Censorship-Resistant File Storage

David Vargas, Robert Tran, Omar Gonzalez

Summary

This project tackles the problem of designing a file system that is resistant to censorship; that is the deletion, alteration, or distribution of a user's data that is hosted by a third party. Third party and cloud hosting is getting more popular, and finding ways to protect yourself against these kinds of attacks has become ever more important. Our group presents Storj, a decentralized peer-to-peer file storage system that achieves this protection while also allowing for reasonable data availability and integrity. Finally, we discuss Storj Sync (`https://github.com/rtran9/6.857-storj-sync`), a client side tool to increase usability with a relatively new and growing system.

1 Introduction

The majority of popular file storage options today mostly involve the data owner storing their data with a single entity, may that be with the data owner's own local machine or with a cloud service providing third party host such as Google Drive or Dropbox. It is becoming increasingly evident that storing all of one's data locally is becoming infeasible due to memory restriction and risk of hardware failure or damage. So now, the question is, is there any risk or concern for data owners to store their data on cloud servicing third party hosts?

Almost all of these services offer a centralized solution in the sense that a single entity controls one's data. When a user decides to host their data on Google Drive or Dropbox, they agree that this third party host now becomes owner of the data and all the rights that come with hosting that data. From Google's own Terms of Service:

> When you upload, submit, store, send or receive content to or through our Services, you give Google (and those we work with) a worldwide license to use, host, store, reproduce, modify, create derivative works (such as those resulting from translations, adaptations or other changes we make so that your content works better with our Services), communicate, publish, publicly perform, publicly display and distribute such content. [2]

The concern with giving a third party host control of your data in this way comes from the fact that your data at any moment could be deleted, modified, or given to undesirable parties (e.g. government subpoenas) without the approval of the data owner. This is the problem of Censorship-Resistance: to be able to create a file storage system where no single location and no single entity has control over a user's data vulnerable to Censorship.

In this paper, our group presents a Censorship-Resistant File Storage system, as well as a client facing tool we built to help interface with the system. In section 2, we discuss decentralized peer-to-peer networks including other related works that are available. In section 3, we present Storj, the decentralized peer-to-peer file storage service we chose to research. In section 4, we present the tool we implemented to help extend Storj, a file sync for the user. Finally, in section 5 we discuss other future work that could help extend this system.

2 Decentralized Peer-to-Peer Networks

In order to avoid having a single entity obtain full control over a user's data, our group began to look into decentralized peer-to-peer networks. The idea is with each node in the peer-to-peer network being its own entity, and no single entity having special privilege over another, this would remove the power of any single entity having the power to abuse censorship over the user, making the overall system censorship resistant.

2.1 Properties

In its simplest form, a decentralized peer-to-peer network is one that uses many entities (for our purposes, servers) that communicate together to act as a single entity to its user. A decentralized file storage network allows the user to send their data to multiple servers on the network, allowing for replication and sharding of the file. In this way, if any one location becomes corrupted or unavailable for any reason, the user has other hosts to retrieve its data from. Even still, data availability and data integrity are the two hardest problems for decentralized file storage systems to solve, since they need to establish a reliable protocol that allows users to retrieve the same data they stored, as if the network was a single machine. For our project however, not only did we want a system with high data availability and integrity guarantees, but the data also has to be stored securely, private to the user.

2.2 Other Systems

In researching different systems, we came across different related work in the field that could have served the functionality we were looking to achieve. Below is a discussion of a couple of these systems.

2.2.1 IPFS

IPFS is a single file storage system distributed among all the devices in the system. It acts as a cross between a single git repository and the web in some ways. For example, any user could retrieve data posted by other users and could revert to previous versions of data. There are many other features that the system provides, such as strong data integrity by use of Merkle Trees and high availability through replicating data across the networks that could be retrieved by use of a routing network and distributed hash tables. [3]

Although this system satisfied the high data availability and integrity requirement functionality we wanted our system to achieve, it is restrictive enough in read access to fit our security requirement. Since this system is aiming to replace HTTP/HTTPS, it doesn't do as good of a job at acting as a private data storage center. Nonetheless, one could imagine extending this project in such a way that permissions could be enforced on subdirectories to work more towards that Private Information Retrieval goal. Even then, servers would be able to host entire files, which could give the machine the power to compromise Censorship Resistance.

