. To serialise nodes, the JSON format is used since (1) it is the standard for transferring data in HTTP APIs and (2) it has first-class support in the Go standard library.

```
type Node struct {
    // Links to child nodes in the DAG
    Links []Link

    // Content data
    Data []byte
}
```

Listing 4.5: Initial definition of a node

#### Discussion

Throughout the implementation, a DAG nodes are (constantly) serialised, since their binary representation is required for calculation of their CID. Initially, nodes were serialised on demand, however, this led to an inconvenient API where many functions were required to handle the effect of a failed serialisation, greatly increasing the code's complexity and reducing its readability.

Instead, the implementation provides a compile-time guarantee that a node is serialisable at the cost of increased memory usage. As the manual creation of nodes is inhibited, a `NodeBuilder` API is supplied. The builder serialises the node during its creation and then caches this serialised representation inside the node itself; if the node is unserialisable the builder fails. This has proved effective in creating an elegant API by reducing the effectful handling of nodes solely to node creation.

## 4.4 File System Abstraction

In order to support multi-file sharing, we define a file system abstraction which is DAG-encodable and can represent both files and directories. Each DAG node stores a discriminated union which represents either (a) raw file chunks, or (b) a directory marker; if a node stores a directory marker, the name field of its links corresponds to the name of the target in the file system.

`inu` naively partitions each file into 256 KB chunks which it then stores in the raw variant of the discriminated union. This approach was chosen because it is also the one deemed appropriate for use in IPFS. However, more optimal chunking algorithms exist which offer an improved deduplication ratio for content, such as FastCDC [30]; these are left as an extension for the system.

### 4.4.1 Example

We illustrate the abstraction using an example; consider a directory which has the following structure and contents.

```
|-- test
|   |-- h.txt --> hello
|   |
|   |-- w.txt --> world
```

Figure 4.2: Example directory

Using the `inu` CLI, we're able to (1) import the directory into the store, (2) list the root CIDs for each directory item, and (3) export each file from the store. (CIDs in the following listings are shortened for clarity).

```
> inu add test
XFGZB test/h.txt
BAZRO test/w.txt
DTNCM test
```

Listing 4.6: Importing files using the `inu` CLI

When exporting a file, we can use either a file's CID or its path relative to another CID to refer to it.

```
> inu cat XFGZB
hello

> inu cat DTNCM/w.txt
world
```

Listing 4.7: Exporting files using the `inu` CLI

Below we approximate how we would encode this directory into a DAG.

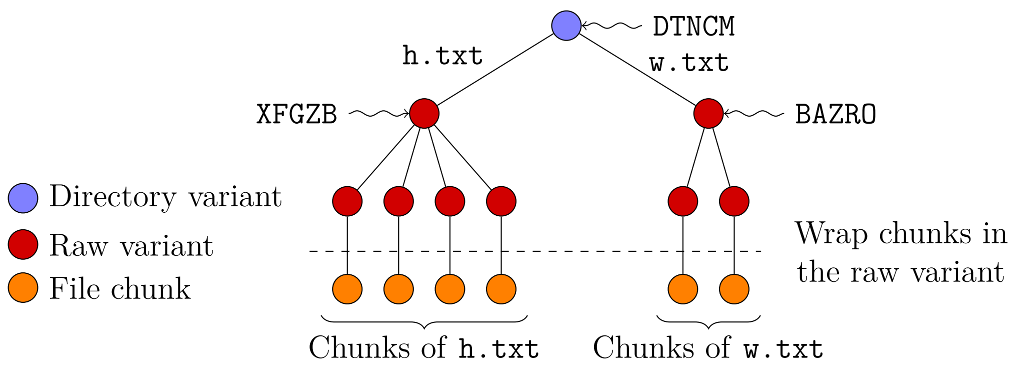

Figure 4.3: Merkle DAG for the example directory

## 4.5 Distributed Hash Table

The `inu` DHT is based on both the Kademlia and BitTorrent DHT specifications, and thus shares a similar overall description:

- Keys are opaque 256-bit quantities, i.e. the SHA256 hash of some data
- Participating computers each have a node ID in the 256-bit key space
- `(key,value)` pairs are stored on nodes with IDs ”close” to the key, (we define the notion of closeness in Section 4.5.3)
- A node-ID-based routing algorithm for efficient key-peer lookup lets anyone efficiently locate nodes near any given target key

### 4.5.1 Keys

Keys are 256-bit identifiers generated by the SHA256 hash function. SHA256 is suitable because it is able to generate a uniform distribution of keys, this is an important invariant for the correctness of the system [31].

```
type Key [32]byte
```

Listing 4.8: Definition of a key

Keys and DAG nodes must both be 256-bit quantities, otherwise DAG nodes could not be stored on the DHT ”as-is” without additional processing.

### 4.5.2 Routing

Nodes store contact information about each other to route messages.

```
type Contact struct {
  ID      Key
  Address string // IPv4 Address
}
```

Listing 4.9: Definition of a contact

Each node maintains a routing table of known nodes organised into buckets, where each bucket covers a portion of the address/key space. Buckets contain up to $k$ contacts, and are refreshed every $t_{Refresh}$[1] seconds. $k$ is chosen such that any given $k$ nodes are very unlikely to fail within $t_{Refresh}$ seconds of each other, thereby guaranteeing that each node contains redundant paths for each portion of the address space, ensuring routing resilience and fault-tolerance.

```
type Bucket struct {
    min, max Key
    contacts [K]Contact // Where K is a system parameter
}
```

Listing 4.10: Definition of a bucket

Buckets implement a least-recently seen eviction policy[2], except that live nodes are never removed from the list. By keeping the oldest live contacts around, buckets maximise the probability that the nodes they contain will remain online, further improving the resilience of the system. A second benefit of preferring old nodes is that they provide resistance to certain denial of service attacks: it is impossible to flush a node's routing state by flooding the system with new nodes, because new nodes will only be inserted into the bucket once old nodes leave the system [17].

The Kademlia spec defines the routing table as a binary trie [17], instead we take the BitTorrent approach which defines the routing table as a sorted list of buckets [12] – this approach is simpler to implement in a performant manner. Finally, routing tables are structured differently based on the source node we are routing from, (i.e. `self` in the below listing).

```
type RoutingTable struct {
    self    Contact
    buckets []Bucket
}
```

Listing 4.11: Definition of the routing table

[1]Refreshing is defined in Section 4.5.4.
[2]The exact policy is documented in Appendix **??**.

An empty routing table has one bucket which covers the whole address space, (i.e. $2^0$ to $2^{255}$). When inserting a node with ID $N$ into the table, it is placed within the bucket that has min $\leq N <$ max. After a bucket becomes full, (by storing $k$ nodes), no more nodes may be added unless our own node ID (`self`) falls within the range of the bucket, (in other words: unless our own node ID is "close" to the node ID we want to insert). In this case, the bucket is replaced with two new buckets, each with half the range of the old bucket, and the nodes from the old bucket are distributed among the two new ones.

This places a logarithmic bound, (specifically $O(\log_2(n))$ where $n$ is the size of the network), on the number of buckets which the routing table contains[3]. Consider that a network of 10 million nodes only requires 23 buckets: this has great implications for the scalability of the system, since the routing table design keeps its memory and computational overhead manageable, (i.e. it is trivial for most consumer hardware to operate a routing table of this size).

