

IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)

Juan Benet
juan@benet.ai

ABSTRACT

The InterPlanetary File System is a peer-to-peer distributed and versioned file system. It is capable of sharing the same set of files with millions of nodes. It provides a content-addressed Merkle DAG data model that applications build data structures directly, and it can be mounted to explore as any other filesystem. IPFS combines a distributed hashtable, cryptographic techniques, merkle trees, content-addressable storage, bittorrent, tag-based filesystems, and SFS to build a single massive file system shared between peers. IPFS has no single point of failure, and nodes do not need to trust each other.

1. INTRODUCTION

There have been many attempts at constructing a global distributed file system. Some systems have seen significant success, and others failed completely. Among the academic attempts, [?] has succeeded widely and is still in use today. [?], [?], and others have not been so lucky. Outside of academia, the most successful have been peer-to-peer file-sharing systems primarily geared towards large media (audio and video). Most notably, Napster, KaZaA, and BitTorrent [?], the winner of all, deployed massive systems with millions of simultaneous users employing various research-honed techniques. Even today, BitTorrent maintains a massive DHT deployment – with millions of nodes connecting and disconnecting every day.

These systems have succeeded in serving file archives efficiently. But what they have not achieved is tight integration with the application domain. While some have deployed successful application specific forks (e.g. Blizzard Updates [?]), no general file-system has emerged that offers global, low-latency, and decentralized distribution. Perhaps this is because a “good enough” system exists for most use cases: HTTP. By far, HTTP is the most successful “distributed system of files” ever deployed. Coupled with the Browser, HTTP has produced enormous technical and social impact. Yet, it remains oblivious of dozens of brilliant file distribution techniques invented in the last fifteen years. Evolving the Web infrastructure is near-impossible, given the number of backwards compatibility constraints and the number of strong, well invested parties in the current model. Or maybe, what is lacking is the right design, which simultaneously enhances the current web and introduces functionality not present before.

Regardless, most organizations have gotten away with HTTP for now: moving small files around is relatively cheap even for small organizations with lots of traffic. But, we are enter-

ing a new era of data distribution: instead of sharing static media or data processing in large in-house clusters, new challenges are emerging: (a) distributing and hosting large (petabyte) datasets, (b) computing on large data across organizations, (c) high-volume high-definition on-demand or real-time media streams, (d) versioning of massive datasets, and more. All of these can be boiled down to “lots of data, accessible everywhere.” Pressed by critical features and bandwidth concerns, we will turn to different data distribution protocols.

There is one system that has had remarkable success and influence when it comes to data workflow design. Git, the distributed source code version control system, has discovered many right ways to handle data operations. The toolchain around Git offers versatile versioning functionality that large file distribution systems severely lack. New solutions are emerging, such as Dat [?], to provide these features and workflows. And, there is another lesson to learn from Git: its content addressed, Merkle DAG data model enables powerful file distribution – and data structure distribution – strategies.

This paper introduces IPFS, a novel peer-to-peer version-controlled filesystem seeking to reconcile these issues. IPFS learns from many past successful systems. The core driver of its design is this: careful interface-focused integration of these compatible pieces yields a system greater than the sum of its parts. The central principle is modeling *all data* as part of the same Merkle DAG.

Notation Notes: (a) data structures are specified in Go syntax, (b) rpc protocols are specified in capnp interfaces, (c) wire protocols are specified in capnp schemas.

2. BACKGROUND

This section reviews important properties of successful peer-to-peer systems, which IPFS combines.

2.1 Distributed Hash Tables

Distributed Hash Tables (DHTs) are widely used to coordinate and maintain metadata about peer-to-peer systems. For example, the BitTorrent MainlineDHT tracks sets of peers part of a torrent swarm.

2.1.1 Kademlia DHT

Kademlia [?] is a popular DHT that provides:

1. Efficient lookup through massive networks: queries on average contact $\lceil \log_2(n) \rceil$ nodes. (e.g. 20 hops for a network of 10,000,000 nodes).

2. Low coordination overhead: it optimizes the number of control messages it sends to other nodes.
3. Resistance to various attacks, by preferring nodes who have been part of the DHT longer.
4. wide usage in peer-to-peer applications, including Gnutella and Bittorrent, forming networks of over 100 million nodes.

2.1.2 Coral DSHT

While some peer-to-peer filesystems store data blocks directly in DHTs, this “wastes storage and bandwidth, as data must be stored at nodes where it is not needed” [?]. Coral extends Kademlia in three particularly important ways:

1. Kademlia stores values in nodes whose ids are “nearest” (using XOR-distance) to the key. This does not take into account application data locality, ignores “far” nodes who may already have the data, and forces “nearest” nodes to store it, whether they need it or not. This wastes significant storage and bandwidth. Instead, Coral stores addresses to peers who can provide the data blocks.
2. Coral relaxes the DHT API from `get_value(key)` to `get_any_values(key)` (the “sloppy” in DSHT). This still works since Coral users only need a single (working) peer, not the complete list. In return, Coral can distribute only subsets of the values to the “nearest” nodes, avoiding hot-spots (overloading *all the nearest nodes* when a key becomes popular).
3. Additionally, Coral organizes a hierarchy of separate DSHTs called *clusters* depending on region and size. This enables nodes to query peers in their region first, “finding nearby data without querying distant nodes” and greatly reducing the latency of lookups.

