IPFS: InterPlanetary File System

The Distributed, Permanent Web

Project Report

Akash Trehan - 150050031

EE 465: Cryptocurrency And Blockchain Technologies
IIT Bombay
November 9, 2018

Contents

Chapter 1

Introduction

The InterPlanetary File System (IPFS)[1] is a peer-to-peer distributed file system that aims to upgrade how the web works currently. It combines various research ideas to create a web which is distributed, safer, smarter, faster and permanent.

The current HTTP protocol for sharing data over the web does not take advantage of the various brilliant file distribution techniques that have emerged in the previous few decades. HTTP powers a large portion of the internet today, making it almost impossible to replace. IPFS wants to achieve this change in a way that the end users do not have to change how they use the web. A lot of engineering effort is being put in to gain wide acceptance and fit into the existing infrastructure as smoothly as possible.

Chapter 2

Why IPFS?

2.1 Distributed

The structure of network decided what kind of applications can be built upon it, what protocols can be used and how much power users have on their own data.

A centralised system allows changing of content quickly but all of the power is present at one location. The end users do not have the same capabilities or control over the data. This also leads to a single point of failure.

A decentralised system divides the duties of the central node to several nodes to which end users connect, thus increasing the resiliency of the network. But this is still less resilient than a completely distributed systems can be.

IPFS for a distributed network, which means that all of the nodes within the system contribute almost equally to the ecosystem. It is a peer-to-peer systems and hence everyone speaks the same protocol. This kind of system is extremely resilient to failures.

In this age where data is a commodity, a distributed system prevents a single entity from controlling a large portion of the data.

2.2 Fast

The current web is location addressed. Our URLs resolve to IP addresses of servers which can be thousands of miles away from our own computers, which is where we fetch our data from. This is the case even when the person sitting next to us has the same data on his machine. IPFS solves this problem by making the web content addressable. This implies that it doesn't matter where the data comes from as long as it is the exact data that we requested. A distributed structure allows data to be present at locations closer to our computers thus reducing communication times and making the web faster.

2.3 Offline First

Think of an example where you are collaborating on a Google Doc with five other friends sitting with you in the same room and suddenly google.com goes down. Now you cannot continue collaborating since all of your computers were talking to google servers. Being a distributed protocol, IPFS allows resiliency to such failures since nodes can directly communicate with one another. It is important to realise here that connectivity is not binary. You can still be connected to other machines in a local area network but be disconnected from the internet. IPFS increases the possibilities even in the scenario of reduced connectivity.

Chapter 3

High Level Overview / Usage

Every node in the IPFS ecosystem has a NodeId. It can upload data to the system and retrieve data. IPFS has both a command-line interface as well as a web UI.

Adding directory to IPFS is as simple as:

```
ipfs add -r /path/to/dir/
```

This recursively adds all the files inside the directory to IPFS and returns an IPFS address for each of them.

To retrieve content, the user needs its address. It can either put the address in the Web UI or use for following terminal command:

```
ipfs cat /ipfs/<content_address>
```

Let us now look at what's happening under the hood.

Chapter 4

The IPFS Stack

The IPFS stack has 6 layers as follows (bottom-up):

1. Network
2. Routing
3. Exchange
4. MerkleDAG
5. Naming
6. Application

The MerkleDAG layer is the most important out of these. It forms a thin waist model where layers above and below it can be developed independent of each other i.e the protocols in the bottom layers can change without the upper layers knowing. Due to this independence, the layers of IPFS are being built as separate projects.

The network, routing and exchange layers are being built under a project named *libp2p*[4].

The merkledag layer is being built as a project named the *InterPlanetary Linked Data (IPLD)*[3].

The naming layer is being built as a project named the *InterPlanetary Naming System (IPNS)*.

4.1 libp2p

4.1.1 Network Layer

This is the lowest layer and is built mostly from existing networks protocol for reliable transfer of data.

