Ivan Planetary File System 🚀

CPSC 416 Project Final Report

Jonathan Budiardjo, Jinny Byun, Bryan Chiu, Tristan Rice, Bea Subion

Problem Statement

The original implementation of the Interplanetary File System (IPFS) was meant to provide an infrastructure for a distributed file system for the dissemination of versioned data. The system supports file chunk versioning using cryptographic hashes, like Git, and uses a BitTorrent-like dissemination protocol, allowing the system to work without a centralized server. Although IPFS supports node verification through public keys, data transferred between intermediate nodes from a source to a destination have unencrypted access to file contents. This allows for potentially malicious third parties to easily censor documents while enroute to the end nodes.

Thus, we replicated a subset of IPFS features while also extending IPFS to be able to encrypt data between the source and destination such that intermediate nodes are not able to peek at the file contents being transferred through the network. We have also implemented a publishing and subscription system that allows for end users to subscribe to a reference, and listen to messages that are published to the reference on a channel.

Commands

The Ivan Planetary File System's command line client has 11 commands in total for the user to interact with an IPFS node. The command line client can be run by calling `go build ipfs.go` in the `app` folder and then running `./ipfs <node_addr>`.

`get <document_access_id>`
Fetches a document with this access ID and returns the contents of the document. The access ID is in the format of `document_id:access_key`. Since all documents are encrypted, the access key is used to decrypt the document so that the contents can be retrieved. If `access_id` belongs to a directory (a document with children), it will return a list of all the children documents' names and their access IDs instead.

`add <path/to/file>`
Adds a local document to the IPFS and returns the access ID of the document, in the format of `document_id:access_key`.

`add -r <path/to/directory>`
Adds a local directory to the IPFS and returns the access ID of the document, in the format of `document_id:access_key`.

`add -c <documents>`
Creates a parent document to a list of existing documents in the IPFS and returns the access ID of the document. Documents must be in the format of `file_name,access_id` and each document must be semicolon separated. For example:

```
index.html,document1_id:access_key1;foo.html,document2_id:access_key2
```

`peers list`
Lists all of the peer addresses of this node.

`peers add <node_address>`
Add a peer to this node using the given address.

`reference get <reference_access_id>`
Fetches what this reference points to (either a document or another reference) and returns its record, in the format of `document@document_id:access_key` or `reference@reference_id:access_key`.

`reference add <record> <path/to/priv_key>`
Adds a reference to the IPFS (or updates an existing reference) and returns the access ID. The record is in the format of `document@document_id:access_key` or `reference@reference_id:access_key`. The returned access ID is in the format of `reference_id:access_key`.

`publish <message> <path/to/priv_key>`
Publishes a message to a reference on a channel. Nodes subscribed to this reference will then see the message.

`subscribe <reference_id>`
Subscribes to an existing reference and listens for messages on a channel. If a message is published to this reference, they will be seen on this channel.

`quit`
Disconnects from this node and exits the Ivan Planetary File System.

How It All Works

Adding a Document

A document is represented as the following:

```
type Document struct {
     Data []byte
     ContentType string
     Children map[FileName]AccessID
}
```

You can either add a single document or a “directory” of documents (a document with children) to the IPFS. The content type of the document is inferred by the file extension. When adding a single document to an IPFS node, the binary data of the document is encrypted and the access key (the encryption and decryption key for the document) is created from the hash of the original document. The encrypted data is stored on the local node, and the document is added to the node’s bloom filter. The bloom filter entry is then disseminated to the node’s neighbours. The node returns an access ID to the end user, which is the document ID and the access key that is used to later decrypt the document.

Adding a directory is a similar process. The content type of a directory document is set as a “directory” type. If you add a directory of local documents, it recursively adds all the children documents first, following the process for adding a single document. The children documents are then added to a children map with their file name and access IDs. If you create a parent document to a list of documents that already exist in the IPFS, the child documents are just added to a children map with their file name and access IDs. Since the child documents have already existed on the IPFS, there is no need to go through the single document adding process for each of them. Once the directory document has been created, it is encrypted and added to the node’s bloom filter, and the IPFS node returns the access ID of the document to the end user.

