18-842 Final Project Design Document
The InterPlanetary File System

Chen, Minghan
minghan2@andrew.cmu.edu

Shirwadkar, Harshad
hshirwad@andrew.cmu.edu

Snavely, Will
wsnavely@andrew.cmu.edu

Wang, Xinkai
xinkaiw@andrew.cmu.edu

April 2, 2015

Abstract

In this document, we'll discuss our implementation of the InterPlanetary File System (IPFS). IPFS combines a variety of technologies, including: distributed hash tables (e.g. Kademlia); version control (e.g. Git); and block exchange protocols (e.g. BitTorrent). IPFS seeks to be the next-gen platform for hypermedia data, a more efficient and scalable replacement for HTTP.

Contents

1 Introduction **2**
1.1 Project Overview . . . 3
1.2 Value Proposition . . . 4
1.3 Skills Challenge . . . 4
1.4 Background . . . 4
1.4.1 BitTorrent . . . 4
1.4.2 Distributed Hash Tables . . . 5
1.4.3 Version Control . . . 5
1.5 Naming . . . 5
1.6 Security . . . 5
1.7 Reliability . . . 6

2 Functional Requirements **6**
2.1 Overview . . . 6

3 System Design **6**
3.1 Identities . . . 6
3.2 Network . . . 7
3.2.1 Concepts . . . 7

3.2.2 Host Connectivity Interface 8
3.3 Routing 8
3.3.1 Bootstrapping 8
3.3.2 Obtaining Resource 9
3.3.3 Routing Table Data Structure 9
3.3.4 Message Passing 11
3.4 Block Exchange 11
3.5 Objects 11
3.5.1 Object Storage 12
3.5.2 Object pinning 12
3.5.3 Object publishing 12
3.5.4 Object Structure 13
3.6 Files 13

4 Demonstration **14**

5 Project Management **15**

1 Introduction

HTTP has been a massive success. It facilitates the most popular distributed file system in history: the World Wide Web. Massive amounts of data are shared over this platform. This includes traditional hypermedia, such as text, images and video, as well as arbitrary files. Many cloud providers implement storage platforms over HTTP, such as Amazon's Simple Storage Service (S3). While HTTP has been successful, there are drawbacks, and detractors. If you've ever taken a class from Garth Gibson at CMU, you've heard from one. Many advances were made in storage systems over the past few decades, and these HTTP-based systems disregard many of them. HTTP is simple and functional, and many successful systems have been built on top of it; but how do we move hypermedia forward?

Enter the InterPlanetary File System (IPFS), a peer-to-peer distributed file system proposed by Juan Benet [1]. Broadly, IPFS seeks to satisfy the present day demand for "lots of data, accessible everywhere." Netflix is a perfect example of this. Users want to watch videos, perhaps high definition videos, no matter where they are, no matter what device they are using. Clearly, given its existence and success, it is possible to build Netflix on top of HTTP, though we must wonder, at what cost, and with what infrastructure? How do we ensure that customers across the globe can watch their videos without much delay?

We can think of IPFS as a platform that abstracts away the complexity of such an infrastructure, facilitating the creation of applications with heavy distributed data requirements. IPFS hopes to accomplish this by leveraging the wealth of edge-nodes in today's Internet. Hosts across the world participate in a peer-to-peer distributed file system, sharing the burden of storing and distributing data. This general idea is not new. Look no further than BitTorrent for a similar concept. Hosts in a BitTorrent swarm share blocks of a file with each other, reducing the need for a central file host, and improving bandwidth.