### 4.5.3 Distance

As specified previously, the design of the DHT requires some notion of "closeness", this can be formally defined using a metric space [32].

Formally, a metric space $(M, d)$ is composed of a set of elements $M$, which has a function $d : M \times M \to \mathbb{R}$ that is a *metric* on $M$, i.e. it defines a distance $\in \mathbb{R}$ between $M$'s elements.

Additionally, $d$ must satisfy the following axioms for all points $x, y, z \in M$:

1. $d(x, x) = 0$, the distance from a point to itself is zero
2. $x \neq y \to d(x, y) > 0$, the distance between two distinct points is always positive
3. $d(x, y) = d(y, x)$, the distance from $x$ to $y$ is always the same as the distance from $y$ to $x$
4. $d(x, z) \leq d(x, y) + d(y, z)$, the triangle inequality holds

In our case, the XOR function is a valid metric on our set $M$ (the key space), i.e. for two keys $x$ and $y$, their distance would be $d(x, y) = x \oplus y$.

Considering that the distance between two keys is frequently computed as part of the protocol, XOR was chosen because it's computationally inexpensive, and is often implemented in hardware, making it efficient even on older systems.

[3]The largest bucket being either $[0, 2^{254}]$ or $[2^{254}, 2^{255}]$ depending on `self`.

### 4.5.4 Protocol & Lookup

In this section, we first define the protocol by which DHT nodes communicate with each other, and then explain the procedure by which nodes look up information on the network.

The protocol consists of four primitive RPCs: we define each RPC, its route's URL and path parameters, and its request body if applicable.

**Ping** Ping a node to see if it's online.

```
GET /rpc/ping
```

**Store** Instruct a node to store peer information for later retrieval, note that it's possible to store multiple peers for a specific key. We post a JSON-encoded key-peer pair[4].

```
POST /rpc/store
{ "k": "ABCDE", // Key shortened for clarity
  "p": [{ "ip": "192.168.0.1",
          "port": 80,
          "asn": 65001,
          "published": "2024-03-31T14:23:02.556330267Z" }] }
```

**FindNode** The recipient returns the $k$ contacts it knows about which are closest to the target key.

```
GET /rpc/find-node/{key}
```

**FindPeers** If peers for the key exist in the recipient's store then they are returned. Otherwise, this RPC acts like *FindNode* and $k$ contacts are returned.

```
GET /rpc/find-peers/{key}
```

For each RPC, the recipient updates the bucket corresponding to the sender in its routing table, simultaneously on a successful reply, the sender updates the bucket of the recipient. To identify themselves, nodes attach the `Inu-Src` header which contains a JSON-encoded blob of their contact information.

```
Inu-Src: {
    "id":"ABCDE", // Key shortened for clarity
    "address":"10.41.0.2:3000"
    }
```

Listing 4.12: Example `Inu-Src` header

[4]We elaborate on the definition of peers in Section 4.5.6.

The node lookup process defines how we find up to the $k$ closest nodes to a given key. It is an iterative algorithm which is loosely parallel, meaning that the number of parallel requests in flight to other contacts is some low multiple of a value $\alpha$ [31].

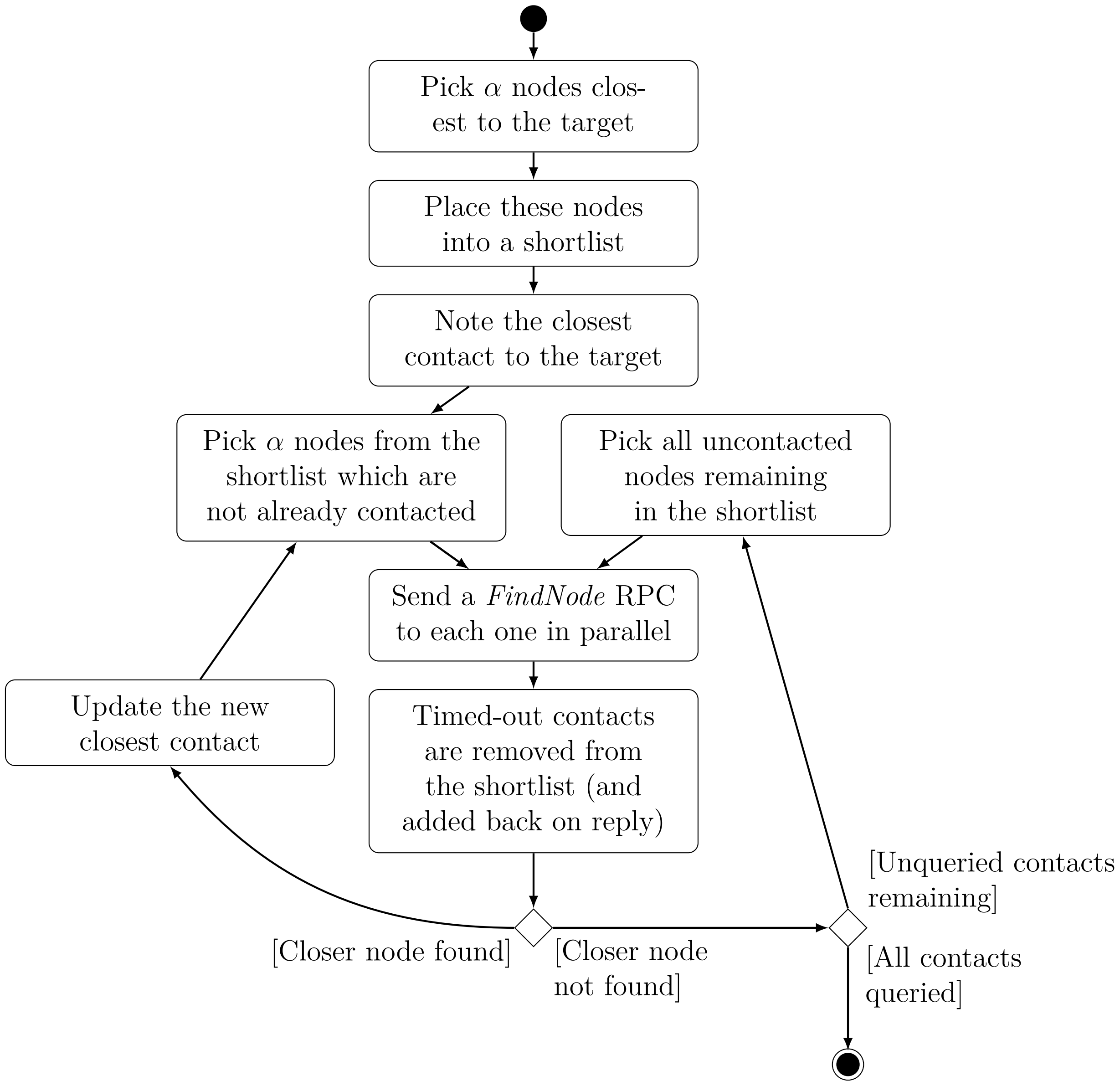
Pick $\alpha$ nodes closest to the target
Place these nodes into a shortlist
Note the closest contact to the target
Pick $\alpha$ nodes from the shortlist which are not already contacted
Pick all uncontacted nodes remaining in the shortlist
Send a *FindNode* RPC to each one in parallel
Update the new closest contact
Timed-out contacts are removed from the shortlist (and added back on reply)
[Unqueried contacts remaining]
[Closer node found]
[Closer node not found]
[All contacts queried]

Figure 4.4: Diagram of the node lookup algorithm

Using the lookup algorithm, we define how to store peer information across the network.