Fetching a Document

When fetching a document, the end user specifies an access ID. The document ID is separated from the access ID, and the node checks the local storage for the given file. If the file is found on the local node, the node will retrieve the document. If the file is not found on the local node, the node will look the document up in the bloom filters and network requests will be issued to retrieve the document. The mechanics of the bloom filter is explained in a later section. After the file is found, the document is decrypted with the access key, and the decrypted document is returned to the user. Since only the node that is initially fetching this document has the access

key, intermediate nodes are not able to decrypt and view the contents of the document as they are passing it along across the network.

If the document retrieved is a single document, the contents of the file will be returned to the end user. If the document is a directory, a list of the child documents with their file name and access IDs will be returned to the end user instead.

Adding/Updating a Reference

A reference is represented as the following:

```
type Reference struct {
      Value string
      PublicKey string
      Signature string
      Timestamp int64
}
```

Adding a reference to the IPFS is very similar to adding a document. References are added to the IPFS by passing in a record (either `document@document_id:access_key` or `reference@reference_id:access_key`) as well as a private key. A reference can be of a document or another reference. If the reference does not exist, adding a reference creates it and if it does exist, adding a reference will update it. The reference's value is set to the given record and the value is encrypted in the same way as the document encryption. An access key is generated so that the reference's value can later be decrypted. The reference is signed by the public key, which becomes the reference's signature. The reference's timestamp is just when the reference was created. The reference is then stored on the local node and added to the node's bloom filter. The node returns an access ID to the end user, which comprises of the reference ID (the hash of the given public key) and the access key.

Fetching a Reference

Similar to document fetching, a reference is fetched using its access ID specified by the end user. The reference ID is separated from its access key and the node checks for the reference locally. If the reference is found locally, the node will retrieve it. If it cannot be found locally, the node will use the bloom filters and network requests will be issued to retrieve the reference. After the reference is found, its value (which is a record) is decrypted with the access key and returned to the end user. Once again, since only the node that is initially fetching this reference has the access key, intermediate nodes are not able to decrypt and view the reference's value as they are passing it along across the network.

At each intermediate node the reference signature is verified to avoid tampering, and the message contents is encrypted with a symmetric AES key that is derived from the ECDSA key

pair. This protects the contents, but still allows detection of tampering by intermediate nodes. The AES key is derived by signing a short message with the private key and then by sha256 hashing it.

Publishing to a Reference

Publishing a message to a reference requires a message and a private key. Since a reference ID is just the hash of the public key, the private key specified by the end user is used to find the reference to publish to. The message is published on a channel and nodes in the IPFS that are subscribed to this reference will see the published message in real-time.

Subscribing to a Reference

Clients can subscribe to a reference to be able to receive real time updates. The internal structure if fairly similar to just fetching a reference. However, this opens a persistent stream between the originating node and the client. This ensures “at most once” delivery. It is possible for messages to be lost, and thus in practice it’s recommended to both update the reference with a new message, as well as publish it via PubSub. This way clients who have dropped connections, or newly connecting clients can see the message history. It is possible to get very strong constraints by doing that.

Currently if a reference has N clients connecting to it, there will be N connections on the hosting node. It would be fairly straightforward to deduplicate streams if two clients both pass through an intermediate node, but for simplicity of implementation we do not currently do that.

Multi-Node References

The system is designed to talk to a node providing the reference in the fewest number of network hops. If that node goes down, clients will be unable to access the reference, or subscribe to it. This can be avoided by having the hoster announce the reference on multiple nodes to provide redundancy. It is then the hoster’s responsibility to ensure those nodes are publishing the same data.

Peers

To add a new node, it needs an address of an initial peer to bootstrap itself. When connecting to a new peer, each node sends the other node a list of its peers and peers it knows about. These peers are then used to connect to more peers until the node hits its user configured maximum number of peers. In a small network, this results in a fully connected graph. When connecting to new peers, a node prioritizes connecting to new nodes that do not share a connection with its current peers. This helps minimize the graph width since it will prioritize connecting to nodes over three network hops away instead of just two hops away.