2.2.2 Gnutella

One of the first of its kind, Gnutella demonstrated the utility of decentralized file sharing. The Gnutella protocol allowed all nodes running the application locally participate with the network primarily in three ways:

- Hosting a list of local files to share across the network
- Querying for files that exist somewhere on the network
- Downloading files that exist somewhere on the network from other nodes

While Gnutella was revolutionary for its time, it is not very widely used anymore and falters in the same way as IPFS did. Both systems were created with the intent of making data sharing easier among peers, missing features necessary to make a data owner's content secure to only it's read access from the network. [4]

3 Storj

We then came across Storj, a peer-to-peer cloud storage network. In the subsections that follow, we discuss more about what Storj is, how it performs its main API calls for its users, and why we decided to go forward with this system.

3.1 About

Like other decentralized file systems, Storj is a community driven network. Nodes in the network play one of two roles; they are either renters or farmers. Renters are data owners looking to store their data on the network. Farmers are nodes that offer their devices as storage space for anyone looking to store data. Signing up for a Storj account is quick and new renters are offered 25 GB of free storage. Farmers have incentive to offer its extra disk space to the network since they are compensated for reliable data availability and integrity. Within each API call, Storj is forming contracts between renters and farmers, which are basically agreements that farmers will provide the data reliably while renters pay a very affordable fee for their service.

Storj could be downloaded through NPM (the node package management utility) and installed directly on the user's machine. All interactions with the Storj network come through the command-line tool. Users could create "buckets" on their account and store their data on the desired "bucket". Buckets do not correspond to servers; they are simply an abstraction mechanism to help users organize their data on the network.

3.2 Upload

As previously mentioned, a user could designate multiple buckets on their account, each of which has its own bucket id. In order for a user to upload a file to the network, they must specify the target bucket id and the file they want to upload.

After receiving the file, Storj encrypts the file with the user's public key using AES and then shards the file into n encrypted pieces. These shards are then each replicated and stored on the various farmers' nodes throughout the network.

It is in this uploading procedure where the strength of the system's censorship resistance comes through. For any given file the data owner owns, it is split throughout the network, so no single node has full ownership of a file they don't have proper access-rights to. Additionally, each shard is encrypted, so no node could read any part of a user's data unless they compromise the data owner's private key (which for each user is created by a pass phrase).

3.3 Download

The ability to effectively encrypt, shard, and scatter user data throughout the network would be useless if there wasn't an algorithm in place to be able to reliably retrieve data a user stores on the network. Storj provides this capability with an API call to download a file, as long as the user provides a bucket id and file object id (there are also API calls where the user could get all of their bucket id and file id mappings).

Once Storj has received the request to download a file for the user, it starts collecting all n shards from the servers that they were stored on upload. If any shard fails to return or is corrupted, then Storj simply requests one of the other servers that has that shard since each shard was replicated.

Once it has a copy of each of the n shards, it runs a Merkle Proof on the shards to verify that the file was uncorrupted. When the file was first uploaded, a salt s_i was saved for each shard, and these salts are reused to calculate $H(shard_i + s_i)$ to create a leaf in the tree. Then the Merkle proof is run all the way up to the root, and the root must be verified. After the file has been verified, Storj combines the shards back together to recreate the original encrypted file. Now that the encrypted file has been recreated, Storj could simply use the private key stored on the client to decrypt the file and store it on the data owner's device. After trying out the API, we verified that our group was able to store and retrieve files successfully in the Storj network.

3.4 Strengths & Weaknesses

Not only was Storj able to achieve data integrity and availability on the same level of other researched decentralized peer-to-peer networks, it also demonstrates strong Censorship-Resistance by having many encrypted shards of data throughout the network. The following will be a discussions of other strengths and weaknesses we found with this distributed file storage.

3.4.1 Strengths

Besides the censorship-resistance property, Storj is also resistant to a number of other types of cyber security attacks.

- Identity Hijacking - The threat of a node stealing/mimicking another user's node id. This attack is well protected by requiring the user's passphrase on every API call to the Storj network. Since the adversary will not have the user's passphrase, the adversary cannot access any of the user's content.