**Algorithm 1** Store a key-peer pair on the DHT

```
procedure STORE($k, p$)
    $cs$ ←LOOKUP($k$)
    for $c$ in $cs$ do
        SENDSTORERPC($c, k, p$)
    end for
end procedure
```

Finding peer information on the DHT requires a modified version of the lookup algorithm which:

- Sends $FindPeers$ instead of $FindNode$ RPCs
- Caches peer information at the node closest to our key which did not return the information, this improves the fault-tolerance of the system since popular data is made increasingly redundant

**Algorithm 2** Find the peers providing a given key

```
function FINDPEERS($k$)
    return LOOKUP'($k$)                ▷ LOOKUP' is the modified algorithm
end function
```

Nodes are contacted in parallel which minimises the latency for the whole lookup procedure. Additionally, lookups have a logarithmic bound on the number of hops they must perform, specifically $O(\log_2(n))$, which further improves the performance and scalability of the system. We are able to quickly "home in" on our target node, since each node we query has an increasingly better view of our target's address space. Consider that a network of 1 million nodes only requires 20 hops at most to find any key-peer pair.

Every $t_{Refresh}$, if no lookups have been performed for any node in a given bucket's range, then we select a random node from said bucket and perform a lookup on it, and then insert the returned nodes into the bucket. This ensures that the routing table remains up to date and accurately reflects the DHT's current topology, facilitating efficient routing.

### 4.5.5 Bootstrapping

Joining an already-existing DHT network consists of three steps:

1. Insert the contact for some node known to be in the DHT into our routing table
2. Perform a lookup on ourself and insert the found nodes into the table
3. Refresh each bucket in the routing table

This procedure ensures that (1) we receive knowledge of our nearby address space, and (2) other nodes in the DHT are aware of our existence.

### 4.5.6 Lookup Service API

We explain the theory by which `inu` can reduce network congestion, and elaborate on how it integrates into the lookup service API which `inu` offers.

The core idea behind `inu` is to reduce traffic which traverses peering exchange boundaries by encouraging traffic to stay within a singular autonomous system (AS), this has an overall effect of reducing network load (and thus congestion). To do this: when a peer performs a `FindPeers` request, the lookup service replies with peers which are part of the same AS as the requester, (unless none exist in which case any arbitrary set of peers is returned). Once the requesting peer begins communicating with its nearby peers, we rely on existing best-effort routing infrastructure to limit all packets sent to within the AS's boundaries.

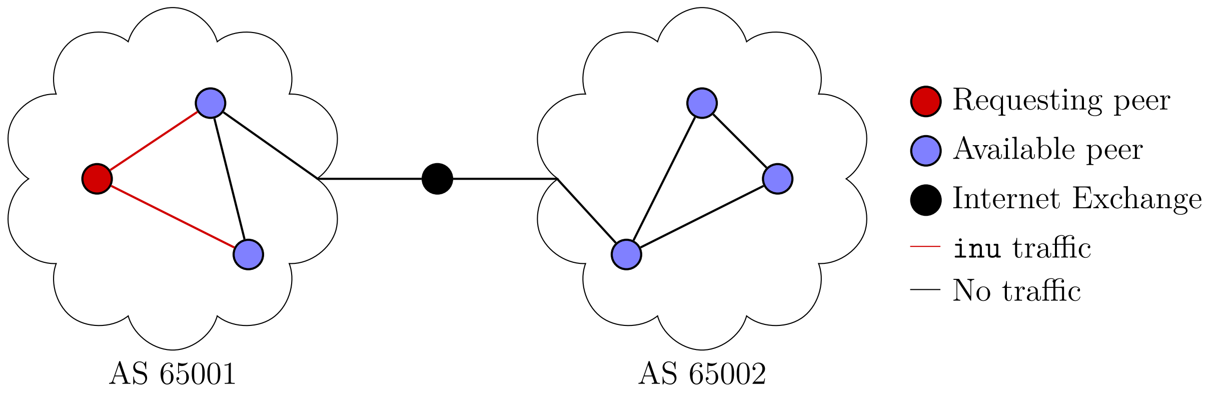

Figure 4.5: Illustration of traffic remaining within an AS

To accomplish this, when a peer uses the `Notify` API endpoint to mark itself as owning a DAG node, we also store the AS number (ASN) of its IP and the time at which it was added to the DHT[5], before propagating the peer information around the network.

[5]Explained in Section 4.5.7.

```
type Peer struct {
  IP        netip.Addr
  Port      uint16
  ASN       int
  Published time.Time
}
```

Listing 4.13: Definition of a peer

To determine the ASN of a particular IP, we use data [33] provided by regional Internet address registries (RIRs) to populate a 32-bit binary trie[6] of CIDR range to ASN mappings. Tries run the the longest prefix match algorithm [34] in $O(32)$, which makes them suitably fast to identify thousands of IPs per second. They are also able to scale in terms of memory usage, which is especially important in our case where there exist approximately 950,000 CIDR to ASN mappings for the global routing table.

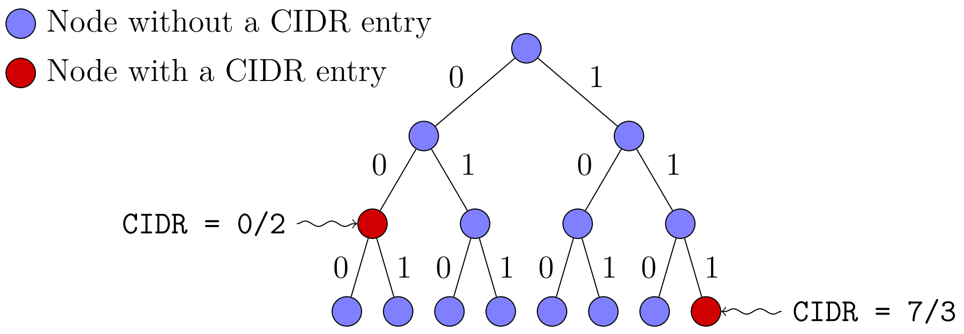

Figure 4.6: Example of a 3-bit binary trie with two CIDR entries

For subsequent `FindPeers` API requests, the DHT node attempts to match the ASN of the requesting IP with peer ASNs stored on the DHT.

We implement the lookup service as a REST API with two routes.

**FindPeers** Request the nearby peers for a given key.

```
GET /key/{key}
```

**Notify** Notify the lookup service that we are providing a given key. We post the JSON-encoded port number we are listening on.

```
PUT /key/{key}
"2000"
```

[6]IPv4 addresses are 32 bits.