4.1.2 Routing Layer

The routing layer is used when we need to find which peers contain data corresponding to a particular hash value on IPFS. This requires the use of distributed hash tables.

A distributed hash table (DHT) is a lookup service similar to hash tables i.e. they store (key, value) pairs and any node in the system can efficiently give a value corresponding to any key by interacting with other nodes in the system. They differ from hash tables in that none of the nodes store the entire hash table, but only a small portion of it. The protocols for who stores which key and what is stored as a value varies between different DHTs.

IPFS used ideas from the Kademlia DHT[5] and the Coral DHT[2].

Now, DHTs are an important component of p2p systems. Let's see how these systems work:

1. Each node which is a part of the p2p system is given a NodeID.
2. Kademlia stores the values in nodes which are nearest to the key. The distance is defined as the XOR of the key value and the NodeID.
3. Every node also stores the addresses of nodes at exponentially increasing distance from itself i.e. node at distance 2^0, 2^1, 2^2 and so on. Thus it stores addresses to at most $log_2(n)$ nodes where n is the total number of nodes in the system. Lets call these the neighbours of a node.
4. When a node needs to find the value corresponding to a certain key, it checks which nodeID, say *nid*, would be closest to the particular key and then looks for the node with nodeID *nid*.
5. If the node it is looking for is one of its neighbours, it is done and can communicate directly with it to get the value corresponding to the key.
6. If not, then it find the neighbour nearest to the node with nodeID *nid* (can be again done using XOR distance). We'll call this neighbour n
7. The node asks neighbour n if it knows the address of node with nodeID *nid*.
8. If n knows the address, it returns it, otherwise returns the address of it's own neighbour closest to nodeID *nid*, say $n2$
9. Now the node makes a query to $n2$ and the process continues. This iterative process is similar to what happens in iterative DNS.
10. When the correct node is finally found, a query is made to it to retrieve the value corresponding to the key.

Using the above protocol, Kademlia does an efficient lookup through massive networks by querying a maximum of $log_2(n)$ nodes. S/Kademlia also provides resistance to Sybil attacks and a low coordination overhead. It is used in systems like Gnutella and BitTorrent.

S/Kademlia[7] also extends Kademlia to lookup values over disjoint paths so that even in the presence of a lot of adversaries, honest nodes can connect to each other.

IPFS also uses ideas from Coral DSHT (Distributed Sloppy Hash Table) to prevent hot-spots (overcrowding of all the nearest nodes when a key become popular). Coral also prevents "near" nodes from storing the data they don't need/want when "far" nodes are already storing it. To do this, the DHT stores the address of nodes containing values of a particular key rather than the value itself. Thus when the correct node is found using the above algorithm, it returns the address of all the other nodes which store that value. A query can now be made to one of these nodes to retrieve the value.

In IPFS, nodes are identified by a NodeId, the cryptographic hash of a public-key, created with the static crypto puz- zle used in S/Kademlia. The identity generation uses the following method:

```
difficulty = <integer parameter>
n = Node{}
do {
    n.PubKey, n.PrivKey = PKI.genKeyPair()
    n.NodeId = hash(n.PubKey)
    p = count_preceding_zero_bits(hash(n.NodeId))
} while (p < difficulty)
```

The interface used by IPFS's DHT is as follows:

```
type IPFSRouting interface {
    FindPeer(node NodeId)
    // gets a particular peer's network address
    SetValue(key []bytes, value []bytes)
    // stores a small metadata value in DHT
    GetValue(key []bytes)
    // retrieves small metadata value from DHT
    ProvideValue(key Multihash)
    // announces this node can serve a large value
    FindValuePeers(key Multihash, min int)
    // gets a number of peers serving a large value
}
```

4.1.3 Exchange Layer

This layer is built to achieve interchange of data blocks between the nodes in the p2p system. For this, IPFS takes inspiration from BitTorrent[8].

