18-842 Final Project Report
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

May 4, 2015

Abstract

In this paper, we'll discuss our attempt to implement the Interplanetary File System, a peer-to-peer (P2P) distributed file system. We'll discuss our design, implementation, and test approach, as well as future work.

Note to Instructor: You have permission to publish to document to the course website, if desired.

Contents

1 Introduction

1.1 Motivation

Suppose a classroom full of students are told to watch a video on their personal device, perhaps supplemental material for a lecture. Instead of having each student download the content from a central server, what if they could share the content amongst themselves? In brief, the students could enlist their devices in a P2P network, sharing pieces of the content with each other as they become available. Thus, the external content provider only needs to be contacted once, to seed the required data into this network.

1.2 Existing Work

1.2.1 BitTorrent

BitTorrent is a widely-used P2P file sharing protocol; 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 [1]). Nodes in a BitTorrent swarm share chunks of files with each other, using multiple connections to pull in chunks from peers. A designated node called the tracker coordinates the swarm of peers.

1.2.2 Distributed Hash Tables

DHT's address the problem of sharing object ownership among a group of hosts. As with conventional hash tables, objects are mapped into a key space with a hash function. A node participating in the DHT takes ownership over a range of the key space, thereby becoming responsible for storing objects with hashes falling in that range. Nodes also serve as routers, directing object hash queries.

Kademlia is a very popular DHT. In Kademlia, both nodes and objects are assigned a 160-bit identifier. The *distance* between two identifiers is defined as their bit-wise XOR. Objects are stored on the nodes they are closest to, according to this distance metric. Nodes are required to maintain information about both nearby and distant peers, which facilitates fast routing convergence.

1.2.3 Content-Addressable File Systems

In content-addressable file systems, the identifier used to retrieve a file is inextricably linked to the contents of that file. For example, we might reference 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).

1.2.4 Distributed Version Control Systems

Various distributed version control systems (such as Git[2]) have had been successful in developing new distributed work-flows. Git had a lot of impact of various distributed file system design. For example, various distributed file systems (including IPFS) borrow the idea of self certifying names (SHA1 hash) from git. However, the ideas of git were never studied in context of very high throughput systems (like web itself). In our IPFS proposal, we planned to assimilate various git structures (blobs, objects, trees etc). As the following sections show, we were partially successful in achieving this goal. To be precise, we implemented *Content Addressed Blobs* and File Objects. We could not implement trees due to time limitation.

1.2.5 Information Centric Networking

In the context of Future Internet Research, addressing content by its hash and providing a look-up based on the hash is a very popular trend. Many Future Internet Proposals address this in various ways. Collectively, these architectures are referred to as Information Centric Networks. Named Data Networks (NDN) [3] has the notion of *Interest Packets* and *Data Packets*. Every data packet has a name associated to it. A data packet satisfies an interest if the data packet is sent as an response to the interest packet. XIA (eXpressive Internet Architecture)[4] identifies a content chunk by SHA1 hash of its content. It relies on a consistent, reliable naming service to provide hash corresponding to a name. So, the important takeaway is that the world is indeed becoming content-centric. A lot of active research in various organizations and universities is based on moving towards information centric world. Our proposal of IPFS is in a way, aligned with that goal.

1.2.6 IPFS

This project was motivated by research by Juan Benet [5], regarding a replacement for HTTP called the Interplanetary File System (IPFS). This system, in turn, incorporated concepts from BitTorrent, Kademlia, and Git. In brief, nodes across the Internet participate in a massive BitTorrent swarm. Data is located in this network using a Kademlia-style distributed hash table. Data is versioned using a Git-like scheme. Benet implemented a prototype of this system in the Go programming language.

1.3 Proposed Work

Our goal was to implement a functional subset of the IPFS system (the 'back-end'), and host a simple hypermedia application on top of it (the 'front-end'). For the back-end, we planned to build a P2P network of nodes, sharing files with each other using a block-sharing protocol. We planned to distribute and locate objects based on concepts from Kademlia. We also wanted to implement a simple form of file versioning.

For the front-end, we planned to build a simple video viewing service, DSFlix. Distributed systems students can watch lecture videos on this platform.

2 Functional Specification

In this section, we'll discuss the functional requirements for our project. We will talk about the requirements for both the back-end and front-end.

2.1 Globals

There are several global configuration options in this system.

1. A *distance metric* is defined, $D(x, y)$, where x and y are object identifiers (either for nodes or data blocks).
2. A *replication factor* K. This defines the number of replicas that host a given block.

2.2 Entities

2.2.1 Blocks

The basic unit of data in this system shall be a *block*. A block is simply a chunk of binary data with an identifier. Blocks shall be *content addressed*. That is, the block identifier uniquely corresponds to the binary data within the block. The size of a block is not specified.

A *file* is a concatenation of blocks. A file is described by a *metadata block*, a block whose contents is a list of block identifiers. See Figure 1 on page 6 for an example. For this project, metadata blocks support just one level of indirection. However, a slightly more sophisticated implementation would allow metadata blocks to contain links to other metadata blocks, ad infinitum.

A file may have a human-readable name. A mapping must be maintained between this name and the identity of the file's metadata block. This mapping is maintained by a *naming service*, described below.

2.2.2 Nodes

A node is any host participating in the network. Nodes maintain the following properties.

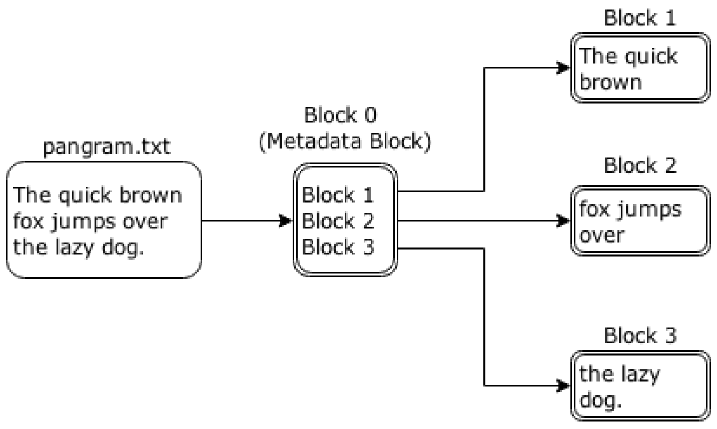


Figure 1: A high level view of files.

Property	**Description**
ID	A globally unique identifier for this node.
Public Key	A public key, for asymmetric encryption. This is shared with other nodes.
Private Key	A private key, for asymmetric encryption. This should be kept secret.
Network Address	The details required to communicate with this node over the network.
Peer List	A list of other nodes we know about.
Block List	A list of blocks we have stored locally.

A node must be prepared to handle a variety of messages, received at its network address. When a node receives a message from a peer it has not seen before, that peer is added to the node's local peer list.