As mentioned previously in Section 3.4, the DHT must be resistant to hijacking by malicious actors who would intend to use it to propagate illicit content. To facilitate this, peers may provide a secret key to the *Notify* endpoint by attaching the `Inu-Upload` header which contains the Base32 encoding of the key. If the key is valid then the DHT node accepts any key which the peer is attempting to notify for, otherwise, the DHT must perform a *FindPeers* request on the network to validate that the key already exists before accepting the request.

```
Inu-Upload: ABCDE // Key shortened for clarity
```

Listing 4.14: Example `Inu-Upload` header

### 4.5.7 Replication and Republishing

To avoid data loss when nodes leave the network, each node republishes every key-peer pair that it contains every $t_{Replicate}$ seconds; this involves performing a *Store* operation on the key-peer pair as defined in Algorithm 1. If more than $t_{Expiry}$ seconds have passed since the key-peer pair was inserted into the network, (as defined by the `Published` field in the `Peer` struct), the pair is removed from the node's store; this removes dead information from the DHT in addition to reducing its total load, thereby increasing its performance.

The onus of keeping a key-peer pair alive is on its original publisher: the peer; the pair should be republished every $t_{Republish}$ seconds.

### 4.5.8 Parameters

We summarise the parameters which affect the DHT including their default values (which were taken from the Kademlia spec).

Table 4.1: DHT Parameters

| Name | Description | Default |
|---|---|---|
| $k$ | Bucket size | 20 |
| $\alpha$ | Degree of parallelism | 3 |
| $t_{Refresh}$ | Refresh interval | 1 hour |
| $t_{Replicate}$ | Replication interval | 1 hour |
| $t_{Expiry}$ | Expiry timeout | 1 day |
| $t_{Republish}$ | Republish interval | 1 day |

## 4.6 Peers

Each `inu` peer provides one simple HTTP endpoint to comply with the peer API specification. However, instead of providing nodes, our implementation provides blocks which must then be deserialised into DAG nodes.

**Block** Request a block with the given CID from the peer.

```
GET /block/{cid}
```

This is because, in its serialised form, a block can be instantly checked for integrity; whereas checking a node for integrity requires serialising the node and (possibly) handling the effect of a failed serialisation which is slower.

```
func (b *Block) Valid() bool {
  return cid.New(b.Data) == b.CID
}
```

Listing 4.15: Checking block validity

## 4.7 Discussion

We discuss the challenges and considerations taken when implementing `inu`, in to using the Go programming language.

**Vague Specifications** Both the IPFS white paper and the Kademlia specification are somewhat vague; Kademlia's specification predominantly consists of descriptions of its key functions and behaviours, whilst offering minimal insight into the rationale behind its design decisions. Additionally, Kademlia contains no reference implementation, whereas the IPFS implementation is extremely convoluted, making it difficult to decipher its core functionality and adapt it towards our implementation. This presented a significant challenge towards understanding the ”why” and ”how” for these two systems, which was crucial in order to implement `inu` in a correct and efficient manner.

**Inherent Complexity** A major challenge was the non-linear relationship between the complexity of the textual description of the system and the actual code implemented. A goal of the project was to create a high quality implementation which was both maintainable and performant, particularly in relation to the concurrent components of the system such as the file downloading procedure. Go remains an excellent choice in this regard because, (1) it has an extensive collection of concurrency primitives which facilitate clear modelling of business logic, and (2) its syntax is especially simple and contains no ”bells and whistles”, making it easy to parse for newcomers.

**Testing** The ambiguity within both specifications necessitated an extensive testing methodology, comprising of unit, integration, and end-to-end tests, in order to validate the correctness of the system (and adherence to the original Kademlia spec when implementing the DHT). This led to a long testing process which took more time than anticipated, (there are approximately 2500 lines of testing code written). An advantage of using Go was that, (1) the standard library has first class support for testing HTTP APIs through the use of the `httptest` library, and (2), Go contains a built-in race detector, both of these massively increased development velocity proving invaluable during development.

**Readability** A lot of care was taken into ensuring the clarity and comprehensibility of the final implementation, this resulted in extensive effort to write (over 1000 lines of) self-documenting code which outline the operation and rationale behind the system.

**Performance** During analysis of earlier iterations of the implementation, notable instances of unexpectedly slow behaviour were observed. To solve this we incorporated observability mechanisms, notably: logging, profiling and tracing, which aided in identifying bottlenecks. The fact that these mechanisms are built into the Go toolchain was especially helpful since profiling and tracing were both novel concepts which demanded additional time to grasp.

An example of a particularly slow piece of code was the HTTP handler for the DHT's *Notify* endpoint. The function can be split into two parts: (1) store the peer information in the node's local store, and (2) store the peer information on the network. Originally, the handler only returned after the storing the peer on the network, however, using profiling tools we were able to identify that this became prohibitively slow as the size of the network increased, (this was due to the lookup operation). Instead, the handler returns straight after it stores the peer in its local store, whilst scheduling a thread which stores the peer on the network. This represents an acceptable compromise between increasing the performance of the system whilst weakening its consistency model.

# 5 Evaluation

In this section we explain the approach used to measure the performance of our implementation, including the environmental setup. We then analyse the results and draw conclusions regarding the system.

## 5.1 Environmental Setup

Previous approaches to testing DHTs have either used specialised hardware to virtualise networks [35] or entirely foregone simulating the network itself [36]; since we did not have access to custom hardware, we took an involved approach which simulates the Linux networking stack on consumer hardware, specifically an AMD Ryzen 5 3600X CPU at 3.6 GHz and 32 GB of RAM at 3200 MHz.

We decided to use the `Gont` library [37] which enabled us to virtualise network topologies from within Go. The principle behind `Gont` is the concept of Linux network namespaces: these partition kernel resources related to networking from a process-perspective. This means that processes can be isolated from each other or grouped together to have a shared independent view of the networking stack. `Gont` is particularly useful because it is both programmable and lightweight, making it more ergonomic than containerised methods to virtualise networks, in addition to incurring less memory and computational overhead.

In order to virtualise `inu`, we defined (1) a CLI which lets a process participate in the network as either a peer or a DHT node, and (2) a controller dubbed `inuctl` which communicates with peers using an RPC API to simulate specific scenarios to evaluate. Finally, we used the `netem` [38] Linux module to simulate latency, jitter, packet loss, and rate limiting for egress links between peers in order to represent real-life network conditions.

## 5.2 Scenarios

In this section, we describe the scenarios which we will evaluate `inu` under, to see if it has met the non-functional requirements defined in Section 3.1.

### 5.2.1 Churn

One of the most popular evaluation scenarios in past literature for DHT-based networks, is their performance under churn, where churn is defined as ”the phenomenon of the continuous arrival and departure of participating peers of a P2P overlay” [35]. Churn is a significant concern because it increases the volatility of the global routing table, in addition to possibly eliminating whole sections of the DHT. Churn resistance is indicative of the system’s fault tolerance.

To model churn, past approaches [36] used the exponential distribution to generate session times for the mean uptime and downtime of a node; the higher the uptime the lower the churn and vice versa. We chose to follow this approach.

Churn resistance is measured using a metric called ”success rate”, this is the proportion of `FindPeers` requests in a given time period that return the same peer that was originally set for a given key. For our test, we evaluated how the rate changes as the parameters of the DHT are changed, namely $k$ and $\alpha$, in addition to the length of the mean uptime time for all nodes.