- Sybil - The threat of a large number of nodes in the network colluding to disrupt the network, say by dropping messages or sending malicious ones. This attack is well protected by replicating information on numerous nodes. Therefore, the probability that the adversarial collusion could take over enough of the network to make an impact on a user is very small.

- Faithless Farmer - The threat of a data host taking data from one data owner and giving it to other users on the network. This attack is well protected because even if another user receives a shard it was not supposed to receive, the nature of the shard being a fraction of the data and being encrypted makes it almost impossible for adversary to recreate the original file.

Another strong feature that is currently implemented on the Storj is the ability to stream both audio and video files from the Storj network. A user, as opposed to keeping these files stored on his own device, could keep it in the Storj network and listen/watch the file on media players like Mplayer or VLC.

3.4.2 Weaknesses

Storj is very new in development, with the group's white paper posted on December 2016 and remains unpublished. Because of this, there is still a lot of room for growth for the system and many features missing from the current implementation. In trying out Storj for ourselves, we encountered the following in which we thought to be weaknesses of the system:

- Public Sharing - Although the system prevents unauthorized access reliably, the feature to allow other trusted parties have access to a user's bucket and/or files is currently unstable. There are use cases where as a data owner, granting push or pull access to other data owners would be useful.

- Command-line only - There are no stable client tools out right now for Storj to interface with the network in a more user friendly way as opposed to command line inputs. Details such as inputing a bucket id and inputing a file id should be abstracted away from the user.

- Overwriting Files - Currently a data owner cannot edit a file on the Storj network. To update a file, one must download the file, make said changes, delete the file on the network, and then push the new file. At the very least, the option to replace the existing file would be useful.

- Compromised Data Owner Attack - There is no defense in place if an adversary gained access to both a user's node id and private key. This in of itself is a hard attack to protect against in a distributed system, and a worthwhile trade off for all the other attacks that Storj defends against.

Upon experiencing these weaknesses, our group thought what could we build on top of Storj to provide an easier to use distributed file storage experience. We felt we could most improve the second and third weaknesses in building a client side file sync.

4 Storj Sync

Drawing inspiration from Dropbox sync, which constantly keeps a user's files on Dropbox in sync with ones that the user is editing locally, our group wanted to extend our final project research in building a similar tool on top of Storj. Storj Sync aims to allow users to easily sync their files to Storj network and keep up to date after edits. It also aims to allow users to revert to previous versions of a file if they feel the current version is corrupted for any reason or because of user error.

The code is available publicly at `https://github.com/rtran9/6.857-storj-sync`.

4.1 How To Use

Storj Sync could be installed using npm after cloning the repository. The instructions are as follows:

```
$ git clone <repo>

$ npm install -g
```

```
$ storjsync register
```

This triggers a prompt for the user to create an account with Storj, asking for an email and password. Upon completion, the user would need to confirm this with an email.

```
$ storjsync login
```

The user logs into the storjsync system. This creates a key pair. The private key is kept locally, and the public key is sent to Storj.

The user is now ready to sync folders.

```
$ storjsync init-sync
<file-directory>
```

This command configures the folder to be synced to Storj and initiates the first upload of all the files contents. The user will be asked for a passphrase for a keyring if this is the first time the user is uploading a file.

The user may manually sync the folder with the following command:

```
$ storjsync sync <file-directory>
```

This syncs any local changes to what is being stored in the Storj network.

The user can also create snapshots.

```
$ storjsync snapshot
<file-directory>
```

This creates a backup of the entire directory, which may be downloaded at any time in the future.

Users can list the snapshots that are available for the synced folder using:

```
  $ storjsync list-snapshots
<file-directory>
```

This returns a list of timestamps of when the snapshots were created.

The user can download the snapshot to recover a previous version of the file with:

```
  $ storjsync download-snapshot
<file-directory> <timestamp>
<output-path>
```

4.2 How It Works

4.2.1 The Initial Sync

Upon logging in, storjsync will keep the private key and keyrings in a folder in the home directory:

```
~./.storjsync
```

Any time the user syncs a new folder, they will be using that private key.

On the initial sync, a directory is created within the specified directory to be synced at

```
~/synced-folder/.storjsync
```

This is where storjsync keeps information specific to the folder to track changes. This directory is more or less hidden. Nothing in these files will be uploaded to Storj.