BitTorrent has some very useful features like sending the rarest pieces of data first so the non-seed peers can trade with each other. It also rewards nodes who share data and contributes to the system while punishing ones which only leech resources.

Taking the above ideas, IPFS built it's own protocol known as BitSwap. The nodes have a *want_list* and a *have_list*. The *want_list* stores which blocks the nodes is looking to acquire and the *have_list* store the blocks that it is willing to share. BitSwap forms a kind of credit system to incentivize nodes to seed even when they do not need anything in particular. This credit score also incentivizes the nodes to find, cache and distribute rare pieces of data even if doesn't want them itself.

BitSwap Credit

In BitSwap, nodes sending and receiving data from other nodes track their balance with the other node. This causes nodes to share data in the present and hoping to have the debt repaid in the future when they need data.

The data is sent with a certain probability which depends on how much debt the receiver has with that sender. As the debt increases, the probability of receiving data decreases. After every denied request the peer is ignored for an *ignore_cooldown* timeout to prevent the peer from gaming the probability by just sending a lot of requests (more dice-rolls).

BitSwap Strategy

BitSwap aims to be harsh to leechers and lenient to trusted peers even when they are temporarily unable to share data. The current algorithm:

1. The node calculates a value named *debt ratio r* between a node and it's peer.

$$r = \frac{bytes\_sent(inpast)}{bytes\_recv(inpast)+1}$$

2. Given the *debt ratio r*, the probability of sending to a debter is given as:

$$P(\text{send}|r) = 1 - \frac{1}{1+\exp 6-3r}$$

A lower debt ratio means high probability of getting the data.
This method has certain other advantages as well:

1. It prevents Sybil attacks since newly created nodes would all have shared no data in the past
2. It protect nodes who have previously shared a lot of data but are temporarily unable to do so

BitSwap Ledger

The data needed to calculate the debt ratio needs to be stored somewhere. Each node stores a *Ledger* to keep track of exchange with all the other nodes.

The *Ledger* structure looks as follows:

```
type Ledger struct {
    owner NodeId
    partner NodeId
    bytes_sent int
    bytes_recv int
    timestamp Timestamp
}
```

This ledger is sent to the peer while requesting information. If the data inside the ledger does not exactly match what the peer has, the ledger is re-initialized to zero. A node may try to lose the debt by sending a random ledger, but they would also lose the acquired credit so there is a trade-of.

BitSwap Interface

The BitSwap protocol has the following interface:

```
// Protocol interface:
interface Peer {
    open (nodeid :NodeId, ledger :Ledger);
    send_want_list (want_list :WantList);
    send_block (block :Block) -> (complete :Bool);
    close (final :Bool);
}
```

1. open (nodeid :NodeId, ledger :Ledger) : This function is used to open a connection with a peer a node wants data from. The node send the ledger corresponding to that peer in the request. The peer can choose to accept or reject the connection based on the debt of the node or the values in the ledger.
2. send_wan_list (want_list :WantList) : After the connections has opened, the node sends its want list to the peer. The peer checks if it has any of the wanted blocks and then sends it according to the BitSwap strategy.

3. send_block (block :Block) : This is used by the peer to send the requested block to the node. After the node has received all the data, the node calculates the multihash checksum to see it if indeed received the correct data. If the block is accepted it moves from the want_list to have_list. The ledgers are updated accordingly.

4. close (final :Bool) : This is used to close the connection after the block has been received from the peer (or in case of a timeout).

4.2 IPLD - Creating the merkle forest

The InterPlanetary Linked Data (IPLD) is at the core of IPFS. It emerges as a direct result of trying to make the web content-addressable. It allows the treatment of all hash-linked data structures under a single category. Thus any data model which links data using hashes can be treated as an instance of IPLD. Example of technologies using such data structures are git, Bitcoin, Ethereum and now IPFS. With IPLD, links can be traversed across protocols boundaries.