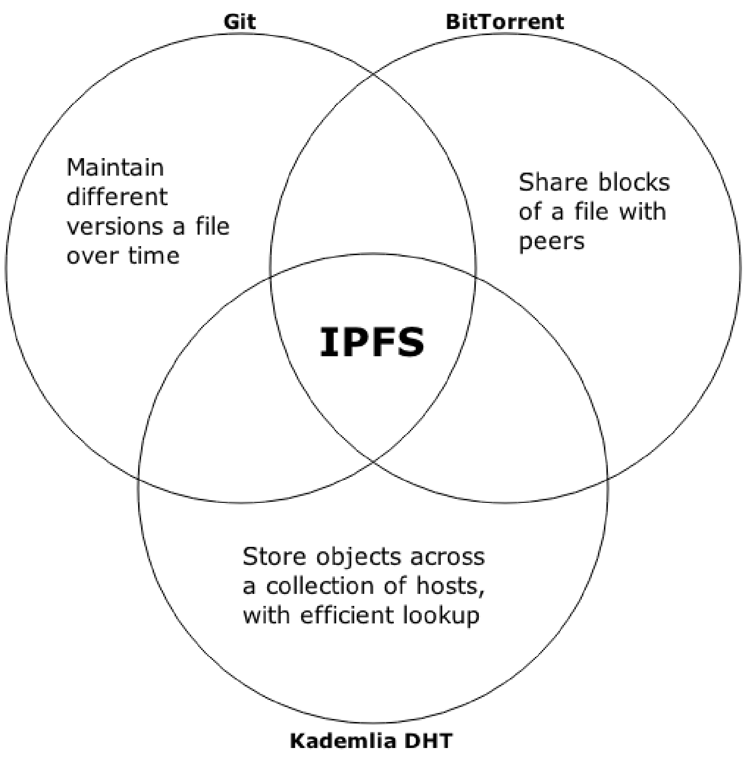
Git
BitTorrent
Maintain different versions a file over time
Share blocks of a file with peers
IPFS
Store objects across a collection of hosts, with efficient lookup
Kademlia DHT

Figure 1: IPFS synthesizes three technologies.

IPFS takes BitTorrent and synthesizes it with other successful technologies, with the hope of building a general-purpose distributed file system. These other technologies include distributed hash tables, and GIT version control.

1.1 Project Overview

We intend to implement a functional subset of the IPFS protocol (the 'back-end'), and host a simple hypermedia application on top of it (the 'front-end').

The back-end part of this project will involve building a peer-to-peer network of nodes, who will share files with each other using a block-sharing protocol, in the manner of BitTorrent. The structure of this network will be motivated by the Kademlia distributed hash table. The IPFS specification calls for versioning files using a Git-inspired mechanism, which will be a stretch goal in our implementation.

The front-end will be a video-viewing service, like Netflix. Ideally, this application will be trivial to implement on top of our back-end. In fact, we hope to construct this front-end from existing applications. Our IPFS file system will be mounted in the operating system like any other file system, and video files will be played using an existing media player.

1.2 Value Proposition

Building a data-intensive application in today's Internet is not easy, especially at a global scale. We hope to make inroads towards implementing an easy-to-use data layer for these applications. The components of this layer are not novel, but their synthesis results in a system greater than the sum of its parts.

The hope is that such applications are not only easier to build, they are more efficient, and provide better consumer experiences. It's inarguable that consumers will demand more and more data, as time moves forward. At some point, the current way of doing things will start to break down. One way to address this problem is to push more work to the edge of the network. Today, this is done accomplished with content distribution networks and other forms of caching. We think that we can push things even further to the edge of the network, letting end-hosts themselves share the work of distributing data. This is one way we can facilitate the data-intensive applications of the future.

1.3 Skills Challenge

We intend to implement our IPFS stack using the Erlang programming language, in order to satisfy the skills challenge requirement of this project. Erlang is particularly suitable for distributed, event-based programs, and we hope that, despite our inexperience with the language, we'll be able to put together a working stack.

The biggest potential issue here is an unfamiliar programming paradigm. Erlang borrows techniques from functional programming languages. We do not, collectively, have much experience with functional programming, so it will prove interesting to push our comfort zone in that respect.

1.4 Background

1.4.1 BitTorrent

BitTorrent is a peer-to-peer file sharing protocol that is used very widely; in fact, it is one of the most common P2P protocols. As of February 2009, P2P traffic from BitTorrent collectively accounted for 43%-70% of total Internet traffic (depending upon the geographic location [2]).