`inuctl` was used to create a churn scenario where half of all nodes were live in the network at any given time: it operated network of size $N + 1$, (where the last node is a bootstrap node that always stays online), with nodes that churn for mean session time $T$, as following:

1. Bring a dedicated bootstrap server online that stays online for the whole duration of the scenario
2. Bring half of the nodes online and bootstrap them to the network at a rate of 20 nodes per second
3. Store $N$ randomly generated keys on the network at a rate of 50 keys per second
4. Wait 15 seconds for the DHT to stabilise, (i.e. distribute keys around the network through RPCs)
5. Begin churning the nodes, this is where nodes which are currently online stay online for the next generated $t$, after which they crash and stay offline for the next generated $t$, and subsequently come online again. This happens continuously until the scenario has finished. Offline nodes operate in the same manner but in reverse, (i.e. stay offline, then come online). Nodes use the same IP address and ID when rejoining the network.
6. Until 2 hours have passed, request a randomly chosen key 20 times per second, and track the rolling success rate of all requests

### 5.2.2 Download Speed

This scenario attempts to verify that increasing the number of peers in the network decreases the time taken to download a file since (theoretically) the total upload bandwidth increases, this is indicative of how scalable the system is. `netem` is used to limit the upload speed of all peers to be similar to the average Wi-Fi connection speed.

We vary both the number of peers in the network $P$ and the size of the shared file $F$ in the following scenario:

1. Bootstrap a lookup service composed of 2 DHT nodes
2. The first peer notifies itself as a provider for a randomly generated file of size $F$ to the lookup service
3. Instruct $P - 1$ peers to download the file and wait for them to finish
4. Apply the `netem` configuration to all peers in the network
5. Instruct the final peer to download the file and measure the time taken

### 5.2.3 Congestion-Avoidant Routing

This scenario aims to evaluate the amount by which `inu` is able to reduce load flowing through Internet exchanges by providing peers with informed routing decisions. We are able to measure the total load of communication between peers through packet-capture utilities which `Gont` provides.

We vary the size of the shared file $F$ in the following scenario:

1. Bootstrap a lookup service composed of 2 DHT nodes
2. Consider AS 1 with two nodes $A$ $B$ and AS 2 with one node $C$, both AS's are connected via a bridge interface
3. Peer $C$ notifies itself as a provider for a randomly generated file of size $F$ to the lookup service
4. Instruct peer $A$ to download the file and measure the total load for packets sent between $A$ and $C$
5. Instruct peer $B$ to download the file and measure the total load for packets sent between $B$ and $C$

For the second download, we expect to see a drastically reduced load compared to the first one, since most requests should be towards peer $A$ because it's part of the same AS as peer $B$.

## 5.3 Results & Conclusions

In this section we present our results for each scenario and draw conclusions in relation to the system.

### 5.3.1 Churn

We analysed how varying $k$ and $\alpha$ affects the performance of the system.

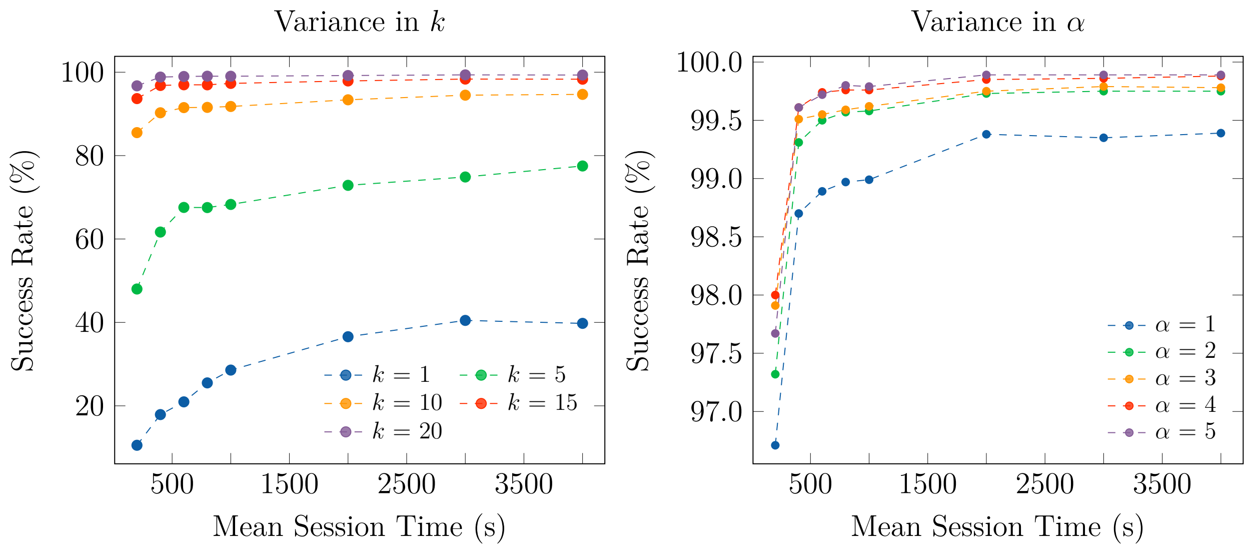

Figure 5.1: Effects of $k$ and $\alpha$ on success rate

***k*-Variance** To observe the effect of increasing the bucket size on the success rate we fixed the parallelism degree at $\alpha = 1$ so that it wouldn't influence the evaluation. We determined that as $k$ increases, the system becomes more fault tolerant since the success rate increases, however, after about $k = 15$ there is little difference in the increased success rate. The pathological case for the system is $k = 1$, where the success ratio reaches as low as 10%. This is due to the way in which the lookup operation works: consider that every node is aware of the bootstrap when initiating lookup, but the bootstrap node is purged from the shortlist after the first round of lookups due to the minimal bucket size. When churn levels are high it's unlikely for the node in the next lookup round to be online, meaning that the lookup almost always fails since the shortlist will always be empty.

***α*-Variance** To observe the effect of increasing the parallelism degree of lookup operations on the success rate we fixed the bucket size at $k = 20$, (the default value), so that it wouldn't influence the evaluation. It's clear that as the degree increases, the success rate also increases; however, after

$\alpha = 3$, there is only negligible increases in the success rate. Indeed, previous research has concluded that $\alpha = 3$ is the optimal parallelism degree [39].

Increasing both $k$ and $\alpha$ improves the system's fault tolerance. To ensure that the lookup service remains robust, redundant, and stable we select $k = 5$ and $\alpha = 3$, which is able to achieve success rates of 98% even at high levels of churn. Our conclusion is that the system is fault-tolerant.

### 5.3.2 Download Speed

Observing Figure 5.3, we identify the key takeaway that the download speed for a file increases proportionally to number of peers in the network. This effectively sets no bounds on the number of peers which can participate in the `inu` overlay, (apart from the capacity of existing networking infrastructure), meaning the system is (1) highly scalable due to the number of peers it supports, and (2) fault-tolerant since information is made redundant across a large number of peers.