2.1.3 S/Kademlia DHT

S/Kademlia extends Kademlia to protect against malicious attacks:

1. S/Kademlia provides schemes to secure `NodeId` generation, and prevent Sybill attacks. It requires nodes to create a PKI key pair, derive their identity from it, and sign their messages to each other. One scheme includes a proof-of-work crypto puzzle to make generating Sybills expensive.
2. S/Kademlia nodes lookup values over disjoint paths, in order to ensure honest nodes can connect to each other in the presence of a large fraction of adversaries in the network. S/Kademlia achieves a success rate of 0.85 even with an adversarial fraction as large as half of the nodes.

2.2 Block Exchanges - BitTorrent

BitTorrent [?] is a widely successful peer-to-peer filesharing system, which succeeds in coordinating networks of untrusting peers (swarms) to cooperate in distributing pieces of files to each other. Key BitTorrent features that inform IPFS design:

1. BitTorrent’s data exchange protocol uses a quasi tit-for-tat strategy which rewards nodes that contribute to each other, and punishes nodes who would only leech others’ resources.
2. BitTorrent peers track the availability of file pieces, prioritizing sending rarest-first. This takes load off seeds, making non-seed peers capable of trading with each other.
3. BitTorrent’s standard tit-for-tat is vulnerable to some exploitative bandwidth sharing strategies. PropShare [?] is a different peer bandwidth allocation strategy that better resists exploitative strategies, and improves the performance of swarms.

2.3 Version Control Systems - Git

Version Control Systems provide facilities to model files changing over time and distribute different versions efficiently. The popular version control system Git provides a powerful Merkle DAG ¹ object model that captures changes to a filesystem tree in a distributed-friendly way.

1. Immutable objects represent Files (`blob`), Directories (`tree`), and Changes (`commit`).
2. Objects are content-addressed, by the cryptographic hash of their contents.
3. Links to other objects are embedded, forming a Merkle DAG. This provides many useful integrity and workflow properties.
4. Most versioning metadata (branches, tags, etc) are simply pointer references, and thus inexpensive to create and update.
5. Version changes only update references or add objects.
6. Distributing version changes to other users is simply transferring objects and updating remote references.

2.4 Self-Certified Filesystems - SFS

SFS [?] proposed a compelling solution to both (a) implementing distributed trust chains, in a (b) egalitarian shared global namespace. SFS introduces a technique for building *Self-Certified Filesystem*: address remote filesystems via the following scheme

`/sfs/<Location>:<HostID>`

Where `Location` is the filesystem’s server network address, and:

`HostID = hash(public_key || Location)`

Thus the *name* of an SFS file system certifies its server. The user can verify the public key offered by the server, negotiate a shared secret, and secure all traffic. Additionally all SFS instances share a global namespace where name allocation is cryptographic, not gated by any centralized body.

¹Merkle Directed Acyclic Graph – similar but more general construction than a Merkle Tree. Deduplicated, does not need to be balanced, and non-leaf nodes contain data.

3. DESIGN

IPFS is a distributed file system which synthesizes successful ideas from previous peer-to-peer systems, including DHTs, BitTorrent, Git, and SFS. The contribution of IPFS is simplifying, evolving, and connecting proven techniques into a single cohesive system, greater than the sum of its parts. IPFS presents a new platform for writing and deploying applications, a new system for distributing and versioning large data, and could evolve the web itself.

IPFS is peer-to-peer; no nodes are privileged. IPFS nodes store IPFS objects in local storage. Nodes connect to each other and transfer objects. These objects represent files and other data structures. The IPFS Protocol is divided into a stack of sub-protocols responsible for different functionality:

1. **Identities** - manages node identity generation and verification. Described in Section 3.1.
2. **Network** - manages connections to other peers, using various underlying network protocols. Configurable. Described in Section 3.2.
3. **Routing** - maintains information to locate specific peers and objects. Responds to both local and remote queries. Defaults to a DHT, but is swappable. Described in Section 3.3.
4. **Exchange** - a novel block exchange protocol (BitSwap) that governs efficient block distribution. Modelled as a market, weakly incentivizes replication. Trade Strategies swappable. Described in Section 3.4.
5. **Objects** - a Merkle DAG of content-addressed immutable objects with links. Used to represent arbitrary datastructures, e.g. file hierarchies and communication systems. Described in Section 3.5.
6. **Files** - versioned file system hierarchy, inspired by Git. Described in Section 3.6.
7. **Naming** - A self-certifying mutable name system. Described in Section 3.7.

These subsystems are not independent; they are integrated and leverage blended properties. However, it is useful to describe them separately, building the protocol stack from the bottom up.

3.1 Identities

Nodes are identified by a `NodeId`, the cryptographic hash² of a public-key, created with S/Kademlia's static crypto puzzle [?]. Nodes store their public and private keys (encrypted with a passphrase). Users are free to instantiate a "new" node identity on every launch, though that loses accrued network benefits. Nodes are incentivized to remain the same.

```
type NodeId Multihash
type Multihash []byte
// self-describing cryptographic hash digest

type PublicKey []byte
type PrivateKey []byte
// self-describing keys
```

²throughout this document, *hash* and *checksum* refer specifically to cryptographic hash checksums of data

```
type Node struct {
    NodeId NodeId
    PubKey PublicKey
    PriKey PrivateKey
}
```

S/Kademlia based IPFS identity generation:

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

Upon first connecting, peers exchange public keys, and check: `hash(other.PublicKey) equals other.NodeId`. If not, the connection is terminated.

