The Bes' IPFS That's Not A Mess

CPSC 416 Project Proposal

Jonathan Budiardjo, Jinny (Hyojin) Byun, Bryan Chiu, Tristan Rice, Bea Subion

Problem Statement

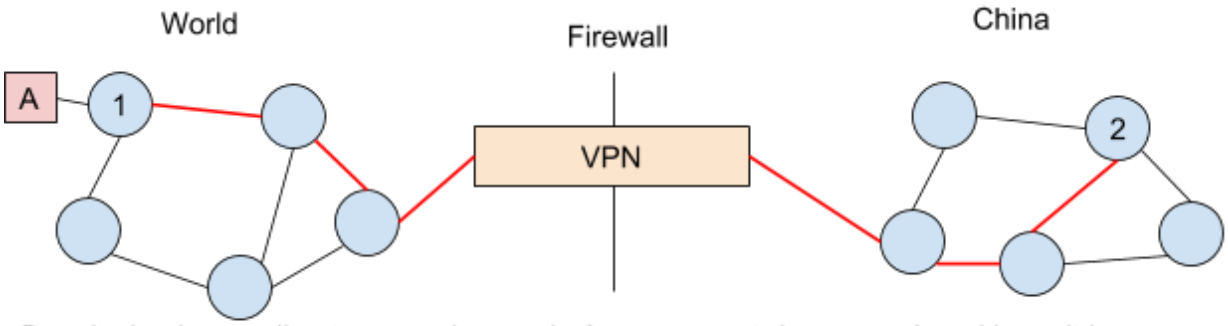
Interplanetary File System (IPFS) is a file system that was originally meant to provide a distributed file system for the dissemination of versioned scientific data. It supports file chunk versioning using cryptographic hashes similar to Git, utilizes a BitTorrent-like file dissemination protocol, and does not require a central hosting 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 makes it very easy to censor files on IPFS. Considering that recently the Catalan government used IPFS to circumvent blocks implemented by the Spanish government, it is likely that tools to censor IPFS content will be created. Making a system that is more resistant to censorship is a useful contribution.

Thus, we seek to replicate 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.

Problem Approach

Network Topology

The network is composed of a graph of nodes who are connected to each other. The network is fully decentralized such that it does not need a central server. The network has the ability to be in a partially partitioned without loss of connectivity. This makes it such that, as long as we have a single node connecting between a sub-graph of nodes, a node is able to access the data of a particular node across the subgraph.



Despite having no direct connection, node 2 can request document A and have it be retrieved from node 1.

Peer Discovery

When a new node starts, it will need to be told of the IP of another node in the system. In an actual production system, a small number of bootstrap nodes would be distributed with the node binary.

After the initial connection to a node, a node can send a `RequestPeers` request and the responding node will reply with a list of peers. Each node will try and maintain a certain number of connections by requesting peers from its existing peers. A node can keep track of potential peers without being actively connected to those. To maximize cross section bandwidth and partition tolerance, each node should try and maximize the diversity of peers it is connected to. For example, if two nodes are connected to each other they should try and minimize the number of overlapping peers while maximizing total number of peers. This does not need to be strictly enforced and probabilistic efforts is likely sufficient. This might be achieved by peer A sending a list of known peers to peer B with prioritization of peers that do not have connections to A.

Documents

All the documents in the distributed file system are referred to by the hash of the data. When creating a new document the hash is returned and to request the document you will need the hash. Once a document is created, it can never be modified. The hash should be cryptographically secure.

Document Structure

```
type Document struct {
      Body []byte
      ContentType string
      Children map[Name](Document or Reference Hash)
}
Document{
      ContentType: “directory”,
      Children: map[string]string{
            “index.html”: “document:YXNrZGZqYWxrc2Q=",
            “foo.html”: “reference:MzgzODM4Mw==",
      }
}
Document{
      Body: “<h1>Hello!</h1>”,
      ContentType: “text/html”,
}
```

To allow for copy-on-write semantics and for directory like structures, each document can have a number of children documents and references. Each child is referred to by a name that corresponds to a hash. This allows intermediate nodes to download and cache children as requested to make things like directory structures more efficient. These documents form a directed acyclic graph (DAG). Cycles are strictly prohibited though they are also enforced by the computational infeasibility to create a self referential document.

References are in the format of `document:<hash>` or `reference:<hash>`. If it is a document, the contents are immutable and the hash is the hash of the contents. If it is a reference, it can be updated to allow for mutability and the hash is the hash of a public key.

Document Retrieval

Since the files are only requested as needed, no single node has a view of all documents or the exact nodes that have them.

One way to solve this is to do a recursive search for the file on all nodes. This has a worst case runtime of at least O(n), where n is the number of nodes, or possibly more depending on how it is implemented.

Another way is to publish indexes of all the files every node has. The naive approach is O(d) memory usage, where d is the number of documents in the system. However, if we use bloom filters, each node only needs O(p*h) where p is the number of peers and h is the node with the furthest number of hops away. Each node publishes a set of `(num hops, bloom filter)` to its peers so when looking for a document you can do best-first search via the bloom filters. Since bloom filters are probabilistic, there may be some false positives but the number of nodes searched is limited by the num hops parameter so it will still be efficient. We will never get false negatives with bloom filters. If we use a fixed size bloom filter across the entire network, we can take the union of two bloom filters to produce the higher num hops filters. This has a slightly higher false positive rate, but should not be a problem given a good initial size.