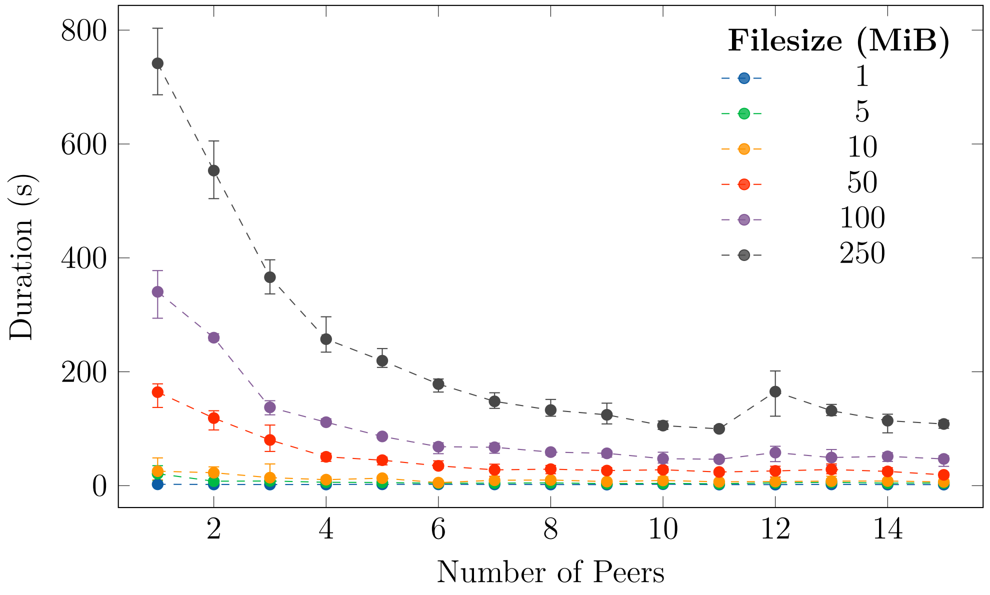

Figure 5.2: Download Duration for Final Peer on Wi-Fi

There are two more observations which are elaborated upon in Section 6.2:

- We observe that for larger file sizes, the benefit of having more peers becomes more apparent: for the largest file size $F = 250$ MiB, the system is able to quarter the time taken to download the file. For each file size $F$, there exists a number of peers after which the download duration plateaus; we believe this is due to the naivety of the downloading mechanism which randomly selects peers to download from, and is perhaps not utilising the available peers as efficiently as possible.

- A point of interest was the fact that the download speed was much slower than the speed which we capped the egress links at (5 MB per second). After further investigation we determined that this was not a problem with the virtualisation technology, but instead it was due to inflation of the amount of data sent over the wire. This was caused by two factors: (1) the additional size of HTTPS and TCP headers, and (2) the encoding a file into a DAG; in the worst case, a 250 MiB file was represented as a 444 MiB DAG. This is especially egregious because the file was generated completely randomly, meaning no deduplication could take place.

### 5.3.3 Congestion-Avoidant Routing

Observing this scenario, we see that the presence of a peer in the same AS reduced the load flowing through the IXP by 85%, showing that the system is able to effectively mitigate IXP load, this ratio seems to be consistent across the multiple file sizes which we evaluated.

After further investigation we noted that the reason the load for the second download is not completely eliminated, is due to the weakly consistent model of the system, where the nodes in the lookup service do not have a totally consistent view of which peers provide which key. As an implementation detail, peers are able to query specific DHT nodes in the lookup service; we performed an additional evaluation where all peers queried the same node, so that they would all have a totally consistent source of truth. In that situation, all load through the IXP was completely eliminated. We discuss this further in Section 6.2.

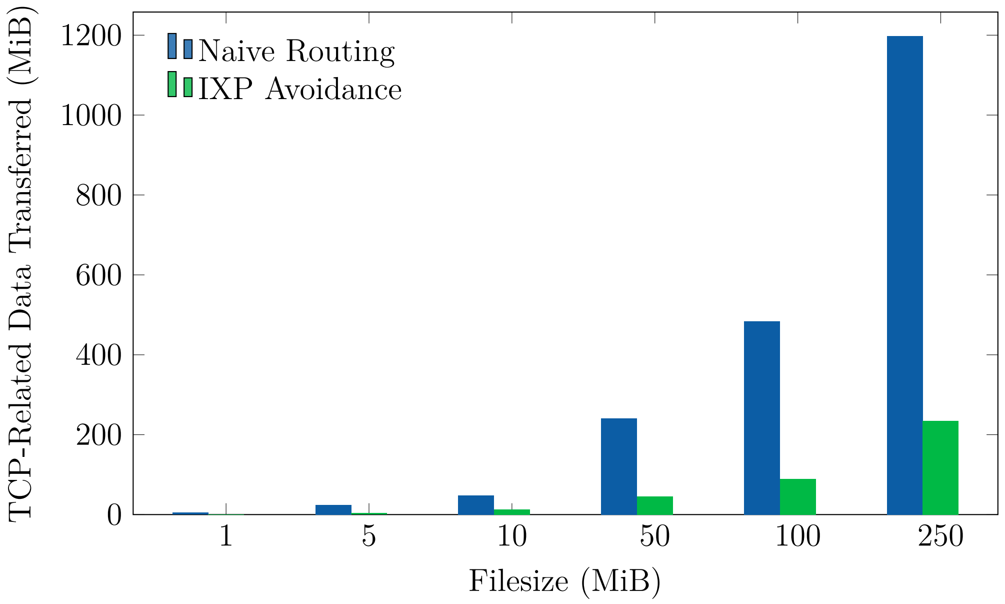

Figure 5.3: Ability Of DHT To Reduce Load At Different File Sizes

# 6 Contributions & Future Directions

## 6.1 Contributions

This study attempts to address the gaps present in past filesharing systems by synthesising their design in addition to introducing a novel routing strategy, and in doing so, we make important contributions:

- We extend past filesharing system designs, (specifically BitTorrent), in a way which makes them non-reliant on external metadata
- We introduce a novel routing strategy which takes advantage of existing best-effort routing infrastructure to successfully reduce network congestion; there is no need for additional specialised software or hardware to be deployed
- The filesharing process is able to deduplicate across all files shared, this makes it especially useful for sharing different collections of data which vary slightly and may have different versions installed simultaneously, e.g. consider training datasets, video game assets, or software packages which vary slightly by version (i.e. v0.8, v1.1, v1.2.5)
- Our system provides granular control over the filesharing process in addition to the type of data which can be shared whilst remaining simple to reason about and implement compared to past research, (specifically IPFS)

## 6.2 Future Directions

As the system currently stands it's inefficient, and not fit for release in all production environments. In this section we discuss future directions which would make the system suitable for deployment in either (1) sub-optimal networks which suffer from harsher conditions, or (2) in services which have harsher quality of service constraints.

### 6.2.1 Reducing Network Load

We discuss multiple techniques to reduce the throughput needed to communicate over the network.

**WebTransport** A current limitation of the lookup service is the over-utilisation of network bandwidth to send DHT RPCs which are mainly dynamic in nature, (i.e. messages which will not benefit from some sort of caching). This could be alleviated through the use of a stateful protocols which are often much leaner than stateless ones. We recommend WebTransport[1] (WT) [40] in comparison to a classic secure reliable transport such as TCP/TLS for the following reasons: (1) streams can be multiplexed over a single connection, (2) improved congestion control versus TCP, and (3) WT performs more efficient optimisation for reducing latency.

**HTTP/3** HTTP/3 is the next iteration of the HTTP protocol. Even though it's standardised [41], there does not yet exist a working implementation in Go. Once implemented, one of the main benefits of switching to HTTP/3 is the improved performance of fetching multiple blocks of data simultaneously, including better compression ratios for data sent over the wire.

**Encoding** A more efficient scheme such as CBOR [42] could be used to serialise data; this includes serialising network communication in addition to serialising the representation of nodes in the DAG store. This should be trivial to implement since many different Go libraries use the same encoding/decoding API, meaning that switching encoding libraries should be a "drop-in replacement".