4.2.1 MerkleDAG

IPLD is based on the concept of MerkleDAGs, which are directed acyclic graphs where links between objects are cryptographic hashes of the targets embedded in the sources. It is inspired from the Git data structures but does make some modifications to it.

This allows for many useful properties like:

1. **Integrity**: The hashes act as checksums to make sure that the data you get is the one you requested

2. **Deduplication**: Different files can have certain blocks of data which are same. Since these blocks have the same hash, they only need to be stored once.

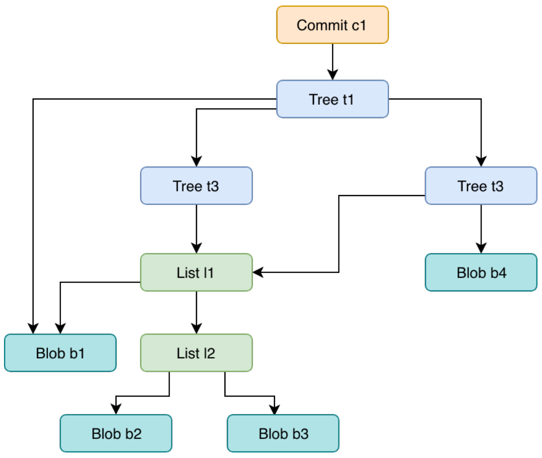
3. **Content Addressibility**: Since the hashes are cryptographic, all content can be uniquely identified by their hash

4.3 Versioned File System

IPFS defines a set of objects, inspired from git, to allow a versioned file system on top of the MerkleDAG. These objects are as follows:

1. **blob**: A blob is just raw data which is content-addressible.It does not link to anything else.

Figure 4.1: Sample object graph



```
{
    "data": "some data here",
    // blobs have no links
}
```

2. **list**: Lists are not present in git and is something that IPFS has introduced. It is used to represent a large or deduplicated file which has been broken into various pieces. Lists can have other lists or blobs inside them. The order in which the blobs or lists occur is important. This also allows in-file deduplication i.e. when blobs in the file have the exact same data. In addition to links, a list also stores the size of content referred by the link. This helps in making size calculations easier.

```
{
    "data": ["blob", "list", "blob"],
    // lists have an array of object types as data
    "links": [
    { "hash": "QmcCcJMotPnNS8dUbybG7vH32u27smnCfrYWWhazyKMT18",
    "size": 9458 },
    { "hash": "QmRjRnx15pXLKbXz3y62wsJarDRGdrhDbu3AZUQmhqgugh",
    "size": 19441 },
    { "hash": "QmRy2xuqAgJxThesKFBMosAne6rxPKbmYNYkYJZwnsXgiE",
    "size": 5286 }
    // lists have no names in links
    ]
}
```

3. **tree**: Trees are used to map names to hashes. They represent directories.

They can link to blobs, lists, other trees and commits. The names help in path resolution.

```
{
    "data": ["blob", "list", "blob"],
    // lists have an array of object types as data
    "links": [
    { "hash": "QmcCcJMotPnNS8dUbybG7vH32u27smnCfrYWWhazyKMT18",
    "name": "hello", "size": 9458 },
    { "hash": "QmRjRnx15pXLKbXz3y62wsJarDRGdrhDbu3AZUQmhqgugh",
    "name": "demo", "size": 19441 },
    { "hash": "QmRy2xuqAgJxThesKFBMosAne6rxPKbmYNYkYJZwnsXgiE",
    "name": "world", "size": 5286 }
    // trees do have names for links
    ]
}
```

4. **commit**: A commit represents the version history of an object. It links to previous commit as well as the objects in the new commit. It also stores additional information like commit message, link to author etc.

```
{
    "data": {
        "type": "tree",
        "date": "2018-11-2 12:20:00Z",
        "message": "This is a commit message."
    },
    "links": [
    { "hash": "QmcCcJMotPnNS8dUbybG7vH32u27smnCfrYWWhazyKMT18",
    "name": "parent", "size": 21309 },
    { "hash": "QmRjRnx15pXLKbXz3y62wsJarDRGdrhDbu3AZUQmhqgugh",
    "name": "object", "size": 4568 },
    { "hash": "QmRy2xuqAgJxThesKFBMosAne6rxPKbmYNYkYJZwnsXgiE",
    "name": "author", "size": 120 }
    ]
}
```

4.3.1 Content Addressing

The way in which the above objects are defined makes it very easy to address a particular object as well as traverse through the merkledag.