The high level view of BitTorrent is as follows: a group of hosts want to possess a file. Instead of every host downloading the file from a central server, hosts share chunks of the file with each other, using multiple connections to pull in chunks from peers. A special host called the tracker coordinates the swarm of peers.

A significant issue in BitTorrent is fairness. Some implementation of BitTorrent use a "Tit-For-Tat" strategy to mitigate the problem of "free-riding": downloading chunks without sharing them [3]. Our realization of IPFS will use a simpler version of BitTorrent's Tit-for-Tat strategy to enforce fair block exchange.

1.4.2 Distributed Hash Tables

We are proposing a peer-to-peer, distributed file system. A central question is: how will we structure our network, and how do you find data in the network? We intend to address these issues with the Kademlia Distributed Hash table. Kademlia specifies both the structure of the P2P network as well as the protocol for node communication and data look-up. We want to use Kademlia primarily because of its efficiency: $O(\log N)$ nodes are contacted to locate a given node. We will provide more details on the nature of this network later in the document.

The basic idea behind a distributed hash table is as follows. We want to share the responsibility for object ownership among a group of hosts. A key space is specified, and hosts claim ownership over segments of this key space. Hosts then form an overlay network. Essentially, each host becomes a router, informing peers how to find a certain segment of the key space. The nature of this routing dictates the speed at which a desired node can be found. This is a dynamic, fault tolerant system: hosts may leave and join at will. Kademlia uses a hierarchical organization of nodes to improve the performance of routing, as discussed above.

1.4.3 Version Control

Version Control Systems, like Git, model changes to files over time. IPFS uses ideas from Git to manage modifications to the file system over time (adding/removing files, changing files). However, Git can also be interpreted as a standalone file system, in particular a *content-addressable* (CAS) file system. This means that the identifier used to retrieve a file is inextricably linked to the contents of that file. For example, we might retrieve a file with the SHA-1 hash of its contents. Files are therefore immutable (as changing a file changes its hash, and thus its identity). The authors of IPFS claim that Git lends many useful properties to distributed file systems. We do not intended to aggressively pursue this aspect of the project. We will, at least, maintain the property that files are content-addressable, as this is fundamental to the file system. Any further work in this direction will be considered a stretch goal.

1.5 Naming

A significant issue with content-addressed systems is human-friendly naming. It is not practical nor user-friendly to always reference objects by their hash. The IPFS paper suggests developing a naming service to map names to hashes, called IPNS (InterPlanetary Naming Service). While this is required for a real-world deployment of IPFS, we will assume that users can look up hashes out of band. That is, we are scoping IPNS out of our implementation.

1.6 Security

Security is always a significant issue in P2P systems. When peers have the power, one must worry about Byzantine peers. While we will explore some techniques for ensuring fairness (e.g. the tit-for-tat approch of BitTorrent), security and Byzantine-resistance will not be a focus of this project. We will largely assume that participating nodes correctly follow the IPFS protocol.

1.7 Reliability

Reliability is a central concern in file systems. One simply cannot afford to lose certain files. Kademlia has some mechanisms built in to increase the reliability of stored objects, which we will explore over the course of this project.

2 Functional Requirements

2.1 Overview

The functional requirements of our system are somewhat simple. Most of the complexity is pushed into the overlay network. A user or programmer will interact with IPFS much as if it were a normal file system, as the following diagram demonstrates.