Note on Cryptographic Functions.

Rather than locking the system to a particular set of function choices, IPFS favors self-describing values. Hash digest values are "multihashes", a format including a short header identifying the hash function used, and the digest length in bytes. Example:

`<function code><digest length><digest bytes>`

This allows the system (a) chose the best function for the use case (e.g. stronger security vs faster performance), and (b) evolve as function choices change. Self-describing values allow using different parameter choices compatibly.

3.2 Network

IPFS nodes communicate regularly with hundreds of other nodes in the network, across the wide internet. IPFS can use any reliable transport protocol, and is best suited for WebRTC DataChannels [?] (for browser connectivity) or uTP [?] (LEDBAT) [?]. IPFS also uses the ICE NAT traversal techniques [?] to increase connectivity between peers.

- **Reliability:** IPFS can provide reliability if underlying networks do not provide it, using uTP or SCTP.
- **Integrity:** optionally checks integrity of messages using a hash checksum.
- **Authenticity:** optionally checks authenticity of messages using HMAC with sender's public key.

3.3 Routing

IPFS nodes require a routing system that can find (a) other peers' network addresses and (b) peers who can serve particular objects. IPFS achieves this using a DSHT based on S/Kademlia and Coral, using the properties discussed in 2.1. The size of objects and use patterns of IPFS are similar to Coral [?] and Mainline [?], so references are stored in the DHT instead of entire blocks. References are the `NodeIds` of peers who can serve the block.

The interface of this DSHT is:

```

routing.findPeer(node NodeId)
// gets a particular peer's network address

routing.findValuePeers(key Multihash, min int)
// gets a number of peers serving a value

routing.setValue(key []bytes, value []bytes)
// stores a small metadata value in the DHT

routing.provideValue(key Multihash)
// announces that this node can serve a value

```

Note: different use cases will call for substantially different routing systems (e.g. DHT in wide network, static HT in local network). Thus the IPFS routing system can be swapped for one to fit the users' needs. As long as the interface above is met, the rest of the system will continue to function.

3.4 Block Exchange - BitSwap Protocol

The exchange of data in IPFS happens by exchanging blocks with peers using a BitTorrent inspired protocol: BitSwap. Like BitTorrent, BitSwap peers are looking to acquire a set of blocks, and have blocks to offer in exchange. Unlike BitTorrent, BitSwap is not limited to the blocks in one torrent. BitSwap operates as a persistent marketplace where node can acquire the blocks they need, regardless of what files the blocks are part of. The blocks could come from completely unrelated files in the filesystem. But nodes come together to barter in the marketplace.

While the notion of a barter system implies a virtual currency could be created, this would require a global ledger to track ownership and transfer of the currency. This can be implemented as a BitSwap Strategy, and will be explored in a future paper.

In the base case, BitSwap nodes have to provide direct value to each other in the form of blocks. This works fine when the distribution of blocks across nodes is such that they have complements, what each other wants. This will seldom be the case. Instead, it is more likely that nodes must *work* for their blocks. In the case that a node has nothing that its peers want (or nothing at all), it seeks the pieces its peers want, with lower priority than what the node wants itself. This incentivizes nodes to cache and disseminate rare pieces, even if they are not interested in them directly.

3.4.1 BitSwap Credit

The protocol must also incentivize nodes to seed when they do not need anything in particular, as they might have the blocks others want. Thus, BitSwap nodes send blocks to their peers optimistically, expecting the debt to be repaid. But, leeches (free-loading nodes that never share) must be protected against. A simple credit-like system solves the problem:

1. Peers track their balance (in bytes verified) with other nodes.
2. Peers send blocks to debtor peers probabilistically, according to a function that falls as debt increases.

Note that if a peer decides not to send, the peer subsequently ignores the other node for an `ignore_cooldown` timeout. This prevents senders from trying to game the

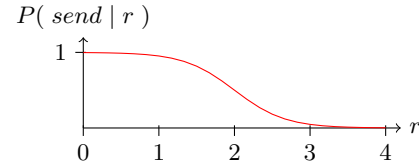


Figure 1: Probability of Sending as r increases

probability by just causing more dice-rolls. (Default BitSwap is 10 seconds).

3.4.2 BitSwap Strategy

The differing strategies that BitSwap peers might employ have wildly different effects on the performance of the exchange as a whole. In BitTorrent, while a standard strategy is specified (tit-for-tat), a variety of others have been implemented, ranging from BitTyrant [?] (sharing the least-possible), to BitThief [?] (exploiting a vulnerability and never share), to PropShare [?] (sharing proportionally). A range of strategies (good and malicious) could similarly be implemented by BitSwap peers. The choice of function, then, should aim to:

1. maximize the trade performance for the node, and the whole exchange
2. prevent freeloaders from exploiting and degrading the exchange
3. be effective with and resistant to other, unknown strategies
4. be lenient to trusted peers

The exploration of the space of such strategies is future work. One choice of function that works in practice is a sigmoid, scaled by a *debt ratio*:

Let the *debt ratio* r between a node and its peer be:

$$r = \frac{\text{bytes_sent}}{\text{bytes_recv} + 1}$$

Given r , let the probability of sending to a debtor be:

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

As you can see in Figure 1, this function drops off quickly as the nodes' *debt ratio* surpasses twice the established credit. The *debt ratio* is a measure of trust: lenient to debts between nodes that have previously exchanged lots of data successfully, and merciless to unknown, untrusted nodes. This (a) provides resistance to attackers who would create lots of new nodes (sybill attacks), (b) protects previously successful trade relationships, even if one of the nodes is temporarily unable to provide value, and (c) eventually chokes relationships that have deteriorated until they improve.