The path format for IPFS is as follow:

```
/ipfs/<hash-of-object>/<name-path-to-object>
```

The hash of the object retrieves the object and then the links inside the objects are followed according to the name-path mentioned. This allow for multiple ways to access a file. For example, given three objects in path <foo>/bar/baz, the last object is accessible by all:

```
/ipfs/<hash-of-foo>/bar/baz
/ipfs/<hash-of-bar>/baz
/ipfs/<hash-of-baz>
```

4.3.2 Splitting files

When a file is being added to IPFS, it needs to be broken into blobs and lists. Finding the optimal way to break the file is a hard problems and depends on many factors including what all blocks are already present in the system. IPFS allows users to define their own block splitting methods specific to files.

4.3.3 The Merkle Forest

IPFS is like a forest of linked merkleDAGs since it allows systems like ethereum, bitcoin, git to interoperate. Some sample uses of this could be:

1. Referencing a commit in bitcoin to timestamp it - Using IPLD would allow us to transparently traverse from the bitcoin transaction into the files present in the commit.
2. Store Ethereum media on IPFS: Using IPLD would allow seamless jumping from the Ethereum contract or transaction to the media.

4.3.4 Multiformats

Multiformats is a project to create self-describing values which means that looking at the value helps you understand how it was obtained and how you can process it. It includes multiple subprojects namely multihash, multicodec, multibase, multiaddr, multikey and multistream.

Lets take the example of hashes. Based on which has we are using, the same content can hash to different values. Now, only by looking at the hash value, you cannot know which hash function was used. That is where multihashes come in. Multihashes have a hash function code and hash length appended to them at the beginning. The format is as follows:

```
<fn-code><hash length><hash>
```

This is the reason why, by default, IPFS hashes start with *Qm*.

4.4 IPNS - Naming the data

As we discussed immutable paths provide various benefits to IPFS like integrity checking and indefinite caching (If you know an object is not going to change it's cache need not expire ever).

But this implies that if an object is changed, then the new IPFS link needs to be sent to people who want to access it. Thus we need some way to retrieve mutable state at the same path. This is where InterPlanetary Naming System (IPNS) comes in.

4.4.1 Self-certified Names

IPFS uses the naming scheme from SFS [6] to construct a mutable namespace for every node. These names are self-certified, which means that the information used to retrieve the actual data is signed by the private key of the node. This provides a guarantee that the data indeed belongs to that node.

In IPFS, $NodeID = hash(node.PubKey)$ and each user gets a mutable namespace at

```
/ipns/<NodeID>
```

Now since everything in IPFS is content addressed, we need a way to go from an IPNS address to an actual IPFS address. The Routing system comes to rescue here.

The first step is to publish the object in a regular manner and get it's IPFS address. Then we publish this hash on the routing system as a metadata value with the NodeId as the key.

```
routing.setValue(NodeId, <ns-object-hash>)
// Here NodeId is the IPNS address of the object
```

This mapping can be changed by the private key holder in the future, thus allowing the same name to point to different things at different time.

4.4.2 Human Readable Names

We can see that IPNS names are hard to remember and not very user friendly.

One of the ways in which IPFS solves this problem (especially useful for websites hosted on IPFS) is to allow ipfs and ipns address in the DNS TXT records.