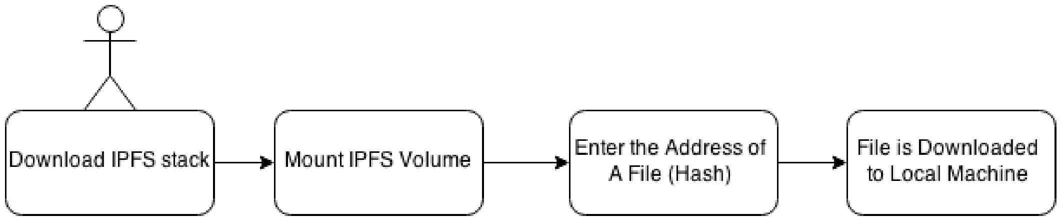
Download IPFS stack
Mount IPFS Volume
Enter the Address of A File (Hash)
File is Downloaded to Local Machine

Figure 2: IPFS is simple from the user's perspective.

Issues like bootstrapping, and joining the Kademlia DHT, are handled by the mounting process. As mentioned above, a real-world implementation of IPFS should include a naming service, but we will assume that the client obtains file identities (hashes) through some other means, due to lack of time. For example, a client might be e-mailed a list of hashes identifying files of interest. Once a user knows the hash of a file or directory, they can browse to it and download it to their local system. At this point, they will share the file with others. Sharing is handled transparently by the IPFS client code.

So, for all intents and purposes, the client will interact with the mounted IPFS volume like a local drive. The latency associated with obtaining files of course will be higher, but the access method will be similar.

3 System Design

3.1 Identities

Let's first establish what individual nodes look like in our system. Nodes participate in a peer-to-peer distributed hash table, based on Kademlia.

When a node wishes to join the P2P network, they first must establish their identity. Every node in the network has a unique identity. While there are various techniques for

```
def GenerateIdentity(int difficulty) {
    do {
        (PubKey, PrivKey) = PKI.genKeyPair()
        NodeId = SHA1(n.PubKey)
        p = count_preceding_zero_bits(SHA1(NodeId))
    } while (p < difficulty)
    return (NodeId, PubKey, PrivKey)
}
```

Figure 3: Identity generation via static crypto puzzle.

establishing this identity, we will use a technique from Kademlia. First, we generate a public/private key pair. This assumes the existence of a public key infrastructure (PKI). A node's identity (NodeId) is then derived from the generated public key, using a cryptographically secure hash function. No specific hash function is specified by IPFS, so we will likely use SHA-1. See Figure 3 for a possible implementation of this identity generation scheme, which makes use of a static crypto puzzle [4].

Once we have our identity established, we need to start talking to peers. Next, we'll discuss the underlying networking layer on which this communication happens.

3.2 Network

3.2.1 Concepts

IPFS nodes communicate regularly with many other nodes in the network, potentially across the wide Internet. In this section, we'll discuss the features of the IPFS connectivity layer

First, regarding the transport layer: IPFS is able to use any transport procotol, though is best suited for WebRTC Datachannels, which is mainly for browser connectivity, or UTP [1]. For the sake of simplicity, we will likely use TCP for our transport layer.

The second feature is the reliability. IPFS can provide reliability if underlying networks do not provide it, using UTP or SCTP.

IPFS also uses ICE NAT traversal techniques. NAT traversal is of importance for P2P and Voice over IP (VoIP) deployments. Many techniques exist, but no single method works in every situation, since NAT behavior is not standardized. Many NAT traversal techniques require assistance from a server at a publicly routable IP address. Some methods use the server only when establishing the connection, while others are based on relaying all data through it, which adds bandwidth costs and increases latency, detrimental to real-time voice and video communications. We will likely scope out NAT traversal from our project, due to these complexities.

Lastly, IPFS also provides mechanisms for integrity and authenticity. It optionally checks the integrity of messages using a hash checksum, verifies the authenticity of messages using a HMAC with sender's public key (generated when the node enters the system).

Conn	a connection to a single Peer
MultiConn	a set of connections to a single Peer
SecureConn	an encrypted (tls-like) connection
Swarm	holds connections to Peers, multiplexes from/to each MultiConn
Muxer	multiplexes between Services and Swarm. Handles Requeat/Reply
Service	connects between an outside client service and Network
Handler	the client service part that handles requests

Figure 4: Subcomponents of a host in IPFS

3.2.2 Host Connectivity Interface

A host in IPFS connects to other hosts, encrypts communications, and multiplexes messages between the network's client services and target hosts. See Figure 4 for details on the connectivity interface.