Next, the first sync creates a table at

```
~/synced-folder/.storjsync/main.table
```

Then storjsync will begin to recursively upload all the files in the directory. Files are tracked by their filepath. The table keeps track of where the files have been uploaded to in Storj and metadata about the file. Most importantly, it keeps a last modified field on the file.

4.2.2 Syncing Changes

When syncing, storjsync explores the entire local directory. For each file, it looks at the last modified timestamp and compares it to the table. If they are different, then storjsync will delete the current version of the file in Storj then upload the new version. The table will be updated to include the new modified timestamp.

If there are any new files compared to what is in the table, it will upload them to Storj.

There is also the case where files are deleted in the local directory, in which case there are files in the table that were not found when exploring the directory. In this case, storjsync will delete these files in the Storj network.

4.2.3 Snapshots

Snapshots are created by re-uploading the directory into a new location in Storj. We manage these by keeping a folder of snapshots in

```
~/synced-folder/.storjsync/snapshots
```

Similar to how a table keeps track of files for syncing, snapshots are also tracked by a table. A separate table is created, named by a timestamp, into the snapshots directory. Storjsync can show the user which snapshots are available by looking in the snapshots directory. The snapshot table contains all the information necessary for the user to be able to download the snapshotted folder.

4.3 Results

We built on top of the open-sourced Storj library and CLI (command-line interface). We expanded the Storj CLI to include the functions we wanted in the storjsync CLI. The current status of the project doesn't include any processes that automatically sync. Performing an operation like syncing every hour was very specific to the operation system so we refrained from doing so.

Also, to use Storj itself right now isn't free. If a user registers for an account, they will receive 25 Gb of free space. Though, a user could accommodate this by Storj farmer.

Additionally, a user cannot download their files from Storj on another device unless there exists a copy of their private key and keyring on that device. If the Storj team develops a more efficient way for users to retrieve their files on any of their devices, storjsync could be able to help Storj become more like Dropbox and Google Drive, where users can comfortably store their files on Storj and download or stream their files on any of their devices.

5 Future Work

As previously mentioned in section 3.4 during the discussion on Storj weaknesses, Storj as a system has a lot of room to grow both within the system and with user applications built on top of Storj. The following are other possible areas of future research to help improve the Storj system.

5.1 Secret Sharing

Currently, to regenerate the encrypted file on download, Storj requires the retrieval of **all** n shards to create the file. Granted, it only needs to find one copy of each shard, but one could imagine a secret sharing algorithm that sends n shards and only requires copies of k shards where $k < n$ to recreate the encrypted file. The Storj team mentions implementing this feature at some point in the future using Erasure Coding. Shamir's secret sharing would also be able to work in this system and it would be interesting to investigate which of the three (Shamir, Erasure, or none) produces the most reliable results.

5.2 Reputation System

As previously mentioned, Storj implements a contract system in their upload API calls to match renters (data owners) to farmers (data hosts). As part of this contract system, it would be useful to know which members of their network were not reputable (corrupting own data, avoiding payment, etc.) so that it could assign contracts accordingly. A reliable distributed reputation system is still an active area of research and Storj would greatly benefit from an effectively reliable one. [1]

6 Conclusion

In trying to produce a censorship-resistant file storage system, our group directed towards building on top of a decentralized peer-to-peer file system. Storj was able to provide very reliable and reasonable data availability and integrity guarantees, while also providing a secure enough uploading and downloading protocol to make the system very censorship-resistant. The early life of the project allows great room for growth, which led our group to building a client file syncing tool.

References

[1] S. Wilkinson, T. Boshevski, J. Brandoff, J. Prestwich, G. Hall, P. Gerbes, P. Hutchins, C. Pollard : Storj. Second edition. https://storj.io/storj.pdf.

[2] Google Terms and Services. https://www.google.com/policies/terms/.

[3] J. Benet: IPFS - Content Addressed, Versioned, P2P File System. Third Edition. https://github.com/ipfs/papers/blob/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf.

[4] J. E. Berkes: Decentralized Peer-to-Peer Network Architecture: Gnutella and Freenet. 9 April 2003. http://www.berkes.ca/archive/berkes_gnutella_freenet.