3.4.3 BitSwap Ledger

BitSwap nodes keep ledgers accounting the transfers with other nodes. This allows nodes to keep track of history, and to avoid tampering. When activating a connection, BitSwap nodes exchange their ledger information. If it does not match exactly, the ledger is reinitialized from scratch,

loosing the accrued credit or debt. It is possible for malicious nodes to purposefully “loose” the Ledger, hoping the erase debts. It is unlikely that nodes will have accrued enough debt to warrant also losing the accrued trust, however the partner node is free to count it as *misconduct* (discussed later).

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

Nodes are free to keep the ledger history, though it is not necessary for correct operation. Only the current ledger entries are useful. Nodes are also free to garbage collect ledgers as necessary, starting with the less useful ledgers: the old (peers may not exist anymore) and small.

3.4.4 BitSwap Specification

BitSwap nodes follow a simple protocol.

```
// Additional state kept
type BitSwap struct {
    ledgers map[NodeId]Ledger
    // Ledgers known to this node, inc inactive

    active map[NodeId]Peer
    // currently open connections to other nodes

    need_list []Multihash
    // checksums of blocks this node needs

    have_list []Multihash
    // checksums of blocks this node has
}

type Peer struct {
    nodeid NodeId
    ledger Ledger
    // Ledger between the node and this peer

    last_seen Timestamp
    // timestamp of last received message

    want_list []Multihash
    // checksums of all blocks wanted by peer
    // includes blocks wanted by peer's peers
}

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

Sketch of the lifetime of a peer connection:

1. Open: peers send ledgers until they agree.
2. Sending: peers exchange want_lists and blocks.

3. Close: peers deactivate a connection.

4. Ignored: (special) a peer is ignored (for the duration of a timeout) if a node's strategy avoids sending

Peer.open(NodeId, Ledger).

When connecting, a node initializes a connection with a **Ledger**, either stored from a connection in the past or a new one zeroed out. Then, sends an Open message with the **Ledger** to the peer.

Upon receiving an **Open** message, a peer chooses whether to activate the connection. If – according to the receiver's **Ledger** – the sender is not a trusted agent (transmission below zero, or large outstanding debt) the receiver may opt to ignore the request. This should be done probabilistically with an **ignore_cooldown** timeout, as to allow errors to be corrected and attackers to be thwarted.

If activating the connection, the receiver initializes a **Peer** object, with the local version of the **Ledger**, and setting the **last_seen** timestamp). Then, it compares the received **Ledger** with its own. If they match exactly, the connections have opened. If they do not match, the peer creates a new zeroed out **Ledger**, and sends it.

Peer.send_want_list(WantList).

While the connection is open, nodes advertise their **want_list** to all connected peers. This is done (a) upon opening the connection, (b) after a randomized periodic timeout, (c) after a change in the **want_list** and (d) after receiving a new block.

Upon receiving a **want_list**, a node stores it. Then, it checks whether it has any of the wanted blocks. If so, it sends them according to the *BitSwap Strategy* above.

Peer.send_block(Block).

Sending a block is straightforward. The node simply transmits the block of data. Upon receiving all the data, the receiver computes the Multihash checksum to verify it matches the expected one, and returns confirmation.

Upon finalizing the correct transmission of a block, the receiver moves the block from **need_list** to **have_list**, and both the receiver and sender update their ledgers to reflect the additional bytes transmitted.

If a transmission verification fails, the receiver instead *penalizes* the sender. Both receiver and sender should update their ledgers accordingly, though the sender is either malfunctioning or attacking the receiver. Note that BitSwap expects to operate on a reliable transmission channel, so data errors – which could lead to incorrect penalization of an honest sender – are expected to be caught before the data is given to BitSwap. IPFS uses the uTP protocol.

Peer.close(Bool).

The **final** parameter to **close** signals whether the intention to tear down the connection is the sender's or not. If false, the receiver may opt to re-open the connection immediately. This avoids premature closes.

A peer connection should be closed under two conditions:

- a **silence_wait** timeout has expired without receiving any messages from the peer (default BitSwap uses 30 seconds). The node issues **Peer.close(false)**.

- the node is exiting and BitSwap is being shut down. In this case, the node issues `Peer.close(true)`.

After a `close` message, both receiver and sender tear down the connection, clearing any state stored. The `Ledger` may be stored for the future, if it is useful to do so.

Notes.

- Non-`open` messages on an inactive connection should be ignored. In case of a `send_block` message, the receiver may check the block to see if it is needed and correct, and if so, use it. Regardless, all such out-of-order messages trigger a `close(false)` message from the receiver, to force re-initialization of the connection.

3.5 Object Merkle DAG

The DHT and BitSwap allow IPFS to form a massive peer-to-peer system for storing and distributing blocks quickly and robustly to users. On top of these, IPFS builds a Merkle DAG, a directed acyclic graph where links between objects are cryptographic hashes of the targets embedded in the sources. This is a generalization of which the Git data structure is a special case, and upon which it can be built. Merkle DAGs provide IPFS many useful properties, including:

1. **Content Addressing:** all content is uniquely identified by its `multihash` checksum, **including links**.
2. **Tamper resistance:** all content is verified with its checksum. If data is tampered with or corrupted, IPFS detects it.
3. **Deduplication:** all objects who hold the exact same content are equal, and only stored once. This is particularly useful with index objects, such as git `trees` and `commits`, or common portions of data.

The IPFS Object format is:

```
type IPFSLink struct {
    Name string
    // name or alias of this link

    Hash Multihash
    // cryptographic hash of target

    Size int
    // total size of target
}

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

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

The IPFS Merkle DAG is an extremely flexible way to store data. The only requirements are that object references be (a) content addressed, and (b) encoded in the format above. IPFS grants applications complete control over the

data field; applications can use any custom data format they chose, which IPFS may not understand. The separate in-object link table allows IPFS to:

- List all object references in an object. For example:

```
> ipfs ls /XLZ1625Jjn7SubMDgEyeaynFuR84ginqvzb
XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x 189458 less
XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5 19441 script
XLF4hwVhsVuZ78FZK6fozf8Jj9WEURMbCX4 5286 template

<object multihash> <object size> <link name>
```

- Resolve string path lookups, such as `foo/bar/baz`. Given an object, IPFS resolves the first path component to a hash in the object's link table, fetching that second object, and repeats with the next component. Thus, string paths can walk the Merkle DAG no matter what the data formats are.
- Resolve all objects referenced recursively:

```
> ipfs refs --recursive \
  /XLZ1625Jjn7SubMDgEyeaynFuR84ginqvzb
XLLxhdgJcXzLbtsLRL1twCHA2NrURp4H38s
XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x
XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5
XLWVQDqxo9Km9zLyquoC9gAP8CL1gWnHZ7z
...
```

A raw data field and a common link structure are the necessary components for constructing arbitrary data structures on top of IPFS. While it is easy to see how the Git object model fits on top of this DAG, consider these other potential data structures: (a) key-value stores (b) traditional relational databases (c) Linked Data triple stores (d) linked document publishing systems (e) linked communications platforms (f) cryptocurrency blockchains. These can all be modeled on top of the IPFS Merkle DAG, which allows any of these systems to use IPFS as a transport protocol for more complex applications.

3.5.1 Paths

IPFS objects can be traversed with a string path API. Paths work as they do in traditional UNIX filesystems and the Web. The Merkle DAG links make traversing it easy. Note that full paths in IPFS are of the form:

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

# example
/ipfs/XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x/foo.txt
```

The `/ipfs` prefix allows mounting into existing systems at a standard mount point without conflict (mount point names are of course configurable). The second path component (first within IPFS) is the hash of an object. This is always the case, as there is no global root. A root object would have the impossible task of handling consistency of millions of objects in a distributed (and possibly disconnected) environment. Instead, we simulate the root with content addressing. All objects are always accessible via their hash. Note this means that 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>

```

3.5.2 Local Objects

IPFS clients require some *local storage*, an external system on which to store and retrieve local raw data for the objects IPFS manages. The type of storage depends on the node's use case. In most cases, this is simply a portion of disk space (either managed by the native filesystem, by a key-value store such as leveldb [?], or directly by the IPFS client). In others, non-persistent caches for example, this storage is just a portion of RAM.

Ultimately, all blocks available in IPFS are in some node's *local storage*. When users request objects, they are found, downloaded, and stored locally, at least temporarily. This provides fast lookup for some configurable amount of time thereafter.

3.5.3 Object Pinning

Nodes who wish to ensure the survival of particular objects can do so by **pinning** the objects. This ensures the objects are kept in the node's *local storage*. Pinning can be done recursively, to pin down all linked descendent objects as well. All objects pointed to are then stored locally. This is particularly useful for nodes wishing to keep all their own files, or backup references to others. This also makes IPFS a Web where links are *permanent*, and Objects can ensure the survival of others they point to.

3.5.4 Publishing Objects

IPFS is globally distributed. It is designed to allow the files of millions of users to coexist together. The **DHT** with content-hash addressing allows publishing objects in a fair, secure, and entirely distributed way. Anyone can publish an object by simply adding its key to the DHT, adding themselves as a peer, and giving other users the object's path. Note that Objects are essentially immutable, just like in Git. New versions hash differently, and thus are new objects. Tracking versions is the job of additional versioning objects.

3.5.5 Object-level Cryptography

IPFS is equipped to handle object-level cryptographic operations. An encrypted or signed object is wrapped in a special frame that allows encryption or verification of the raw bytes.

```

type EncryptedObject struct {
    Object []bytes
    // raw object data encrypted

    Tag []bytes
    // optional tag for encryption groups
}

type SignedObject struct {
    Object []bytes
    // raw object data signed

    Signature []bytes
    // hmac signature

    PublicKey []multihash

```

```

// multihash identifying key
}

```

Cryptographic operations change the object's hash (defining a different object, as it should). IPFS automatically verifies signatures, and can decrypt data with user-specified keychains. Links of encrypted objects are protected as well, making traversal impossible without a decryption key. It is possible to have a parent object encrypted under one key, and a child under another or not at all. This allows securing links to shared objects.