3.3 Routing

Our goal is to implement Kademlia P2P routing algorithms, which feature $O(\log N)$ time complexity, for finding the peers that hold certain objects. Routing is implemented by the following sequence of steps.

3.3.1 Bootstrapping

A peer must first execute the bootstrapping process, via the interfaces described in Figure 5. The new peer needs to get information about other, existing peers to construct its initial view of the distributed hash table. After bootstrapping, a new host will have acquired a list of active peers in the P2P network.

Bootstrap()	Builds a list of active peers by requesting random peerids
FindPeer(peerid)	Get peer information to construct initial DHT information

Figure 5: Bootstrapping interface

Next, the new host builds its routing table, and propagates updates to peer routing tables. A peer must announce which resources it is able to store, base on its assigned key range. A peer may also replicate its objects to nearby peers, to satisfy reliability constraints, and to improve the availability of that object's chunks. The interfaces required for this step are laid out in Figure 6.

putValueToPeer(peerid, key, value)	Put a resource (value) on a peer with given peerid.
Provide(key, value)	Announce to its nearest peers that I can provide resource with a given key.
AddProvider(key, peerid)	Add a provider peerid with a given key to infer that a specific peer has a resource with key
GetProviders(key)	Get a list of providers with a given resource key, so that a node can request resource from a peer

Figure 6: Provider interface

getValue(key)	Get resource from local with a given key if exists, otherwise, find resource according to the provider list or issue a new query to peers in the routing table.
getClosestPeers(key)	Get a list of closest peers to a resource with key, the distance is also specified by XOR
putValue(key, value)	Put a resource (value) it has received to local data store.

Figure 7: Resource Access Interface

3.3.2 Obtaining Resource

When a peer wants to access a resource, it needs to first determine whether the resource is present locally; if not, it will try to iteratively find the resource, starting with its nearest neighbors in the DHT. When a host locates the resource, other peers can terminate their iterative search. The resource access interface is laid out in Figure 7.

3.3.3 Routing Table Data Structure

In Kademlia, the routing table is represented by a data structure called the K-Bucket. The K-Bucket maintains the ”distance” between the local host and peers in the system. The distance between two hosts in a Kademlia DHT is defined as the bitwise-exclusive-or of their node IDs. If the node ID is 160 bits (a SHA-1 hash), then 160 K-Buckets are required to store different levels of distance relative to local host. Bucket size is limited by the parameter K, therefore some eviction policy is required to handle cases when a bucket overfills.

Generally, Kademlia prefers peers that have maintained lengthy residency in the bucket. This results in an LRU-like eviction policy. When a new peer is discovered that can be inserted into routing table, and the bucket it belongs to is full, then the algorithm first checks whether the least-recently-seen peer is active by sending ping requests. If the ping request fails, Kademlia will assume the peer has left the P2P network, evicts the old peer, and adds the new peer to the bucket.

The K-Bucket interface is shown in Figure 8.

Update(peerid)	Update a bucket in the routing table specified by prefix of the peerid. Move or add peer to front of the bucket if the bucket is not full. Otherwise, evict the least active node from the back of the bucket list.
Remove(peerid)	Remove a peer from a bucket specified by its peerid
Update(peerid)	Update a bucket in the routing table specified by prefix of the peerid. Move or add peer to front of the bucket if the bucket is not full. Otherwise, evict the least active node from the back of the bucket list.
Remove(peerid)	Remove a peer from a bucket specified by its peerid.
MoveToFront(peerid)	Move the a recent seen peerid to front of the bucket.
PushFront(peerid)	Push a new peerid to the front of the bucket.
PopBack(peerid)	Pop back the least recent seen peerid, which is a evict operation in the k-bucket of the routing table.
Ping(peerid)	Ping a peer to test its connectivity.
NearestPeers(peerid, count)	Return a list of at most count number of peers that is closest to itself, closest means one hop to in the XOR distance to itself. We can use this function to find the nearest peers in the routing table

Figure 8: K-Bucket Interface

sendMessage(peerid, message)	Send a message to a specific peer with peerid.
recvMessage(message)	Receive message from a peer; this function can be non-blocking.
handleNewMessage(message)	Handle a received message, performing some action based on its type.