Message	**Description**	**Response**
Ping()	A heartbeat message. Determines if the node is responsive.	The node must send back an acknowledgement to the sender, but the contents of this message are unspecified.
Store(Id, Data)	Store a block with the given id and contents in this node's local storage.	The node must send back an acknowledgement to the sender, but the contents of this message are unspecified.
Get(Id)	Retrieve the block with the given id from the node's local storage.	If the node has a copy of the block locally, the binary content of the block is returned to the sender. Else, the sender is notified that the desired block is not present.

2.2.3 Bootstrap Service

The back end shall implement a node bootstrapping service. The bootstrapper maintains a list of nodes who have joined the network, and exposes the following interface.

Function	**Description**
AddNode(Id, PubKey, Address)	Register a new node with the bootstrapping service. The id, public key, and network address of the node must be specified.
GetNodes()	Return the list of nodes registered with the bootstrapping service.

2.2.4 Naming Service

The back end may optionally implement a naming service. This service maps human-readable names to metadata block ids. A given file name may be associated with multiple metadata id's, representing different versions of the same file. Each id associated with a file name also has a numeric version id. New versions must have a version id strictly greater than all existing version ids.

Function	**Description**
AddFile(Name, Id)	Register a new mapping with the naming service. This mapping will have version id 0. If a mapping already exists for the given name, this should fail.
AddFileVersion(Name, Id, VersionId)	Add a new version of the file with the given name. The version id should be an integer. It must be strictly greater than all existing version ids.
GetFile(Name)	Returns the id of the metadata block associated with the given file name, if one exists. If multiple versions of this file exist, the latest version is returned.
GetFile(Name,VersionId)	Returns the id of the metadata block associated with the given file name and version id, if one exists.

2.3 Operations

2.3.1 Node Creation

The steps for creating a new node are as follows.

1. Generate a unique node id, as well as a public/private key pair. Initialize the Block List and Peer List to be empty. Allocate a network address at which the node can be reached.

2. Publish the node id, public key, and network address to the Bootstrap Service, via the `AddNode` interface.
3. Retrieve the list of active nodes from the Bootstrap Service through the `GetNodes()` interface. Add these nodes to the Peer List.
4. Send a `Ping` message to each node in the Peer List, as a means of informing them we have joined the network.
5. Begin handling messages.

2.3.2 Block Publishing

The following specifies how to publish a block with id b to the network, from a given node.

1. For each node n in the Peer List, compute the distance $D(b, n)$.
2. Send a `Store` message to the the K closest nodes, directing them to store this block.

2.3.3 Block Retrieval

A block with id b is can be retrieved from the network as follows

1. For each node n in the Peer List, compute the distance $D(b, n)$.
2. Send a `Get(b)` message to the K closest nodes, in order, until a positive response is received.

2.3.4 File Publishing

Suppose a file f lives on the local disk of a node. The algorithm for publishing the file is as follows.

```
PublishFile(f):
    # Read the contents of the file
    data = open(f).read()
    # Convert the contents into blocks
    blocks = blockify(data)
    ids = list()
    foreach block in blocks:
        id = hash(block)
        # Publish the block to the network
        PublishBlock(id, block)
        ids.add(ids)
    # Publish the metadata block for this file
    metadata_id = hash(ids)
    PublishBlock(metadata_id, ids)
    # Associate the file name with the metadata id
    AddFile(f, metadata_id)
```

The `blockify` and `hash` functions are implementation defined.

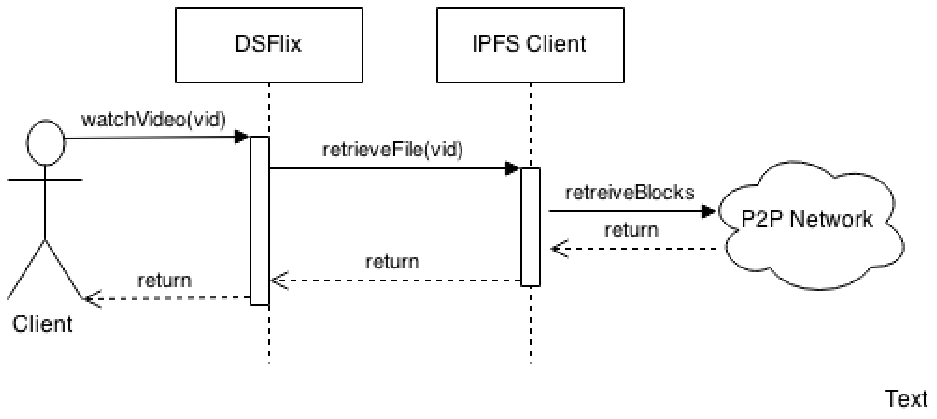


Figure 2: Watching a video in DSFlix.

2.3.5 File Retrieval

To retrieve a file *f* from the network, the following procedure should be followed.

```
RetrieveFile(f):
    # Get the metadata id from the naming service
    metadata_id = GetFile(f)
    # Retrieve the metadata block
    block_ids = RetrieveBlock(metadata_id)
    blocks = list()
    foreach id in block_ids:
        data = RetrieveBlock(id)
        blocks.add(data)
    # Assemble the blocks together
    file_data = concat(blocks)
    # Write the file to the local disk
    open(f).write(file_data)
```

2.4 User Interface

Next, we'll discuss the functional requirements for the user interface.

2.4.1 DSFlix

The main UI deliverable will be a hypermedia application called DSFlix. This application will allow users to browse and watch a selection of distributed-systems related videos. The user's machine will be a participant in an IPFS P2P network. When the user selects a video to watch, the `RetrieveFile` function is used to download the file to the local machine. See Figure 2 for a sequence diagram describing this interaction. See Figure 3 for an example of the dsflix UI.

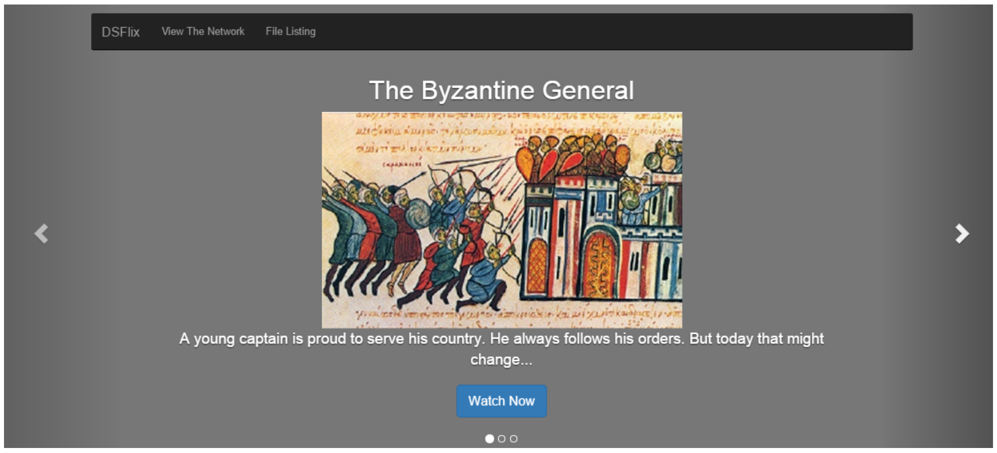

DSFlix
View The Network
File Listing
The Byzantine General
A young captain is proud to serve his country. He always follows his orders. But today that might change...
Watch Now

Figure 3: The DSFlix UI.

