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

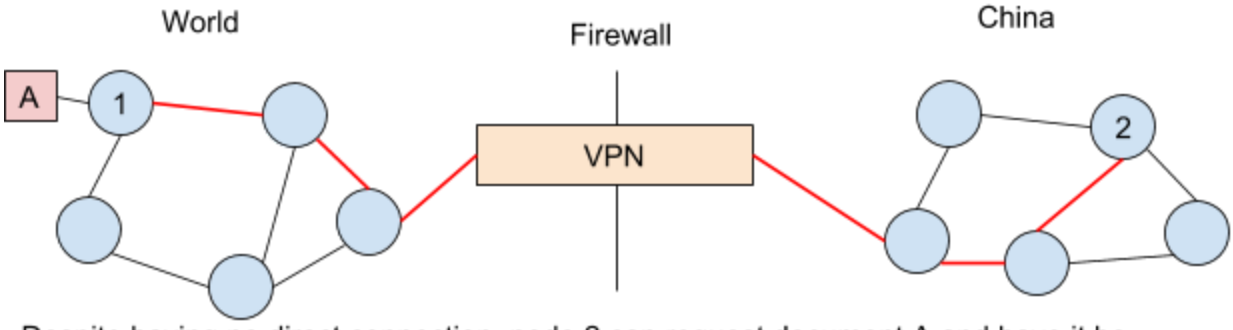
Interplanetary File System (IPFS) is a file system that was originally meant to provide a distributed file system for dissemination of versioned scientific data. It supports file chunk versioning using cryptographic hashes similar to Git, utilizes a BitTorrent-like file dissemination protocol, and does 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 connected state 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 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 nodes starts, it will need to be told of an 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’s connected to.

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 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[string]string
}
```

To allow for copy-on-write semantics and for trees of files, each document can have a number of children. Each child is referred to by a name that corresponds to a hash. This allows nodes to download 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.

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), possibly more depending on how it’s 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 won't ever 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 shouldn't 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 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 won't be able to access the contents. Deriving the encryption key from the hash of the document preserves the property that two identical documents will always have the same ID.

Pseudocode for encryption

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

Stretch Goal: DNS-like system

If we have time, 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.

DNS Entry Structure

```
type DNSEntry struct {
      PublicKey string
      DocumentHash string
      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.

Project Timeline

Deadline	**Task**
March 9	Submit project proposal
	Peer discovery & bootstrapping
	Document adding to local node
	Publishing & merging indexes
	LRU document caching
	DNS entries
	PubSub
April 6	Submit final report

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	● 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