Figure 9: Messaging Interface

3.3.4 Message Passing

In order to achieve the above functionality between peers, peers use message passing to communicate with each other. There are message types of PUT_VALUE, GET_VALUE, PROVIDE, GET_PROVIDER, etc., to indicate the kind of action the sender peer wishes to perform. The message passing interface is specified in Figure 9.

One more thing to notice is that it is inevitable to encounter errors and exceptions when doing routing. As a result, we may as well implement a logger to record these anomalies.

3.4 Block Exchange

IPFS objects consist of one or many blocks. Block exchange layer(BEL) of IPFS is responsible for fetching and serving blocks.

Every block in IPFS is identified by cryptographic hash of its contents. BEL maintains a list of the block IDs that the peer owns (I-HAVE-LIST) and another list of the blocks that the peer wants(WHO-HAS-LIST). BEL sends the WHO-HAS-LIST to all the neighboring peers periodically as well as on updates. BEL can receive WHO-HAS-LIST from multiple peers. It serves all the blocks that it has in its I-HAVE-LIST.

In order to implement fairness, we will use a very simple algorithm. Every peer will store following information for all its neighbors: 1. Number of bytes received from the peer (recvCount), 2. Number of bytes sent to the peer (sendCount). We define “a good peer” as the one who has a high value for ((recvCount) / (sendCount)) (Fairness degree). All the peers should try to maintain the value of fairness degree near 1. If fairness degree falls below a certain threshold, the BEL reduces the rate at which it serves WHO-HAS-LIST for that peer. We are not yet clear of how the sending rate should behave to falling fairness degree. We plan to determine the behavior experimentally. Upper layers communicate with BEL using the API “blockGet(hash)”.

3.5 Objects

The users of IPFS store their content in “IPFS-Objects”. IPFS object layer uses functions provided by BEL and DHT to avail object storage and sharing capabilities. IPFS maintains all of its objects in a directed acyclic graph (DAG). Just like blocks, every object is identified by its cryptographic hash. An edge in the DAG is also a cryptographic hash of the object that it is pointing to.

Using cryptographic hash for the identification provides following advantages:

- Unique identification: All the content objects are identified uniquely.

pinObject(objectID)	Download a global object and store it in local storage.
publishObject(object)	Make object globally routable by adding it to the DHT.

Figure 10: Object layer APIs

- Tamper resistance: Since, the hash of the content is the ID, it is easy to verify if the content is tampered or not.
- Data deduplication: If 2 applications share the same content, only 1 copy of the actual content gets stored.

IPFS objects can be classified into following categories :-

- Global Objects: Objects that are globally addressable. Essentially, these objects must be published using DHT APIs. (Please refer **??** for the publishing API).
- Local Objects: Objects that are not globally addressable. These objects resemble to local files on your computer.

3.5.1 Object Storage

The IPFS stores the objects in a pre-configured directory. Note that IPFS relies on local file system to store and manage content on Disk. The object-store can simply be a native file system or a database application running on top of it. We plan to use Google's levelDB database to manage content on disk. Erlang has LevelDB APIs which would make job of storing data on disk easy for us [5].

At the boot time, IPFS iterates through the object storage directory to populate its internal data structures.

3.5.2 Object pinning

IPFS allows users to download content that they think is useful. The action is called pinning. When pinObject() API is invoked, IPFS downloads the object to the data store thereby saving the lookup time for next reference.