3.6 Files

IPFS also defines a set of objects for modeling a versioned filesystem on top of the Merkle DAG. This object model is similar to Git's:

1. **block**: a variable-size block of data.
2. **list**: a collection of blocks or other lists.
3. **tree**: a collection of blocks, lists, or other trees.
4. **commit**: a snapshot in the version history of a tree.

I hoped to use the Git object formats exactly, but had to depart to introduce certain features useful in a distributed filesystem, namely (a) fast size lookups (aggregate byte sizes have been added to objects), (b) large file deduplication (adding a **list** object), and (c) embedding of **commits** into **trees**. However, IPFS File objects are close enough to Git that conversion between is possible. Also, a set of Git objects can be introduced to convert between the two without losing any information (unix file permissions, etc).

Notation: File object formats below use JSON. Note that this structure is actually binary encoded using protobufs. Though, ipfs includes import/export to JSON.

3.6.1 File Object: blob

The **blob** object contains an addressable unit of data, and represents a file. IPFS Blocks are like Git blobs or filesystem data blocks. They store the users' data. Note that IPFS files can be represented by both **lists** and **blobs**. Blobs have no links.

```

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

```

3.6.2 File Object: list

The **list** object represents a large or de-duplicated file made up of several IPFS **blobs** concatenated together. **lists** contain an ordered sequence of **blob** or **list** objects. In a sense, the IPFS **list** functions like a filesystem file with indirect blocks. Since **lists** can contain other **lists**, topologies including linked lists and balanced trees are possible. Directed graphs where the same node appears in multiple places allow in-file deduplication. Of course, cycles are not possible, as enforced by hash addressing.

```

{
  "data": ["blob", "list", "blob"],
  // lists have an array of object types as data
  "links": [

```

```

    { "hash": "XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x",
      "size": 189458 },
    { "hash": "XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5",
      "size": 19441 },
    { "hash": "XLWVQDqxo9Km9zLyquoC9gAP8CL1gWnHZ7z",
      "size": 5286 }
  // lists have no names in links
}
}

```

3.6.3 File Object: tree

The **tree** object in IPFS is similar to Git trees: it represents a directory, a map of names to hashes. The hashes reference **blobs**, **lists**, other **trees**, or **commits**. Note that traditional path naming is already implemented by the Merkle DAG. Though, collapsing **commits** and **lists**, and representing only **trees** as directories is achieved by a special mounting application which exposes the objects through a different file system interface.

```

{
  "data": ["blob", "list", "blob"],
  // trees have an array of object types as data
  "links": [
    { "hash": "XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x",
      "name": "less", "size": 189458 },
    { "hash": "XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5",
      "name": "script", "size": 19441 },
    { "hash": "XLWVQDqxo9Km9zLyquoC9gAP8CL1gWnHZ7z",
      "name": "template", "size": 5286 }
  // trees do have names
  ]
}

```

3.6.4 File Object: commit

The **commit** object in IPFS represents a snapshot in the version history of any object. It is similar to Git's, but can reference any type of object. It also links to author objects.

```

{
  "data": {
    "type": "tree",
    "date": "2014-09-20 12:44:06Z",
    "message": "This is a commit message."
  },
  "links": [
    { "hash": "XLa1qMBKiSEEDhojb9FFZ4tEvLf7FEQdhdU",
      "name": "parent", "size": 25309 },
    { "hash": "XLGw74KAy9junbh28x7ccWov9inu1Vo7pnX",
      "name": "object", "size": 5198 },
    { "hash": "XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm",
      "name": "author", "size": 109 }
  ]
}

```

3.6.5 Version control

The **commit** object represents a particular snapshot in the version history of an object. Comparing the object (and children) of two different commits reveals the differences between two versions of the filesystem. As long as a single **commit** and all the children objects it references are accessible, all preceding versions are retrievable and the full history