2.4.2 Network viewer

The network viewer UI will display a birds-eye view of the P2P network. We will query the bootstrap table to obtain a list of nodes, and show their relative position on a Kademlia ring. We will also show simple information about each node, such as its id, network address, and block count. See Figure 4 for a screen shot of this UI.

2.4.3 File Lister

The file lister UI visualizes the data in the naming service, similar to how the network viewer visualizes nodes. File names and their corresponding metadata hashes are shown in a simple list. See Figure 5 for a screen shot of this UI.

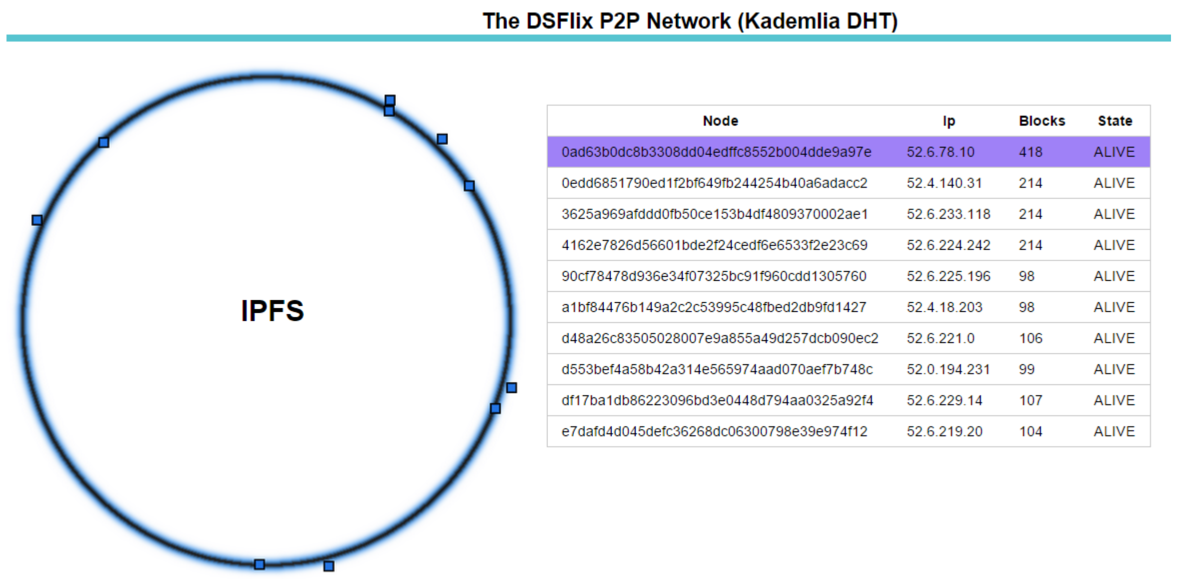
2.4.4 Command-line Interface

To facilitate the above graphical interfaces, we will implement a command-line interface to the IPFS client. This client will support operations such as:

1. Starting a new IPFS node.
2. Publishing a file to the P2P network.
3. Downloading a file from the P2P network.

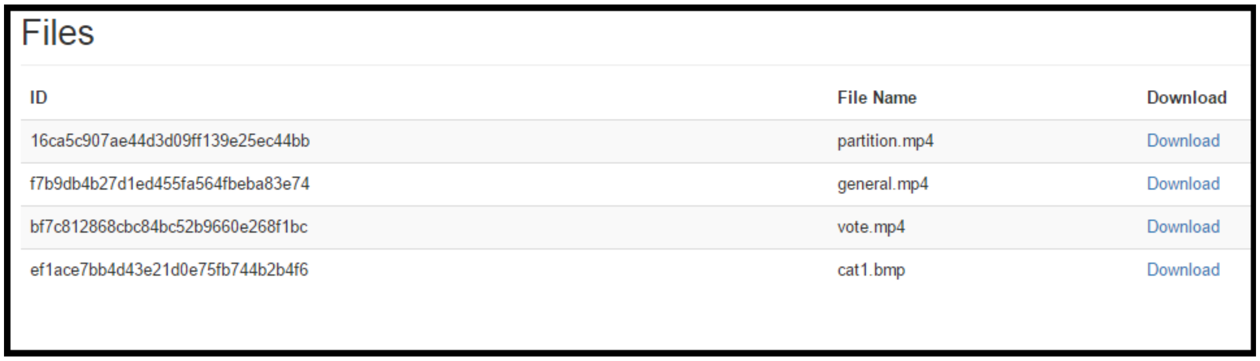
Below, the usage of the command line tool is summarized.

```
usage: ipfs.py [-h] [-s] [-r] [-n HOSTNAME] [-d DIR] [-p PUBLISH]
               [-f GETFILE] [-b GETBLOCK] [-v VERSION] [-x]

An IPFS client.

optional arguments:
  -s, --start           Start the IPFS client
```


The DSFlix P2P Network (Kademlia DHT)

Node	Ip	Blocks	State
0ad63b0dc8b3308dd04edffc8552b004dde9a97e	52.6.78.10	418	ALIVE
0edd6851790ed1f2bf649fb244254b40a6adacc2	52.4.140.31	214	ALIVE
3625a969afddd0fb50ce153b4df4809370002ae1	52.6.233.118	214	ALIVE
4162e7826d56601bde2f24cedf6e6533f2e23c69	52.6.224.242	214	ALIVE
90cf78478d936e34f07325bc91f960cdd1305760	52.6.225.196	98	ALIVE
a1bf84476b149a2c2c53995c48fbed2db9fd1427	52.4.18.203	98	ALIVE
d48a26c83505028007e9a855a49d257dcb090ec2	52.6.221.0	106	ALIVE
d553bef4a58b42a314e565974aad070aef7b748c	52.0.194.231	99	ALIVE
df17ba1db86223096bd3e0448d794aa0325a92f4	52.6.229.14	107	ALIVE
e7dafd4d045defc36268dc06300798e39e974f12	52.6.219.20	104	ALIVE

Figure 4: The network view UI.