**Chunking** As mentioned previously, using a content chunking algorithm which optimises for deduplication such as FastCDC [30] would improve the ability of the DAG store to deduplicate data across all files shared. This would improve performance for systems which have tighter memory constraints.

### 6.2.2 Real-Life Testing in Internet Exchanges

It remains to be seen whether `inu` is able to scale up and reduce congestion across production networks. This should be studied further to confirm the efficacy of the system.

[1]WebTransport still only exists in draft format, even though it is nearing full standardisation.

### 6.2.3 Improving Download Speed

If the system used a more involved scheme to decide which peers to request blocks of data improve, the network utilisation would be higher and download speeds would improve. This is important for networks which suffer harsh network conditions, e.g. it might be prudent to categorise nodes as good or bad based on the latency taken in order to avoid links which suffer from packet loss and high latencies.

### 6.2.4 Improving Congestion-Avoidant Routing

As described in Section 5.3.3, the ability to reduce congestion through Internet exchanges is proportional to the consistency of the data stored in the lookup service.

As outlined by the CAP theorem [14], having a system which is (almost totally) consistent, highly available, and partition tolerant is impossible. However, we still envisage that using a highly-available key-value store, which is more consistent than a standard DHT, will provide improved congestion-avoidance performance at the cost of being more complex to engineer and implement.

# 7 Summary & Reflections

This section covers a discussion of the project's management, reflections on the final outcome of the project, and a discussion of Law, Social, Ethical and Political issues, from the perspective of the author.

## 7.1 Project Management

A Gantt chart of the work is presented in Appendix ??. In the first semester, the focus was on creating the building blocks of the `inu` peer and beginning to create the lookup service, whereas the focus in the second semester was finishing the development of the lookup service, integrating it with the peers, and then evaluating the system's performance.

For the majority of the dissertation up until early February I was ahead of schedule by about 2 to 3 weeks. The biggest time sink however, was the system evaluation which took 8 weeks instead of the scheduled 2. This was due to two reasons: first of all, as mentioned in my interim report, I had less time to work on tasks due to a heavier module weighting for the second semester. Secondly, I had to undertake a "deep-dive" (relative to my experience) into the Linux networking stack, I learnt that there exists a significant difference between simply containerising an application and creating a programmable form of network virtualisation which supports custom topologies and link impairments. Before finding `Gont` I iterated on over 3 different approaches to virtualising the network which were all unsuccessful. Even though this process taught me a lot, (which I am grateful for), it was complicated and frustrating: I encountered many novel tasks which had little documentation such as (1) recompiling the kernel to enable specific networking modules, or (2) make low-level Linux API calls to programmatically apply `netem` configurations on specific network interfaces.

To handle this I removed the advanced content chunking task from the schedule. Even though I did not implement all features the project was ahead of schedule for most of development and I managed to account for the sudden duration increase of the evaluation whilst fulfilling every single requirement, so overall I would say that the project was well managed.

## 7.2 Law, Social, Ethical and Political Issues

**Intellectual Property** This work has generated source code for the design of a filesharing system which counts as my own Intellectual Property. I plan to publish this work so that other people are able to learn from it, to this end it will be released under some open-source license after completion.

**Social Impact** This system will reduce the load which networks undergo making them more reliable at a very low cost, this in turn improves the quality of life that users have when downloading content media over the network. This has a positive social benefit considering the fact that these networks underpin the majority of digital infrastructure which people use in their day to day lives. Overall, this project connect to to UN Development Goal 9: ”Build resilient infrastructure”.

**Legal and Ethical Issues** As mentioned in Section 3.4, the system is designed to prevent malicious actors from participating in the network and propagating illicit content. Without this prevention mechanism, there is: (1) a legal issue where network operators would be accountable for the illicit content shared, and (2) it would be unethical of them to operate a service which could be used to distribute this content. Even so, it is still possible for users to self-host their own version of the system to distribute illegal or unethical content; however, this is also possible with a multitude of other pre-existing technologies [43]. Since we are not offering a new or easier way to distribute this content, we consider the ethical risk of this project mitigated.

## 7.3 Reflections

Overall I think the project went well and met all its requirements, this is validated by the comprehensive test suite and system evaluation. I am surprised by how well the novel routing strategy works given its simplicity. Even though I did not manage to implement all features, all the requirements were still fulfilled, I believe the scale of my project and the rigour with which I evaluated it is significant in itself.

The primary regret with the system is the fact that I didn’t get to implement the aspects that now form part of the future directions which I would view as ”low-hanging fruit”, this was due to the time taken to evaluate the system.

Midway through the project, (around early January), I reassessed its direction; originally it was supposed to be a distributed file system, (hence the initial interest in IPFS), however, I scaled it back to just a filesharing system. When I reconfigured the design and architecture of the project, I still kept the IPFS-inspired components, (namely the DAG store), because I believed they would still be beneficial for the system. Unfortunately, I believe I was too focused on the theoretical aspect of the system, especially the deduplication benefit of the store, instead of considering the practical aspects of the end-user experience. The experience is inconvenient because files must always be exported from the store to be used, and, as soon as they're exported, we lose all deduplication benefits associated with storing the file. I've learnt that in the future I need to be more mindful of the practical application of my project, instead of being blinded purely by its theoretical benefits.

Finally, a great source of pride is the fact that the project reflects the progress I've made in the past three years at University. Three years ago, (on a whim), I attempted to implement the Kademlia specification as an idea for my next programming project, however, this endeavour completely failed: I could not parse the subject manner and I had no clue where to even begin. In contrast to this, with my dissertation I've managed to gain an intimate understanding of Kademlia, in addition to expanding upon it, and integrating it with multiple moving parts to create a fully-fledged distributed system.

# 8 Bibliography

[1] J. Denton. *Learning OpenStack Networking (Neutron)*. Packt Publishing, 2015. ISBN: 9781785280795. URL: `https://books.google.co.uk/books?id=cfWoCwAAQBAJ`.

[2] Mark Jackson. "Fortnite Drives Broadband Traffic Record at UK ISP Virgin Media". In: (Nov. 6, 2023). URL: `https://www.ispreview.co.uk/index.php/2023/11/fortnite-drives-broadband-traffic-record-at-uk-isp-virgin-media.html` (visited on 12/04/2023).

[3] Melanie Mingas. "AMS-IX, LINX record new traffic peaks". In: (Apr. 12, 2021). URL: `https://www.capacitymedia.com/article/29otd15ias3ir9egkxs01/news/ams-ix-linx-record-new-traffic-peaks` (visited on 12/04/2023).

[4] Chris Keall. *NZ broadband use spikes to all-time high - All Blacks v Fortnite*. 2023. URL: `https://www.nzherald.co.nz/business/nz-broadband-use-spikes-to-all-time-high-all-blacks-vs-fortnite/VTIOTYY4VBBBHPMPG674KQJTXA/` (visited on 12/05/2023).

[5] V. Cardellini, M. Colajanni, and P.S. Yu. "Dynamic load balancing on Web-server systems". In: *IEEE Internet Computing* 3.3 (1999), pp. 28–39. DOI: `10.1109/4236.769420`.