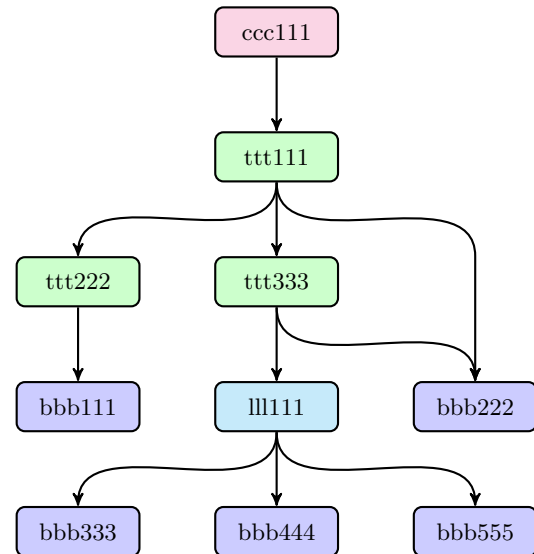


Figure 2: Sample Object Graph

```

> ipfs file-cat <ccc111-hash> --json
{
  "data": {
    "type": "tree",
    "date": "2014-09-20 12:44:06Z",
    "message": "This is a commit message."
  },
  "links": [
    { "hash": "<ccc000-hash>",
      "name": "parent", "size": 25309 },
    { "hash": "<ttt111-hash>",
      "name": "object", "size": 5198 },
    { "hash": "<aaa111-hash>",
      "name": "author", "size": 109 }
  ]
}

> ipfs file-cat <ttt111-hash> --json
{
  "data": ["tree", "tree", "blob"],
  "links": [
    { "hash": "<ttt222-hash>",
      "name": "ttt222-name", "size": 1234 },
    { "hash": "<ttt333-hash>",
      "name": "ttt333-name", "size": 3456 },
    { "hash": "<bbb222-hash>",
      "name": "bbb222-name", "size": 22 }
  ]
}

> ipfs file-cat <bbb222-hash> --json
{
  "data": "blob222 data",
  "links": []
}

```

Figure 3: Sample Objects

of the filesystem changes can be accessed. This is a consequence of the Merkle DAG object model.

The full power of the `Git` version control tools is available to IPFS users. The object model is compatible, though not the same. It is possible to (a) build a version of the `Git` tools modified to use the IPFS object graph, (b) build a mounted FUSE filesystem that mounts an IPFS `tree` as a `Git` repo, translating `Git` filesystem read/writes to the IPFS formats.

3.6.6 Filesystem Paths

As we saw in the Merkle DAG section, IPFS objects can be traversed with a string path API. The IPFS File Objects are designed to make mounting IPFS onto a UNIX filesystem simpler. They restrict `trees` to have no data, in order to represent them as directories, and `commits` can either be represented as directories too, or hidden from the filesystem entirely.

3.6.7 Splitting Files into Lists and Blob

One of the main challenges with versioning and distributing large files is finding the right way to split them into independent blocks. Rather than assume it can make the right decision for every type of file, IPFS offers the following alternatives:

- (a) Use Rabin Fingerprints [?] as in LBFS [?] to pick suitable block boundaries.
- (b) Use the rsync [?] rolling-checksum algorithm, to detect blocks that have changed between versions.
- (c) Allow users to specify block-splitting functions highly tuned for specific files.

Path Lookup Performance.

Path-based access traverses the object graph. Retrieving each object requires potentially looking up its key in the DHT, connecting to peers, and retrieving its blocks. This is considerable overhead, particularly when looking up paths with many components. This is mitigated by:

- **tree caching**: since all objects are hash-addressed, they can be cached indefinitely. Additionally, `trees` tend to be small in size so IPFS prioritizes caching them over `blobs`.
- **flattened trees**: for any given `tree`, a special `flattened tree` can be constructed to list all objects reachable from the `tree`. Names in the `flattened tree` would really be paths parting from the original tree, with slashes.

For example, `flattened tree` for `ttt111` above:

```
{
  "data":
    ["tree", "blob", "tree", "list", "blob" "blob"],
  "links": [
    { "hash": "<ttt222-hash>", "size": 1234
      "name": "ttt222-name" },
    { "hash": "<bbb111-hash>", "size": 123,
      "name": "ttt222-name/bbb111-name" },
    { "hash": "<ttt333-hash>", "size": 3456,
      "name": "ttt333-name" },
```

```
    { "hash": "<lll111-hash>", "size": 587,
      "name": "ttt333-name/lll111-name"},
    { "hash": "<bbb222-hash>", "size": 22,
      "name": "ttt333-name/lll111-name/bbb222-name" },
    { "hash": "<bbb222-hash>", "size": 22
      "name": "bbb222-name" }
  ] }
```

3.7 IPNS: Naming and Mutable State

So far, the IPFS stack describes a peer-to-peer block exchange constructing a content-addressed DAG of objects. It serves to publish and retrieve immutable objects. It can even track the version history of these objects. However, there is a critical component missing: Mutable Naming. Without it, all communication of new content must happen off-band, via sending links to each other. What is required is some way to retrieve mutable state at *the same path*.

It is worth stating why – if mutable data is necessary in the end – we worked hard to build up an *immutable* Merkle DAG. Consider the properties of IPFS that fall out of the Merkle DAG: Objects can be (a) retrieved via their hash, (b) integrity checked, (c) linked to others, and (d) cached indefinitely. In a sense:

Objects are **permanent**.

These are the critical properties of a high-performance distributed system, where data is expensive to move across network links. Object content addressing constructs a web with (a) significant bandwidth optimizations, (b) untrusted content serving, (c) permanent links, and (d) the ability to make full permanent backups of any object and its references.

The Merkle DAG and Naming, immutable content-addressed objects and mutable pointers, instantiate a dichotomy present in many successful distributed systems. Most notably, the `Git` Version Control System with its immutable objects and mutable refs. So does `Plan9` [?], the distributed successor to UNIX, with its mutable `Fossil` [?] and immutable `Venti` [?] filesystems. `LBFS` [?] also uses mutable indices and immutable chunks.

3.7.1 Self-Certified Names

Using the self-certification naming scheme from `SFS` [?] gives us a way to construct (a) self-certified (verifiable) names, (b) in another cryptographically assigned global namespace, that are (c) mutable. The IPFS scheme is as follows.