Files

ID	File Name	Download
16ca5c907ae44d3d09ff139e25ec44bb	partition.mp4	Download
f7b9db4b27d1ed455fa564fbeba83e74	general.mp4	Download
bf7c812868cbc84bc52b9660e268f1bc	vote.mp4	Download
ef1ace7bb4d43e21d0e75fb744b2b4f6	cat1.bmp	Download

Figure 5: The file lister UI.

```
  -r, --routingtable    Dump the local routing table
  -n HOSTNAME, --hostname HOSTNAME
                        The hostname to use when starting the client.
  -d DIR, --dir DIR     The directory where the client is stored.
  -p PUBLISH, --publish PUBLISH
                        Publish a file from this node.
  -f GETFILE, --getfile GETFILE
                        Download a file from this node.
  -b GETBLOCK, --getblock GETBLOCK
                        Download a block from this node.
  -v VERSION, --version VERSION
                        Download this version of the file.
  -x, --list            List Files
```

3 Design and Implementation

In this section, we'll discuss our implementation of the functional specification.

First, a high-level view of the system. On a given node, the IPFS client runs in an instance of the Erlang runtime. We provide various wrappers written in Python to interact with the Erlang runtime, which in turns communicates with nodes on other machines. The other elements of our user interface are HTML5 web pages. Figure 6 shows the entities in our back-end fit together.

In the sections below, we'll flesh out the implementation details of these components.

3.1 Blocks

Blocks are implemented as JSON documents. The binary content of a block is base64 encoded. The identity of the block is computed by taking the SHA1 hash of this JSON document. Block size is not fixed, but, by default, we break files into 16 KB chunks. Figure 8 shows an example of a JSON document representing a block.

3.2 Files

Recall that files are represented by metadata blocks, containing a list of links to other blocks. Metadata blocks are analagous to i-nodes in a traditional filesystem. In our implementation, we call metadata blocks *root blocks*. Root blocks are also implemented as JSON documents. See Figure 9 for an examples. Just like normal blocks, the root block's identifier is computed by taking the SHA-1 hash of this JSON document.

Conceivably, root blocks could link to other root blocks, but our implementation does not currently support this kind of indirection.

3.2.1 Naming

Our system would be rather hard to use if we didn't provide a way to associate hashes with human readable names. But that's not the case. The real implementation of IPFS would

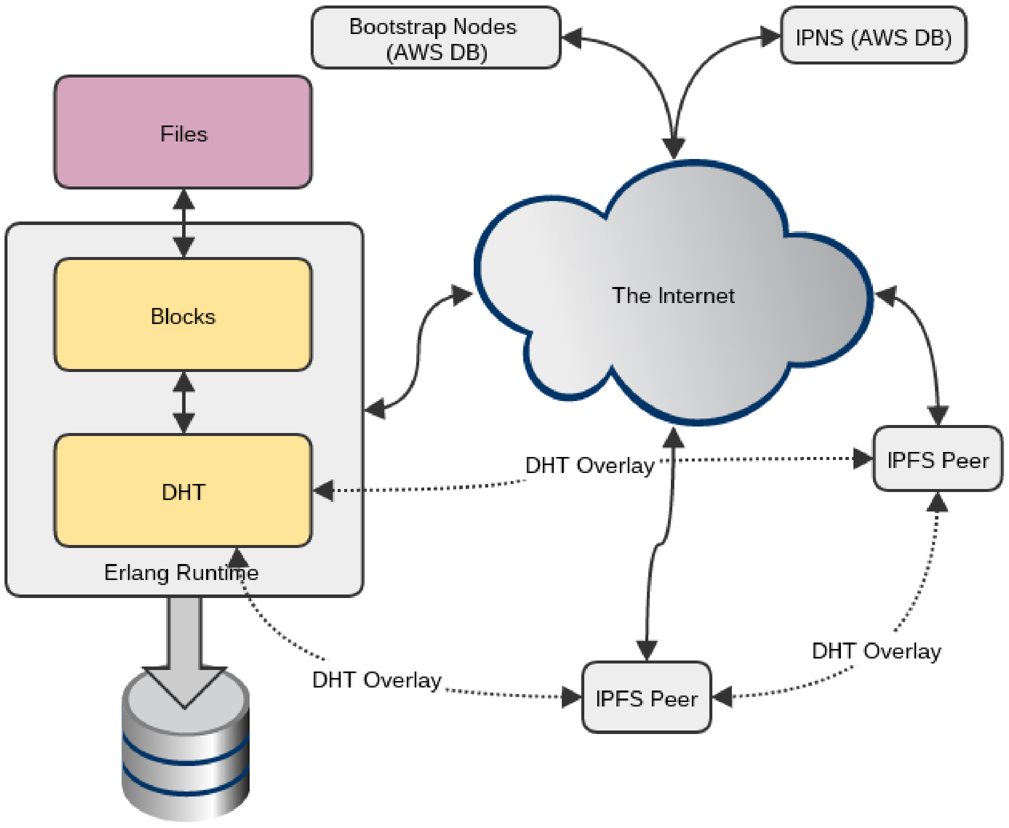


Figure 6: High-level system view

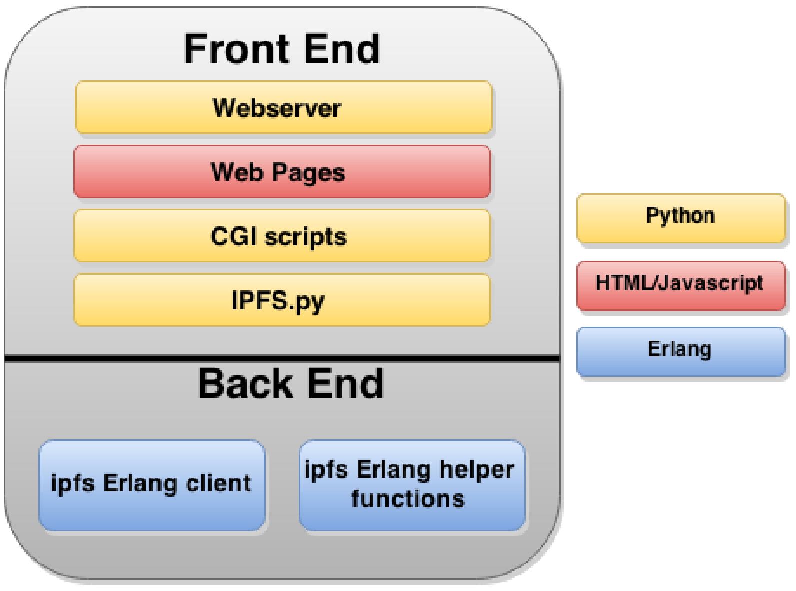


Figure 7: IPFS Modules

```
{"data":"aW1wb3J0IGNvbGxlY3Rpb25zCgpncm..."}
```

Figure 8: IPFS Block

```
{"data": ["blob", "blob"], "links":
[{"hash": "90974818b51cc25b8324f6236957eed656589b72", "size": 21860},
 {"hash": "809896abefcd809347223948288edfabc092938e", "size": 21860}]}
```

Figure 9: IPFS Root block

rely on a highly available naming service, in the style of DNS. To simulate this, we provision an AWS DynamobDB table to store name-to-hash mappings.