3.5.3 Object publishing

As suggested by the original IPFS idea, IPFS has the potential of replacing the world-wide-web. Thus, it is important for IPFS to provide the ability of publishing content online. This task is effectively managed by the DHT module of IPFS. On invocation of publishObject() API, IPFS adds the object in the DHT. Thus, the object becomes accessible to the rest of the the world.

3.5.4 Object Structure

IPFS places very minimal restrictions on object structure. This allows applications to extend upon it and build application specific structures. Here are how object structures look:

```
structure IPFSObject {
    IPFSLink[] links;
    byte[] data; // Object private data
};

structure IPFSLink {
    String name; // Link name
    Hash hashID; // Cryptographic hash identifier of the target object
    Int size; // Size of the target
};
```

3.6 Files

IPFS provides file-system like interface by using the DAG objects. In order to provide inherent versioning support, IPFS borrows file / directory structures from git versioning system. Git comprises of four core structures:

- Blob - Variable sized data object
- List - A collection of blobs and lists
- Tree - A collection of lists and trees
- Commit - A snapshot of a tree

A file in IPFS is either a Blob if the size of the file is small enough otherwise it is a list. Note that using lists, IPFS avoids duplication inherently. Directories are represented by trees.

We use JSON formats to store these structures.

IPFS structures look like this:

```
IPFS Blob:
{
    "data": "blob data"
}

IPFS List:
{
    "data": ["blob", "list", "blob"],
    "links": [
    { "hash": "XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x",
      "size": 1894 },
    { "hash": "XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5",
      "size": 1941 },
    { "hash": "XLWVQDqxo9Km9zLyquoC9gAP8CL1gWnHZ7z",
      "size": 5286 }
    ]
}

{
    "data": ["blob", "list", "blob"],
    "links": [
    { "hash": "XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x",
      "name": "less", "size": 189458 },
    { "hash": "XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5",
      "name": "script", "size": 19441 },
    { "hash": "XLWVQDqxo9Km9zLyquoC9gAP8CL1gWnHZ7z",
      "name": "template", "size": 5286 }
    ]
}
```

Directory traversal becomes very simple because of the DAG links in the object layer. In order to limit the scope, we will only concentrate on Blobs, Lists and Trees. [2]

4 Demonstration

We intend to demonstrate the functionality of our system with a simple hypermedia application. The purpose of this application will be to share a collection of high definition videos. The nature of these videos is still to be determined. One idea we had was to host the collection of class lecture videos from this semester.

We will seed the videos on an initial set of machines. We might use cluster machines, or cheap Amazon machines to host this initial collection of videos.

We will then invite users to try out our application on client machines, laptops running on the normal Internet. We will develop a lightweight application which will mount the IPFS file system locally, and display the list of available videos. Users can then watch videos on their local machine. We will use a simple visualization to show how the file is being realized locally. For example, we might expect to pull some file chunks from the various seed servers, and some chunks from other clients who have already downloaded this video. See Figure 11 for a diagram of this application.

This application can be scaled in several ways. As more peers join, they can share chunks with each other. New seed machines can be added, and they can automatically download chunks from other machines, just like any other peer.