When a node gets a new connection, it establishes a connection back to ensure bidirectional communication, even if it is at the maximum node limit. This is so new nodes are not accidentally partitioned from the graph due to only outward connections.

Network Structure & Security

All communication between nodes is fully encrypted using self-signed certificates and public private key cryptography. This communication runs over HTTP2 and GRPC.

The certificates are verified by requiring a node's metadata before connecting. Each node publishes a "NodeMeta" structure. This contains the node ID, a public key, a list of addresses it can be reached on, a TLS certificate and the last time it was updated. This is then signed with the public key to verify it. The node ID is the hash of the public key. This is done so new nodes can not impersonate other nodes or accidentally establish a connection to an incorrect node.

The TLS certificate is derived from the node's ECDSA key pair and all communications are fully encrypted. The initial bootstrap connection used by a node accepts any certificate initially, but then reconnects with the actual node certificate after fetching the NodeMeta.

Bloom Filters: Adding a Document/Reference

Every single node has a routing table of size h, h representing the peer furthest away. The routing table stores a union of bloom filters of each group of peers `i` hops away. The 0th index of this routing table represents the node's bloom filter. Every time a new file is added to a particular node, that node updates its own bloom filter, and publishes this change to all the other nodes via a grpc stream. Each node will then merge the new filter in the appropriate index of its own routing table.

Bloom Filters: Fetching a Document/Reference

In order to fetch files stored remotely efficiently, we opted to use bloom filters to condense the number of possible peers with a particular file and limit the number of searches we do. That way, a node does not have to do a lookup on all the other nodes. Since bloom filters are a probabilistic data structure that will never give false negatives, a lookup on a node's routing table's bloom filters will tell that node that the file is possibly in a group of peers `i` hops away. The process of fetching a file involves checking whether or not the document ID is in each bloom filter of the routing table. If there is a match, the node asks each peer `i` hops away to do a file lookup and return it if it is found. For example, if a node gets a match 1 hop away, it asks all of their peers whether or not it has the file. If it does, the peer sends back the document hash to the node, which then decodes it. If the node gets a match 2 hops away, none of the node's immediate peers will do a lookup. Instead, each peer will ask all their peers if they have a particular file, return it to the peer one hop away, and return it back to the original node that requested it.