[6] Robin Jan Maly et al. "Comparison of centralized (client-server) and decentralized (peer-to-peer) networking". In: *Semester thesis, ETH Zurich, Zurich, Switzerland* (2003), pp. 1–12.

[7] G. Somasundaram and A. Shrivastava. *Information Storage and Management: Storing, Managing, and Protecting Digital Information*. EMC education services. Wiley, 2009. ISBN: 9780470294215. URL: `https://books.google.co.uk/books?id=UcsX9kvYD-gC`.

[8] *The BitTorrent Protocol Specification*. 3. Bittorrent.org. Jan. 2008.

[9] Bittorrent.com. *BitTorrent and Torrent Software Surpass 150 Million User Milestone*. 2014. URL: `https://web.archive.org/web/20140326102305/http://www.bittorrent.com/intl/es/company/about/ces_2012_150m_users` (visited on 03/26/2014).

[10] Arnaud Legout, Guillaume Urvoy-Keller, and Pietro Michiardi. "Understanding bittorrent: An experimental perspective". In: (2005).

[11] G. Neglia et al. "Availability in BitTorrent Systems". In: *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*. 2007, pp. 2216–2224. DOI: `10.1109/INFCOM.2007.256`.

[12] *DHT Protocol*. 5. Bittorrent.org. Jan. 2008.

[13] Liang Wang and Jussi Kangasharju. "Measuring large-scale distributed systems: Case of BitTorrent Mainline DHT". In: Sept. 2013, pp. 1–10. DOI: `10.1109/P2P.2013.6688697`.

[14] Dennis Trautwein et al. “Design and evaluation of IPFS: a storage layer for the decentralized web”. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM ’22. ACM, Aug. 2022. DOI: 10.1145/3544216.3544232. URL: http://dx.doi.org/10.1145/3544216.3544232.

[15] Einar Mykletun. “Providing Authentication and Integrity in Outsourced Databases using Merkle Hash Tree ’ s”. In: URL: https://api.semanticscholar.org/CorpusID:10853829.

[16] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. 2nd. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016. ISBN: 153028175X.

[17] Petar Maymounkov and David Mazieres. “Kademlia: A Peer-to-peer Information System Based on the XOR Metric”. In: *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS ’02)*. Ed. by Frans Kaashoek and Antony Rowstron. Mar. 2002.

[18] Juan Benet. “IPFS - Content Addressed, Versioned, P2P File System”. In: *CoRR* abs/1407.3561 (2014). arXiv: 1407.3561. URL: http://arxiv.org/abs/1407.3561.

[19] Kubo Authors. *Kubo*. Version 0.24.0. Dec. 2023. URL: https://github.com/ipfs/kubo.

[20] Boxo Authors. *Boxo*. Version 0.18.0. Mar. 2024. URL: https://github.com/ipfs/boxo.

[21] Giuseppe DeCandia et al. “Dynamo: amazon’s highly available key-value store”. In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP ’07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, pp. 205–220. ISBN: 9781595935915. DOI: 10.1145/1294261.1294281. URL: https://doi.org/10.1145/1294261.1294281.

[22] I2P. *The Network Database - I2P*. 2023. URL: https://geti2p.net/en/docs/how/network-database#threat (visited on 12/12/2023).

[23] Can I Use? *HTTP/2*. 2023. URL: https://caniuse.com/?search=HTTP%2F2 (visited on 12/18/2023).

[24] HTTP Working Group. *Implementations*. 2023. URL: https://github.com/httpwg/http2-spec/wiki/Implementations (visited on 12/18/2023).

[25] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. RFC 8445. July 2018. DOI: 10.17487/RFC8445. URL: https://www.rfc-editor.org/info/rfc8445.

[26] Brian Campbell et al. *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*. RFC 8705. Feb. 2020. DOI: 10.17487/RFC8705. URL: https://www.rfc-editor.org/info/rfc8705.

[27] nictuku. *dht*. Version fd1c1dd3d66a75ef5706bb9eb47782afd94cc6f4. Apr. 2024. URL: https://github.com/nictuku/dht.

[28] Arul Lawrence Selvakumar and C. Suresh Ganadhas. “The Evaluation Report of SHA-256 Crypt Analysis Hash Function”. In: *2009 International Conference on Communication Software and Networks*. 2009, pp. 588–592. DOI: 10.1109/ICCSN.2009.50.

[29] SQLite. *Appropriate Uses for SQLite*. 2022. URL: https://www.sqlite.org/whentouse.html (visited on 12/12/2023).

[30] Wen Xia et al. "FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 101–114. ISBN: 978-1-931971-30-0. URL: `https://www.usenix.org/conference/atc16/technical-sessions/presentation/xia`.

[31] *Kademlia: A Design Specification*. XLattice. Feb. 2010.

[32] D. Burago, I.U.D. Burago, and S. Ivanov. *A Course in Metric Geometry*. Crm Proceedings & Lecture Notes. American Mathematical Society, 2001. ISBN: 9780821821299. URL: `https://books.google.co.uk/books?id=dRmIAwAAQBAJ`.

[33] APNIC. *data-raw-table*. 2024. URL: `https://thyme.apnic.net/current/data-raw-table` (visited on 03/31/2024).

[34] D. Comer. *Computer Networks and Internets*. Pearson/Prentice Hall, 2009. ISBN: 9780136061274. URL: `https://books.google.co.uk/books?id=tm-evHmOs3oC`.

[35] Zhonghong Ou et al. "Performance evaluation of a Kademlia-based communication-oriented P2P system under churn". In: *Computer Networks* 54.5 (2010), pp. 689–705. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.comnet.2009.09.022`. URL: `https://www.sciencedirect.com/science/article/pii/S1389128609002990`.

[36] Michaela Murray, Isaac Westlund, and Guy Wuollet. "Reproducing and Performance Testing Kademlia". In: ().

[37] Steffen Vogel. *Gont*. Version 2.0. Apr. 2024. URL: `https://github.com/cunicu/gont`.

[38] Stephen Hemminger et al. "Network emulation with NetEm". In: *Linux conf au*. Vol. 5. 2005, p. 2005.

[39] D. Stutzbach and R. Rejaie. "Improving Lookup Performance Over a Widely-Deployed DHT". In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. 2006, pp. 1–12. DOI: `10.1109/INFOCOM.2006.329`.

[40] Victor Vasiliev. *The WebTransport Protocol Framework*. Internet-Draft draft-ietf-webtrans-overview-07. Work in Progress. Internet Engineering Task Force, Mar. 2024. 12 pp. URL: `https://datatracker.ietf.org/doc/draft-ietf-webtrans-overview/07/`.

[41] Mike Bishop. *HTTP/3*. RFC 9114. June 2022. DOI: `10.17487/RFC9114`. URL: `https://www.rfc-editor.org/info/rfc9114`.

[42] Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 8949. Dec. 2020. DOI: `10.17487/RFC8949`. URL: `https://www.rfc-editor.org/info/rfc8949`.

[43] Paul A. Watters, Robert Layton, and Richard Dazeley. "How much material on BitTorrent is infringing content? A case study". In: *Information Security Technical Report* 16.2 (2011). Social Networking Threats, pp. 79–87. ISSN: 1363-4127. DOI: `https://doi.org/10.1016/j.istr.2011.10.001`. URL: `https://www.sciencedirect.com/science/article/pii/S1363412711000616`.