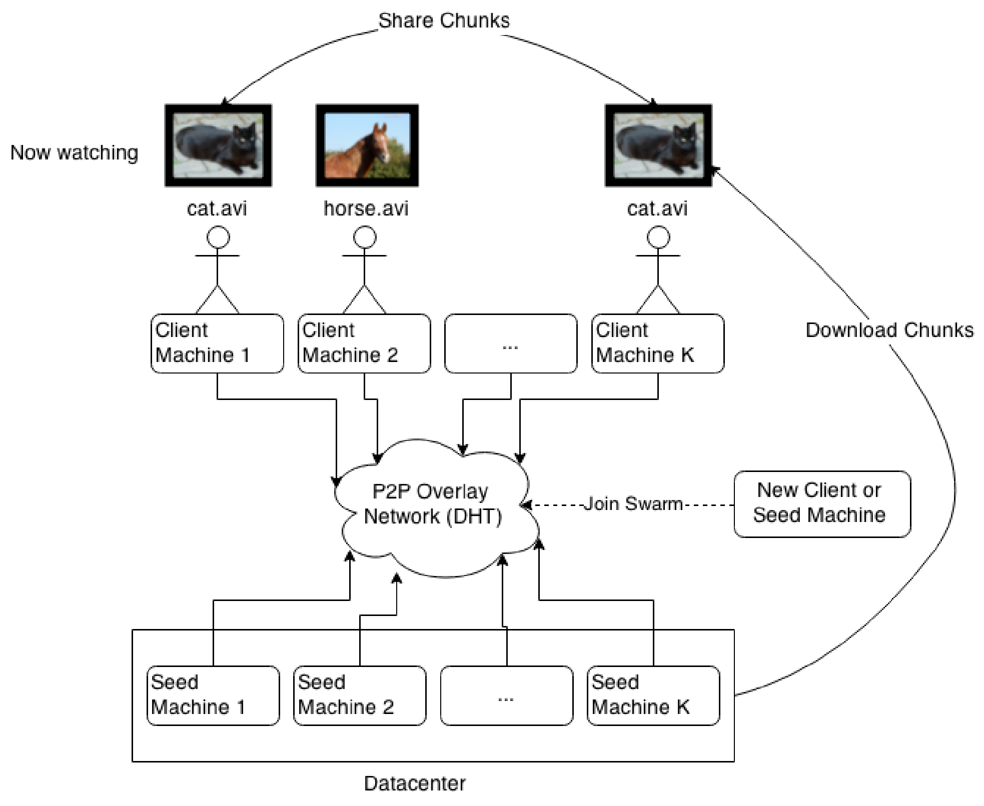

Figure 11: Proposed video application

5 Project Management

Our plans for managing this project are laid out below. First, here are some broad strategies we'll follow.

1. We have arranged for two weekly meeting times, on Tuesday and Thursday. We will determine on Tuesday if we need the second meeting.
2. We have create a private Git repository that we intend to use for sharing code.
3. We have broadly split the major areas of the application (DHT, block exchange, etc.) to different members of the team.

The rough assignment of subject areas is as follows.

Topic	Team Members
Node Interconnectivity	Minghan
Routing (Distributed Hash Table)	Martin
File Block Transfer	Harshad
Project Management, Front End	Will

It's assumed that all members will contribute to programming and assist with design as needed.

Below, we set out a rough schedule for this project.

Week	Milestone
March 23-27	Produce first iteration of the interfaces for block exchange, routing, connectivity. Lay foundation for client code.
March 30-April 3	Decide on transport layer, and implement basic pieces of the connectivity layer. At this points nodes should be able to talk to each other. Implement basic node identity (key, id generation). Finalize all system interfaces.
April 6-10	Produce first iteration of block transfer protocol. No fairness will be implemented yet, simply the ability to exchange blocks of a file.
April 13-17	Produce first iteration of the routing system. Nodes should be able to join the DHT and find the location of objects. Implement the first iteration of the front end.
April 20-24	Start merging the routing and block transfer systems. At this point, nodes should be able to locate and transfer blocks. Fix bugs and iterate on the routing/block transfer implementation as necessary.
April 27-May 1	At this point we should have a mostly functional back-end implementation, and we can start merging it with the back-end.
May 4-May 8	This week will be devoted to polishing our product for the final presentation, and preparing our writeup.

References

[1] Benet Juan. Ipfs - content addressed, versioned, p2p file system. 2014.

[2] Hendrik; Klaus Mochalski Schulze. Internet study 2008/2009. 2009.

[3] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[4] Ingmar Baumgart and Sebastian Mies. S/kademlia: A practicable approach towards secure key-based routing. In *Parallel and Distributed Systems, 2007 International Conference on*, volume 2, pages 1–8. IEEE, 2007.

[5] Basho tech. Erlang bindings to leveldb datastore. *https://github.com/basho/eleveldb*, 2009.