Document Caching

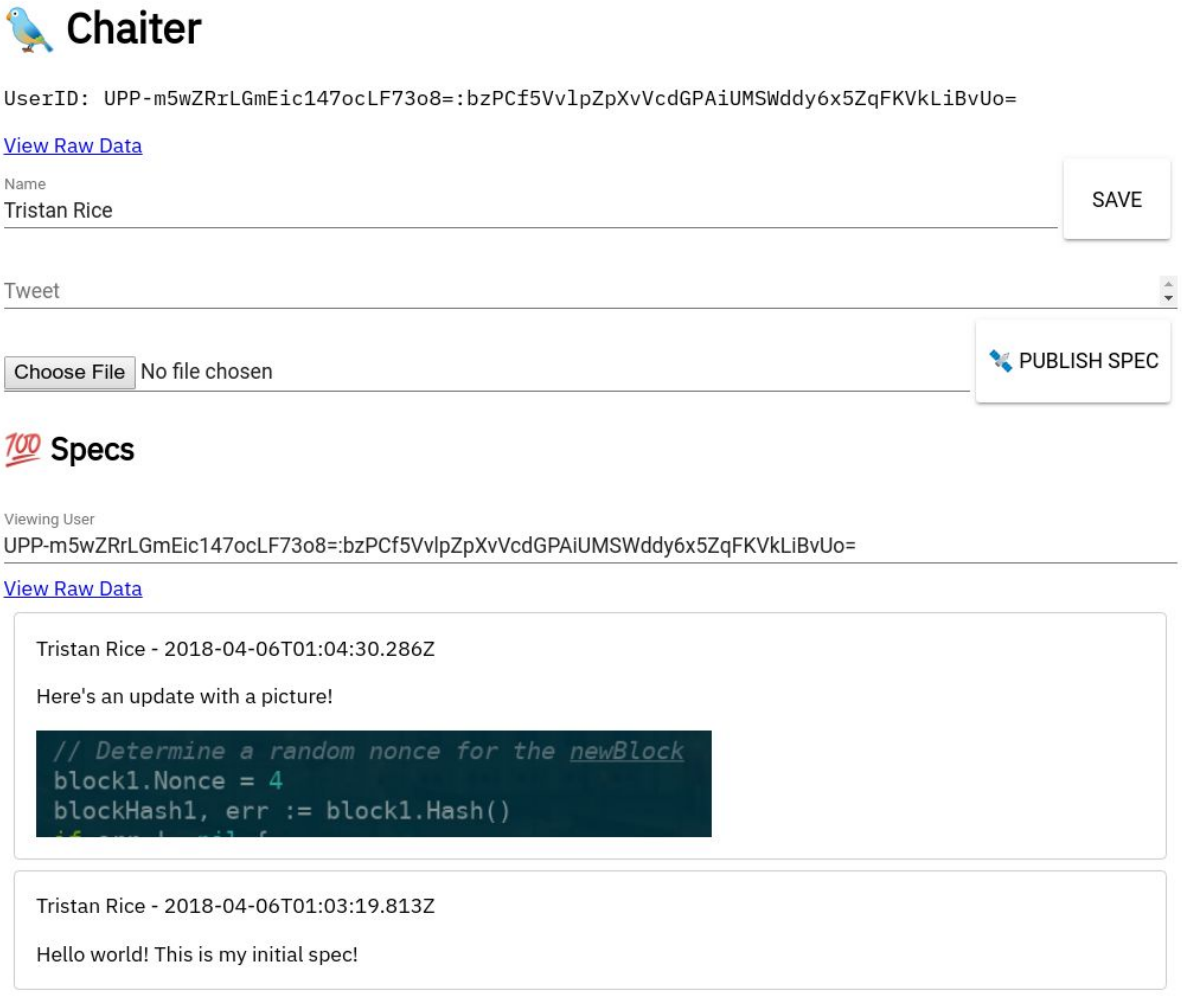
As documents pass through intermediate nodes during a fetch of a document, the intermediate nodes will store the document and a reference to the document in its cache. This reference stores the size of the document that it has cached and the time it was last accessed. Intermediate nodes have a copy of the document and can serve the document. This is advantageous as intermediate nodes act as replicates for documents that pass through them which gives the document more availability.

We evict cached items when the total size of our local items and cached items grows past its configured total size. To select an item to evict, we randomly sample up to 10 documents from our cache and evict the least recently accessed document from our sample. The stored size of the document is used to determine whether we have evicted enough items to be below the total size threshold. Probabilistic LRU caching is faster than LRU caching as we do not need to check the entire cache and it works similarly to general LRU caching Because cached documents may be of different sizes, we continue evicting items until the size of our local storage is below the storage threshold. With this policy of caching, it is possible for a node to have its local data store composed of documents that it added being larger in total local size than our cache threshold total size. In this particular case, this nodes' cache will be always empty.

Handling Failures

To detect failures each node sends heartbeats to its peers. If the heartbeat fails, the node removes that peer and the associated routing tables. The documents and references that node was announcing will eventually disappear from the merged routing tables. In the meantime, nodes will attempt to access that document, but fail due to it being unroutable. Queries that span multiple nodes has a "NumHops" variable that decrements every network hop so false positives in the bloom filters or down nodes can be detected and prevents infinite loops. Queries also use GRPC with timeouts and cancelable contexts so queries can have early terminations.

Example Application: Chaiter



We wrote a simple example application that runs completely on IPFS. Chaiter is very similar to Twitter except you publish “specs” instead of “tweets” and is entirely distributed.

The code is entirely HTML + JavaScript and is hosted as an IPFS document. All data is stored in IPFS. The application generates a ECDSA key pair and uses it to publish a reference as the “UserID”. Other users can then look up the user via that UserID to fetch a list of specs and user information such as name. Images can be uploaded as documents and included in the spec. Other users get notifications about new specs by subscribing to the UserID and displaying the updated specs when they get notified of updates.

The application talks directly to the local IPFS node via grpc-gateway and has access to all of the standard client endpoints.