An example of a TXT record in my own website is as follow:

```
ee465.akashtrehan.com.  1799    IN  TXT
"dnslink=/ipns/QmWBPZZ59ZDgigAthNEa2sjuiDTKbRdaBjVYDH4tfZHdHS"
```

On requesting the location,

```
/ipns/ee465.akashtrehan.com
```

IPFS first looks up the TXT DNS records for *ee465.akashtrehan.com*, then looks for the record with a *dnslink*.
It then "redirects" to that address mentioned in the dnslink. In this case

```
/ipns/ee465.akashtrehan.com
```

resolves to

```
/ipns/QmWBPZZ59ZDgigAthNEa2sjuiDTKbRdaBjVYDH4tfZHdHS
```

which further resolves to the IPFS address

```
/ipfs/QmYz3vZGv3gi2AYNptfYM3DxxrqkE5gbSxq1tKtiGS6DJk
```

The content corresponding to this IPFS address is finally retrieved by the block exchange system.

Chapter 5

Possible Problems

5.1 Is IPFS really permanent?

Although IPFS promises a web where the content is permanent and information cannot be destroyed, it hardly does that. This claim assumes that all of the data is being shared and pinned by lots of people. Before filecoin is in action I don't see any reason why that would happen. It is possible that I upload some content, a few people access it (but none of them pins it), then I delete the content from my node and a few hours later it's gone from IPFS. Currently the IPFS team have their own nodes which look for newly uploaded data continuously and pin it. But this cannot be expected of a rational node in the system.

5.2 What about pirated data?

Another problem is that if the content is permanent, then there is no way to remove pirated data or other illegal data from IPFS once it's uploaded. This might create a legal barrier for global acceptance of IPFS as a backbone of the web. There have been some discussions about creating blocklists which people can subscribe to and the files mentioned in those blocklists won't be ever downloaded on their machine. But who will maintain these lists? Who makes sure legit data isn't added to those list?

Chapter 6

Conclusion

IPFS has used a bunch of great ideas from decades of academic research and open source projects. Ultimately, it is designed to change the web as we know it. It aims to fix some of the major problems with HTTP, one of the most successful protocols ever.

IPFS aims for a completely distributed web where the users can trust the content without trusting their peers. At a bare minimum, it can be used as a global distributed file system. At its best, it could lead us to a faster, more secure, uncensored and permanent web.

Bibliography

[1] 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`.

[2] Michael J. Freedman, Eric Freudenthal, and David Mazières. "Democratizing Content Publication with Coral". In: *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*. NSDI'04. San Francisco, California: USENIX Association, 2004, pp. 18–18. URL: `http://dl.acm.org/citation.cfm?id=1251175.1251193`.

[3] *IPLD: A data model for interoperable protocols*. `https://ipld.io/`.

[4] *libp2p: A modular network stack*. `https://libp2p.io/`.

[5] Petar Maymounkov and David Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. IPTPS '01. London, UK, UK: Springer-Verlag, 2002, pp. 53–65. ISBN: 3-540-44179-4. URL: `http://dl.acm.org/citation.cfm?id=646334.687801`.

[6] David Mazieres. "Self-certifying File System". AAI0802720. PhD thesis. Cambridge, MA, USA, 2000.

[7] Riccardo Pecori. "S-Kademlia". In: *Comput. Netw.* 94.C (Jan. 2016), pp. 205–218. ISSN: 1389-1286. DOI: `10.1016/j.comnet.2015.11.010`. URL: `https://doi.org/10.1016/j.comnet.2015.11.010`.

[8] Johan Pouwelse et al. "The Bittorrent P2P File-sharing System: Measurements and Analysis". In: *Proceedings of the 4th International Conference on Peer-to-Peer Systems*. IPTPS'05. Ithaca, NY: Springer-Verlag, 2005, pp. 205–216. ISBN: 3-540-29068-0, 978-3-540-29068-1. DOI: `10.1007/11558989_19`. URL: `http://dx.doi.org/10.1007/11558989_19`.