3.3 Nodes

3.3.1 Installation

A node is created using the `ipfs.py` script. We first generate an RSA key-pair, then compute the SHA-1 hash of the public key to generate the node id. The network address of the node is specified when launching the script. Next, we create a local directory where the local node will be installed, called the `client directory`. We create a `blocks` folder in the client directory, then save all the node attributes into a configuration file, `.ipfs`.

3.3.2 Bootstrapping

The node bootstrapper is implemented as an AWS DynamoDB table. This table simply stores a list of nodes who have joined the network so far. Nodes can communicate with this table over an HTTP endpoint, to add rows or perform queries.

When a node is installed, it will pull down the full contents of this table, and save the results into a file, `.bootstrap`, in the client directory. The node will also insert themselves into the table.

3.3.3 Launching

Now, the client is ready to be launched. An Erlang runtime is created, and is given a reference to the client directory. Our launch function reads the `.ipfs` and `.bootstrap` files in order to initialized the state of the client. For example, the peer list is seeded with the contents of `.bootstrap`. The last step of the bootstrapping process is executed, by broadcasting a message to the initial peer list, to inform them we have joined the network.

Finally, an Erlang process called `ipfs` is spawned, for receiving and handling messages from peers.

3.3.4 Addressing

The address of an Erlang process has three parts.

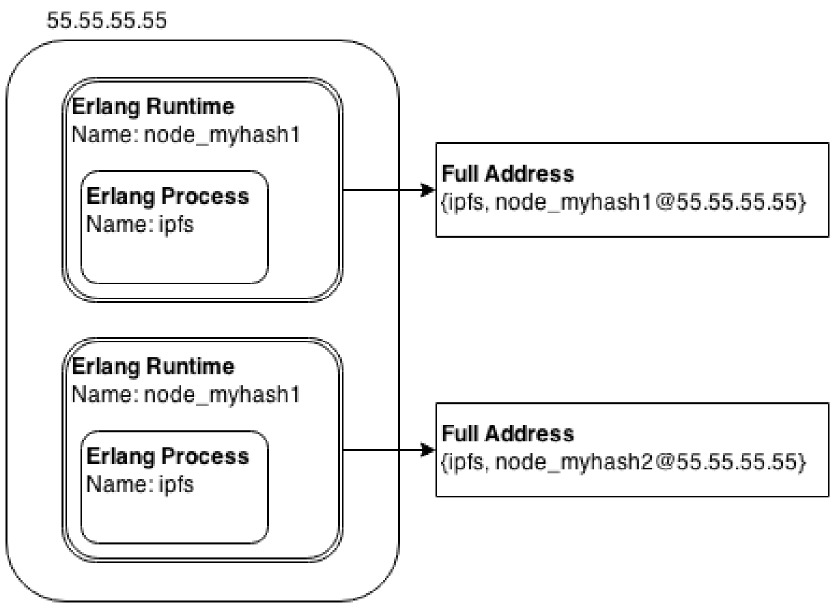


Figure 10: Addressing in Erlang.

1. The name associated with the surrounding runtime, specified at launch.
2. The name of the process, specified when spawned.
3. The network address of the machine.

In our implementation, a node with identifier `NID` will spawn an Erlang process named `ipfs` in a runtime named `node_NID`. See Figure 10 for a high-level view of addressing. Note that port numbers are notably missing from this addressing scheme. Port numbers are transparently assigned by a process called the *Erlang Port Mapper Daemon*, running on each node.

3.3.5 Message Passing

We implement inter-node communication with Erlang's builtin message passing primitives. Two Erlang runtimes can talk to each other over the Internet using these mechanisms. The message handler is summarized below.

```
receive
    {Message, Callback, SourceInfo} ->
        # Insert the sender into the Peer List, if we haven't see it yet
        UpdatedRoutingTable = case SourceInfo of
            nil -> RoutingTable;
            _ -> ipfs_dht:insert(RoutingTable, SourceInfo)
        end,
        case Message of
            {ping} -> # Ping Handler
            {store, Id, Data} -> # Store a block locally
```

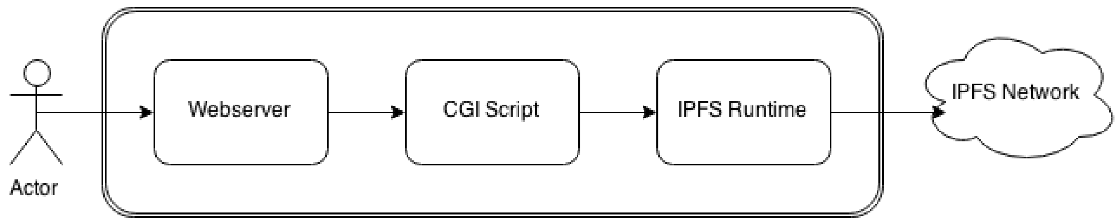


Figure 11: Webserver overview.

```
            {publish_file, MetadataId} -> # Publish a file
            {find_value, Id} -> # Retrieve a block by Id.
          end
end
```

3.4 Front end

Our front end components are implemented as HTML 5 web pages, hosted by a simple web server. We used templates from a service called Bootstrap as the basis for our web pages. We implemented a handful of CGI scripts to interact with the IPFS runtime on the local machine. See Figure 11 for an overview.

4 Distributed Systems Challenges

In this section, we'll discuss the distributed-systems-related challenges addressed in our project.

4.1 Configuration

Nodes in our system require knowledge of their peers. As discussed above, we tackle this problem with a bootstrapping service. This service lives outside of the P2P network, and is expected to be highly available. For our demo, we implemented the bootstrapper with an AWS DynamoDb table, thereby outsourcing the high availability problem. Nodes add themselves to this table at boot time. See Figure 12 for an overview of this system.

4.2 Consistency

A central question in any distributed file system is consistency. How are file updates handled? Let's explore these issues in the context of our file system.

4.2.1 File Updates

Recall that blocks in our system are content-addressed. A consequence of this is that blocks are immutable. A block cannot be modified without also changing it's identity. Therefore, there are no block-level consistency issues to worry about, as clients simply cannot change blocks.