1. Recall that in IPFS:

```
NodeId = hash(node.PubKey)
```

2. We assign every user a mutable namespace at:

```
/ipns/<NodeId>
```

3. A user can publish (described below) an Object to this path **Signed** by her private key, say at:

```
/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/
```

4. When other users retrieve the object, they can check the signature matches the public key and `NodeId`, verifying that this indeed was an Object published by the user, achieving the mutable state retrieval.

Note the following details:

- The **ipns** (InterPlanetary Name Space) separate prefix is to cause a recognizable distinction in human path readers between *mutable* and *immutable* paths.
- because this is *not* a content-addressed object, publishing it relies on the only mutable state distribution system in IPFS, the Routing system. The process is (1) publish the object as a regular immutable IPFS object, (2) publish its hash on the Routing system as a metadata value:

```
routing.setValue(NodeId, <ns-object-hash>)
```

- any links in the Object published act as sub-names in the namespace:

```
/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/  
/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/docs  
/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/docs/ipfs
```

- it is advised to publish a **commit** object, or some other object with a version history so that clients may be able to find old names. This is left as a user option, as it is not always desired.

Note that when users publish this Object, it cannot be published in the same way

3.7.2 Human Friendly Names

While IPNS is indeed a way of assigning and reassigning names, it is not very user friendly, as it exposes long hash values as names, which are notoriously hard to remember. These work for URLs, but not for many kinds of offline transmission. Thus, IPFS increases the user-friendliness of IPNS with the following techniques.

Peer links.

As encouraged by SFS, users can link other users' Objects directly into their own Objects (namespace, home, etc). This has the benefit of also creating a web of trust (and supports the old Certificate Authority model):

```
# Alice links to bob Bob  
ipfs link /<alice-pk-hash>/friends/bob /<bob-pk-hash>  
  
# Eve links to Alice  
ipfs link /<eve-pk-hash>/friends/alice /<alice-pk-hash>  
  
# Eve also has access to Bob  
/<eve-pk-hash>/friends/alice/friends/bob  
  
# access Verisign certified domains  
/<verisign-pk-hash>/foo.com
```

DNS TXT IPNS Records.

If `/ipns/<domain>` is a valid domain name, IPFS looks up key **ipns** in its DNS TXT records. IPFS interprets the value as either an object hash or another IPNS path:

```
# this DNS TXT record  
ipfs.benet.ai. TXT "ipfs=XLF2ipQ4jD3U ..."  
  
# behaves as symlink  
ln -s /ipns/XLF2ipQ4jD3U /ipns/fs.benet.ai
```

Proquint Pronounceable Identifiers.

There have always been schemes to encode binary into pronounceable words. IPNS supports Proquint [?]. Thus:

```
# this proquint phrase  
/ipns/dahih-dolij-sozuk-vosah-luvar-fuluh  
  
# will resolve to corresponding  
/ipns/KhAwNprxYVxKqpDZ
```

Name Shortening Services.

Services are bound to spring up that will provide name shortening as a service, offering up their namespaces to users. This is similar to what we see today with DNS and Web URLs:

```
# User can get a link from  
/ipns/shorten.er/foobar  
  
# To her own namespace  
/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm
```

3.8 Using IPFS

IPFS is designed to be used in a number of different ways. Here are just some of the usecases I will be pursuing:

1. As a mounted global filesystem, under `/ipfs` and `/ipns`.
2. As a mounted personal sync folder that automatically versions, publishes, and backs up any writes.
3. As an encrypted file or data sharing system.
4. As a versioned package manager for *all* software.
5. As the root filesystem of a Virtual Machine
6. As the boot filesystem of a VM (under a hypervisor).
7. As a database: applications can write directly to the Merkle DAG data model and get all the versioning, caching, and distribution IPFS provides.
8. As a linked (and encrypted) communications platform.
9. As a integrity checked CDN for large files (without SSL)
10. As an encrypted CDN
11. On webpages, as a web CDN.
12. As a new Permanent Web where links do not die.

The IPFS implementations target:

- (a) an IPFS library to import in your own applications.
- (b) commandline tools to manipulate objects directly.
- (c) mounted file system, using FUSE [?].

4. THE FUTURE

The ideas behind IPFS are the product of decades of successful distributed systems research in academia and open source. It synthesizes many of the best ideas from the most successful systems to date. Aside from BitSwap, which is a novel protocol, the main contribution of IPFS is this coupling of systems and synthesis of designs.

IPFS is an ambitious, bold vision of new decentralized Internet infrastructure, upon which many different kinds of applications can be built. At the bare minimum, it can be used as a global mounted versioned filesystem and namespace, or as the next generation file sharing system. At its best, it could push the web to new horizons, where Publishing does not require hosting, where users can trust the content they receive without trusting the peers they receive it from, and where old but important files do not go missing. IPFS looks forward to bringing us toward the Permanent Web.

5. ACKNOWLEDGMENTS

IPFS is the synthesis of many great ideas and systems. It would be impossible to dare such ambitious goals without standing on the shoulders of such giants. Personal thanks to David Dalrymple, Joe Zimmerman, and Ali Yahya for long discussions on many of these ideas, in particular: exposing the general Merkle DAG (David, Joe), rolling hash blocking (David), and s/kademlia sybill protection (David, Ali). And special thanks to David Mazieres, for his ever brilliant ideas.

6. REFERENCES TODO