The size of the bloom filter can be increased as the number of nodes increase via later software upgrades.

Document Caching

This network gets similar performance benefits to something like BitTorrent since intermediate nodes should cache documents that pass through them. Since documents are immutable, there are no cache invalidation issues. Documents should be stored as long as possible and old documents are discarded by using a LRU type approach. This will likely be done by randomly sampling ~10 documents and discarding the least recently accessed. This gives us constant time for cache evictions while still having decent accuracy probabilistically.

Since documents can be discarded by intermediate nodes, documents need to be “pinned” by the source so they are always available.

Document Level Encryption

For censorship resistance and general security, each document is encrypted at the document level. The body, content type, and children are all encrypted. The hash of the original document is used as a symmetric AES key for the document. The hash of the encrypted document is used to identify the document within the system. Only the end client/node should have access to the key so the intermediate nodes will not be able to access the contents.

Deriving the encryption key from the hash of the document without salting preserves the property that two identical documents will always have the same ID. This is intentional since it provides significant caching benefits.

Collision attacks are not an issue since if the attacker already has the contents, there is not much else to be gained. The document children are also encrypted so collision attacks do not reveal anything about other documents. It might be possible to do timing attacks by recording the time at which each document is fetched and correlating them, but if the requester randomizes the nodes they fetch documents from this can be mostly mitigated. No individual node has an overall view of which documents are fetched by which node since the intermediate nodes will cache the documents.

Pseudocode for Encryption

```
key = hash(doc)
doc = encrypt(key, doc)
id = hash(doc)
accessID = id + ‘:’ + key
```

API / User Interface

A node needs a way for a user to add files to be pinned on it as well as retrieving files. Each node will be running an HTTP server with a simple API that can be communicated with either a web browser or a simple command line API. If time permits, we may add a fancy web interface for exploring document hierarchies and peers.

The API will be rest-esque and all parameters and results will use JSON encoding.

Example API Endpoints

`POST /api/document/`

Add a document to the node and pin it. Returns the ID of the document.

`GET /api/document/<id>`
Returns the document.
`POST /api/reference/`
Publishes a reference or an update to a reference.
`GET /api/reference/<id>`
Returns what the reference points to.
`GET /api/peers`
Returns a list of connected peers for debugging purposes.

Stretch Goal: References / DNS-like system

If time permits, we are considering adding a DNS-like system to be able to publish a reference that can be modified. This would rely on public/private key cryptography and use the hash of the public key as the ID. Updates can be published by announcing a change to the whole network signed with the private key. There is a timestamp so the network knows which reference is the latest version. Updates to references are eventually consistent and have no guaranteed delivery time. A practical addition would be to add an expiry time to set an upper bound on reference cache invalidation.

Reference Structure

```
type Reference struct {
	PublicKey string
	DocumentHash string
	Published time.Time
	Signature string
}
```

Stretch Goal: Pub-Sub

We will likely not get to this, but we would allow for publishing on channels via a similar way to the DNS-like system. Channel ID would be the hash of the public key and all messages would point to a document hash and signed by the public key. Message routing would have to be implemented via another tiered bloom filter index so messages are routed efficiently. It is possible to combine the bloom filter index but this might cause a lot more false positives.

Deployment

The system is designed to be platform agnostic and can be deployed anywhere. For demo purposes, the system will be deployed on Azure.

Project Timeline

Deadline	Task	Assigned To
March 9	Submit project proposal	Everyone
March 16	Set up networking infrastructure (peer discovery, connections between nodes)	Tristan
March 16	Add API endpoints	Jinny
March 16	Add command line client	Jinny
March 20	Encrypt documents	Jon
March 20	Add support for adding documents to local node	Jon
March 25	Publish and merge document indexes (bloom filters)	Bea
March 28	Implement LRU document caching at each node	Bryan
March 28	Add modifiable references	Bryan
April 1	Stretch goal: Implement Pub-Sub	TBD
April 1	Stretch goal: Create a web interface to interact with the system	TBD
April 3	Deploy and test on Azure	Everyone
April 6	Final code refinements, submit final report	Everyone

SWOT Analysis

Strengths	Weaknesses	Opportunities	Threats
● Everyone is friends with each other and often hang out in the same student lounge making it easy to stay up to date ● Everyone is comfortable building systems in Go at this point ● At least one member is very proficient in Go and has experience creating integration test environments to test distributed systems	● One member will be out of town and working remotely for the second half of the project ● Various members have other end of the term projects that might interfere with time allocated to this project ● Not all members are familiar with algorithms underlying IPFS	● Learning new algorithms and design patterns ● Learning about testing distributed systems and how to make them robust ● Learning about byzantine systems that are not blockchain	● Complex feature interactions might cause flakiness and be hard to debug ● Hostile governments