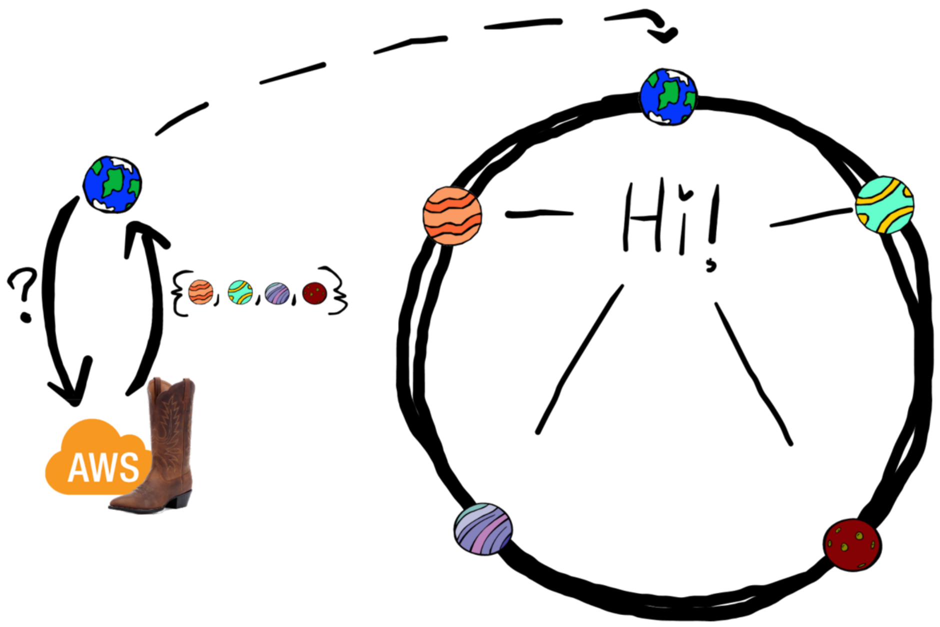


Figure 12:

How, then, does a user modify a file? Very simply: modify the file on the local file system, and publish a new version of the file to IPFS. Only blocks that change require distribution, due to the deduplication inherent in content-addressing. See Figure 13 for an illustration of this.

Consistency issues are therefore pushed to the versioning system. In our implementation, the version service (a component of the name service) is centralized, and thus can handle conflicts due to multiple writers. This is similar to Git: clients who wish to change a file create a set of local changes and submit them to a central authority for approval.

4.3 Consensus

Our project doesn't feature any significant consensus algorithms. The only piece of state that our nodes are expected to agree on is the Peer List. That is, each node is expected to have a complete Peer List. This is accomplished by multicasting a notification to ones peers when joining the network, so the can add you to their list. This is illustrated in Figure 12. We don't account for Byzantine nodes in our current implementation, though this would be a requirement for future work.

4.4 Fault Tolerance

Our network maintains a global replication factor, K. This dictates how many replicas a block is distributed to upon publication. When downloading a block, the list of replicas is computed using the Kademlia distance metric, this list is shuffled, then replicas are tried in

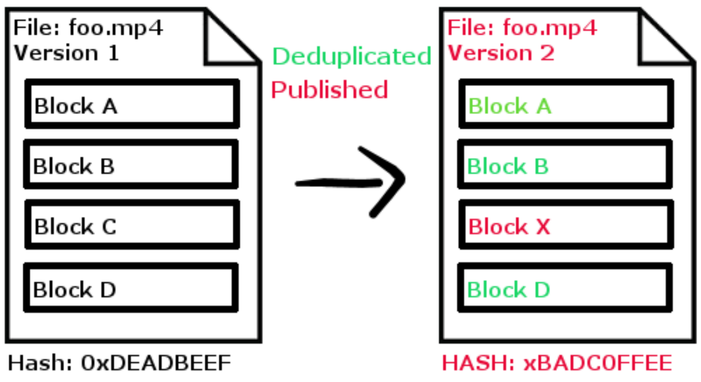


Figure 13: A file update.

turn. The shuffling is a crude form of load balancing. See Figure 14 for an illustration of how a replica failure is handled.

When downloading a file, if a replica fails to respond once, it is blacklisted for the duration of the download, to avoid wasting time on a seemingly failed node.

5 Testing

5.1 Overview

In this section, we'd like to give a brief overview of our test approach. We used both local testing, and testing on AWS virtual machines. I found AWS invaluable as a testing platform, and would like to advocate for giving students AWS credit in future semesters.

5.2 Local Testing

Initially, our tests were conducted on a single machine. This entailed running multiple Erlang runtimes on the same operating system, which seemed to work well enough. We tested with up to 10 runs running locally.

In this scenario, nodes can either be assigned *short names* or *long names*. This is a setting specified when launching the Erlang runtime. If a node is assigned a short name, it can only communicate with nodes on the same subnet, which trivially is true on the same machine. Short names are of the form `process_name@machine_name`. On the other hand, long names are of the form `processname@ip_address`. Long names are required for hosts to communicate over the Internet, but also suffice for local testing.

5.3 Testing on AWS

After verifying that our system worked locally, we wanted to deploy it to a real network of nodes. This seemed a logical step for a distributed systems project. At first it was unclear how to proceed. The cluster machines at our university did not have Erlang installed,

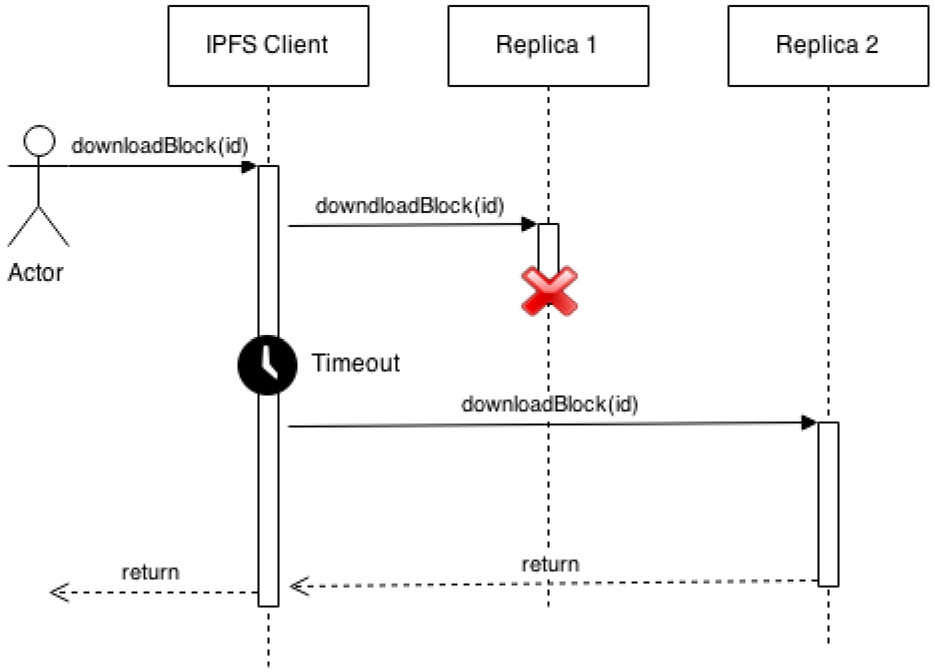


Figure 14: Block replication in action.

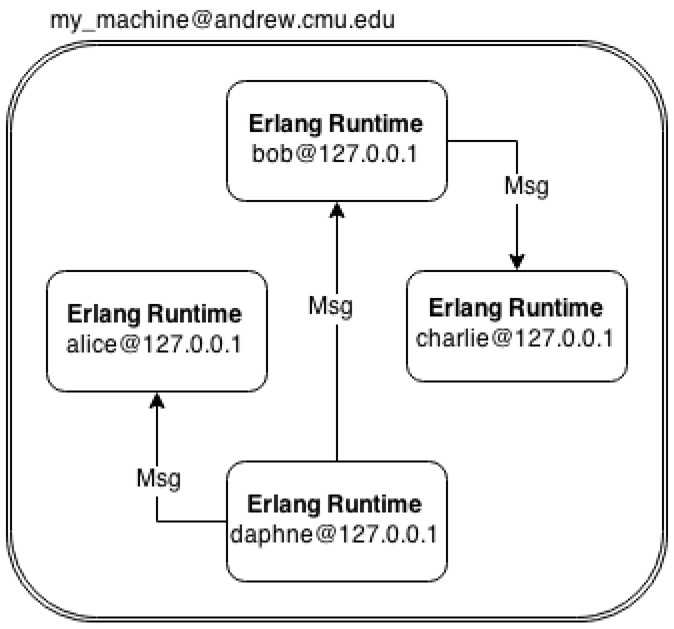


Figure 15: Testing on one machine, in Erlang.

and though we could use our laptops, this was hardly a robust solution, unless we always programmed together. Thus, we decided to use virtual machines from the Amazon Web Service (AWS), in particular using Elastic Cloud Compute (EC2).

The first step of this process was to configure an Amazon Machine Image (AMI) with the necessary software to run our code. This included Erlang, Python, and various other libraries. We also created a *security group*, configured to allow Erlang communication. You can think of a security group as a firewall configuration. Next, we configured our machine image so that associated VM's could pull from our Git repository. Thus, to deploy our latest code, we simply needed to:

1. Create a virtual machine based on our AMI template.
2. Associate it with our security group.
3. Remotely issue commands to pull our repository, build the source, and launch the client.

Since VM creation is programmable, this effort culminated in a script that allowed us to quickly deploy networks of a chosen size, e.g.:

```
python start_nodes.py --count 10
```

If a bug was found during testing, a fix could be pushed to the Git repository, and the existing nodes restarted by running the following.

```
python start_nodes.py --nodes IP1 IP2 IP3 ... IP10
```

This made testing a piece of cake! To reduce costs, we used very cheap virtual machines, called t1.micros. Overall, I estimate we spent less than 20 dollars.

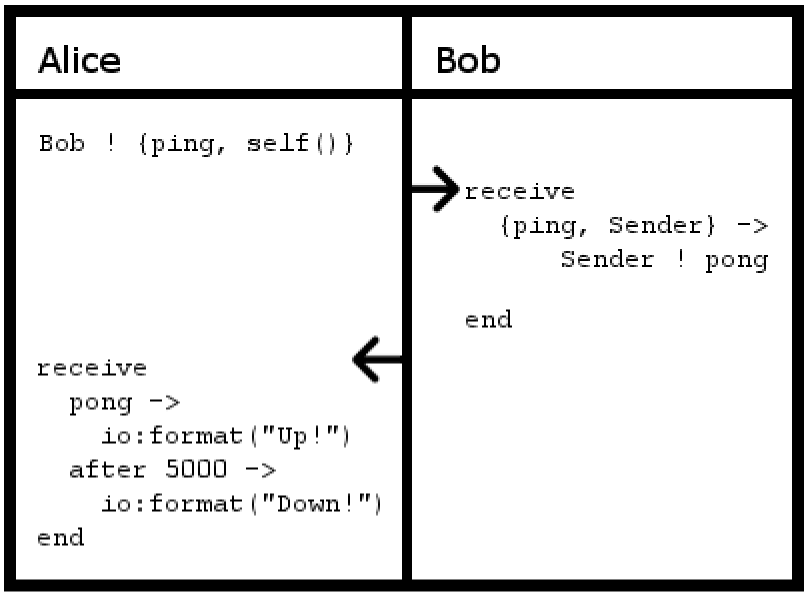
5.4 Two Cents

I think teams should be granted small AWS credits for the 18-842 project, if they desire them (no more than fifty dollars). It gives students a great platform to test their projects. Testing locally, on a handful of laptops, or even on cluster machines is not sufficient for certain projects. I will leave this as a follow-up item, however.

6 Skills Challenge

6.1 Overview

For our skills challenge, we implemented the back end of our service with the Erlang programming language. Erlang was designed by researchers at the Ericsson corporation in the late 80's, with the intention of being used to write software for telephone switches.



```
Bob ! {ping, self()}
```

```
receive
  {ping, Sender} ->
      Sender ! pong

end
```

```
receive
  pong ->
    io:format("Up!")
  after 5000 ->
    io:format("Down!")
end
```

Figure 16: Message passing in Erlang.

6.2 The Good

Given that Erlang was designed to facilitate telephony, it's no surprise that communication is a seamless part of the language. Message passing primitives are built into the core language; see Figure 16 for an example. The `!` operator can be used to send a message to another Erlang node, who can handle the message with a `receive` block, and reply. Various errors, such as timeouts, can be handled easily by different clauses of the `receive` block.

Communicating with nodes is location transparent. I can be running a hundred nodes on my local machine, or spread them out across the Internet, and the code for communicating is the same, modulo some configuration issues (such as firewalls). This makes testing large networks of nodes easy.

While, as we'll discuss below, Erlang has its share of idiosyncrasies, certain features of the language allow for rather concise, elegant constructions. For example, Erlang supports *list comprehensions*, a feature for processing collections of objects, which came in handy at various points during development.

```
[X || X <- [1,2,3,4,5,6], X > 3].
Result: [4,5,6]
```

6.3 The Bad

Erlang is well suited for certain distributed application, but there are caveats, especially in the context of a short project. We only had a few months to move from our proposal to our final implementation, and adding the requirement of a novel language only increased the difficultly level. We were not unique in this, as many teams also used unfamiliar languages in their implementations. However, most teams chose Go or Scala. I believe that Erlang is a significantly more difficult language to work with.

First, Erlang is, for most intents and purposes, a purely functional language. Our team had little experience with this paradigm, and it proved difficult to adjust. One major issue: all data structures in Erlang are immutable. Consider, for example, a dictionary. In Python or Java, you can simply add mappings to a dictionary. In Erlang, the same operation entails copying the dictionary, and appending the new mapping to the end. After a while, one gets used to working with this way of thinking, but, coming from an imperative background, an edge of discomfort never fades.

Next, virtually everything in Erlang is expressed recursively. Certain language constructions, such as the aforementioned list comprehensions, can hide this fact, but in general recursive functions cannot be avoided. Consider, for example, a server thread listening for incoming connections. Most programmers likely conjure the following in their mind for this problem.

```
while(true) {
    client = socket.listen()
    request = client.read()
    resp = process(request)
    client.write(resp)
}
```

In Erlang, this code becomes the following.

```
listen() ->
    receive ->
        {Request, Client} ->
            Response = Process(Request),
            Client ! Response
    end,
    listen().
```

Any computer science graduate should be suspicious of such code, as it seems prone to a stack overflow. However, the Erlang language guarantees *tail call elimination*. In brief, recursive functions where the recursive call is the last statement (a so-called tail call) will be rewritten to resemble iterative routines by the compiler, thus preventing the apparent stack overflow.

Despite this guarantee, one can still shoot themselves in the foot, e.g.:

```
listen() ->
    receive ->
        {Request, Client} ->
            Response = Process(Request),
            Client ! Response
    end,
    listen(),
    io:format("Some log message.").
```

This programmer innocently decided to put in a logging message, perhaps intending to record some information in a failure case. What they've actually done is eliminate the possibility of tail call elimination, and introduced the real possibility of blowing the stack.

Next, to touch on an issue unrelated to functional programming. The standard library of Erlang is vastly less capable compared to that of Python or Java. Moreover, there are far fewer third party libraries for Erlang. For example, where Python and Java have multiple options for parsing JSON, Erlang only has a few scattered, incomplete options. This is likely a popularity problem. Way more people use Java and Python, and this motivates the development of a complete set of robust libraries. Far fewer people use Erlang, and library coverage suffers.

Next, a rather significant issue: the message passing mechanisms in Erlang, while easy to program with, are simply not suited (or designed) for a distributed application over the open Internet. Rather, they are engineered for an application running in a dedicated datacenter or cluster, where one has complete control over the configuration of the nodes. Making abitrary Erlang nodes talk to each other over the Internet is nothing short of a nightmare, due to issues with firewalls, NATs, etc. While one can fall back on conventional communication protocols, like TCP or UDP, much of the magic of Erlang is lost.

Another thing to note: the abstractness of Erlang's message passing comes at a cost. Namely, one has no control over what happens at the transport layer. Distributed Erlang nodes communicate over TCP, using a protocol that may be far more sophisticated and expensive than your application requires.

6.4 The Verdict

Groups considering Erlang should give the matter serious thought. If they have limited experience with functional programming, they may want to consider another option, like Scala or Go. If they want their application to work easily on the open Internet, they definitely should look elsewhere. On the other hand, if their application is designed to run in a more constrained environment, such as a cluster, using Erlang could definitely pay off. In that setting, the message passing primitives really shine.

If a team truly wishes to commit to Erlang, it would be best to consider using it in combination with other languages. Erlang is best suited for back end applications. It's a waste of time to try and leverage it for user interfaces. Use something like Python instead, as a thin wrapper around the Erlang runtime. This will make your life considerably easier.

My personal opinion: while I enjoyed struggling with Erlang for this project, I think teams will be better served, professionally, by toying with more popular languages. Erlang experience may raise a few eyebrows; but ultimately, the good parts of Erlang have long been incorporated into other languages—other languages which also have extensive libraries and other nice features. Save yourself the frustration, unless you enjoy that sort of thing.

```
Be not afeard. Erlang is full of noises,
Sounds, and sweet airs that give delight and hurt not.
Sometimes a thousand twangling message passing primitives
Will hum about mine ears, and sometimes list comprehensions,
That, if I then had waked after long sleep,
Will make me sleep again. And then, in dreaming,
The clouds methought would open and show fault tolerance
Ready to drop upon me, that when I waked
I cried to dream again.
---Shakespeare, on Erlang
```

7 Team Organization

7.1 Collaboration

Our team held weekly meetings over Google hangouts, meeting in person when working on major deadlines, such as the first video, first demo, and final presentation. Hangouts is useful as it maintains all chat history, and can be used both for formal meetings and for informal communication throughout the week. We arranged for two weekly meeting times, on Tuesday and Thursday. We determined on Tuesday if we needed the second meeting.

The rough assignment of tasks was follows..

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

All members contributed to programming and assisted with design as needed.

7.2 Schedule

Though we didn't finish every deliverable, we made healthy progress through the duration of the progress. Our initial proposal was rather ambitious, and we were able to get a decent chunk of it working. By the midway demo, we had our system working on a single machine (multiple processes), and had started working on our front end. By the final presentation, we had our system deployed on multiple Amazon virtual machines, and a much nicer front end.

7.3 Source Control

We used a private Github repository to manage our source code. We had a master branch and a development branch. We typically worked in the development branch, then pushed mature changes to the master branch.

Github proved very useful for our source control needs. It was also useful for the test space: we were able to configure Amazon VMS to clone our repository, making deployment of our code to test machines very simple.

8 Future Work

In this section, we'll discuss potential future improvements to our project.

8.1 Scalability

To make our system truly inter-planetary, we first need to address scalability. Our bootstrapping and routing algorithm need the most attention in this regard. At the moment, we assume that node has a complete picture of the network. Clearly, this doesn't scale to a global scale, let alone a universal one. Fortunately, we have enough infrastructure to easily implement a more sophisticated routing algorithm, such as that used by Kademlia. In particular, we could implement the *lookup-node* operation from Kademlia. This would improve both our bootstrapping and block location procedures.

8.2 Runtime Block Republishing

We discussed a few mechanisms for republishing blocks at runtime. This feature is necessary to seeds blocks onto nodes joining the network after a file is added. One option here is to use a similar feature from Kademlia, which calls for republishing objects on a regular schedule (once per hours, for example). Another option: if a request is made to a node for a block it doesn't have, the node locates the block and stores it locally, so future requests will be satisfied.

8.3 Security

Currently, nodes generate a public/private key pair, but they aren't used for anything. These keys could be used for both message signing and encryption, per usual. However, all the usual issues with asymmetric cryptosystem would need to be addressed, such as building a robust PKI.

8.4 Fairness

Nodes can opt out of delivering blocks to peers with impunity. We could incorporate concepts from BitTorrent to punish freeloaders. This would require maintaining more information at each node about block exchange (who do we send blocks to, and who do we receive blocks from).

8.5 Git-Style Versioning

Our file versioning method is rather simple. The original IPFS proposal calls for a versioning mechanism very similar to that of Git. Out of all tasks, this would require the most work.

References

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

[2] Linus Torvalds and Junio Hamano. Git: Fast version control system. *URL http://git-scm.com*, 2010.

[3] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.

[4] Ashok Anand, Fahad Dogar, Dongsu Han, Boyan Li, Hyeontaek Lim, Michel Machado, Wenfei Wu, Aditya Akella, David G Andersen, John W Byers, et al. Xia: An architecture for an evolvable and trustworthy internet. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 2. ACM, 2011.

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