# A Peer-to-Peer Personal Storage System

Adrien Ecoffet UC San Diego
Email: adrien.ecoffet@gmail.com Fabrice Bascoulergue UC San Diego
Email: f.bascoulergue@gmail.com

*Abstract*—**Personal storage services have become increasingly popular in recent year, with services like Dropbox, Google Drive, Microsoft Skydrive or Apple iCloud[1], [2], [3], [4]. Because these services, by and large, are used to synchronize personal files across devices rather than to increase one's available storage space, It appears possible to devise a system in which users would share some of their existing disk space to be able to store files in the "cloud" for free. We propose a purely distributed peer-to-peer solution to that problem, and examine some of the challenges such a solution must face. We then present Jellyfish, our prototype for such a system, and the results of performance tests on a small cluster of Jellyfish instances.**

## I. Introduction

File synchronization services are more popular than ever today. This is in large part due to the increasing number of computing devices people have in their possession. On the other hand, commodity hard drives can now store huge amounts of data (typically 500GB to 2TB). Those two facts make it possible to design a system that would use unused space on user hard drives to provide an amount of cloud storage space proportional to the amount shared to the network by the user.

The system would store encrypted versions of the files on other users' filesystems so that files can be retrieved at any point in time by the user.

### *A. Related Work*

Some similar systems have already been proposed. Clearly this somewhat resembles some distributed file systems, especially those that support disconnected operations, such as Coda[**?**], but some even more similar commercial services have also been created.

The first of them was probably Wuala[5], which initially featured a way to increase one's available storage space by sharing some disk space. Note that all files were still stored on their servers and that this was only used to increase performance. The feature has since been dropped.

Another example is SpaceMonkey[7], which allows users to rent large hard drives that are constantly connected to the Internet and are part of a peer-to-peer network which stores other user's files.

Very recently, Bittorrent announced Bittorrent Sync[6], which is very similar to our proposal. The differences we have seen are the following: Bittorrent Sync doesn't have a notion of an user account, rather users are expected to share a secret key across their devices. Additionally, Bittorrent doesn't ask the user how much of their hard drive it should share. It is unclear exactly how it determines that. Bittorrent Sync is also tracker-based, while our proposal is fully decentralized. Most importantly, Bittorrent hasn't released any paper or source code that we know of so far.

A company called Maidsafe is apparently currently developing a peer-to-peer system very similar to ours[8], but it hasn't been released so far. However, Maidsafe has developed and released an implementation of the Kademlia distributed hash table[9] which we use in our implementation (its convenience is understandable since it was developed for this purpose exactly).

## II. Efficient storage

It should be quite clear that the chances of failure of any given node are quite high in such a network, since real world computer users generally don't have their computers turned on at all time. As far as metadata is concerned, we choose to use an existing distributed hash table implementation (in our case Maidsafe-DHT, an version of Kademlia). However, it would be impossible to store files, especially large ones, directly in the DHT, partly because of performance reasons (we want to get several parts of the files from different nodes so as to maximize network performance), and mostly because replication in DHTs isn't suited to large files: our particular implementation stores 4 instances of any value in the DHT and proactively transfers the value to other nodes when one of the nodes involved in storing it goes down. Besides, Maidsafe-DHT is a fully in-RAM implementation, so it couldn't sustain a large amount of data.

There must therefore be a ratio between the space a user allocates on her hard drive and the amount of data she is able to store on the network, with the former value being likely higher than the latter. What follows is a set of techniques that can be used to bring that ratio as close to 1 as possible.

## A. *Avoiding Best Fits*

A common technique employed by companies in the personal cloud storage industry is to make it hard for users to get an amount of storage space that actually fits their needs. For example, Dropbox proposes either 2GB, 100GB, 200GB or 500GB of storage for individuals[1], which makes it hard to find a type of account that actually fits the exact need of an individual user. What this means is that users who are underusing their Dropbox account are actually paying for the ones that are using theirs to its fullest.

We believe it is useful to make sure that users can only choose their amount of storage space from a set of exponentially growing possible sizes. This way the network would be able to do more replication than seemingly implied by the ratio, because the network will be underused.

Data released by Dropbox in 2011 indicates users were storing on average 400MB in their Dropbox[**?**], which tends to confirm that using exponentially increasing storage options is a good way to get the service to be underused.

## B. *Compression*

Compression seems like an obvious solution, although in practice it might not be that useful since users are likely to store compressed media files such as MP3 audio files, or even PDF files. These files do not compress very well at all, however, it seems that compressing them using gzip will still often make them 1% or 2% smaller[11], so compression certainly doesn't hurt, which is why we included it in our pipeline.

Gzip compression is commonly used in other Internet transmissions and is currently much faster than the typical bandwidth of a personal computer, so it won't be a bottleneck.

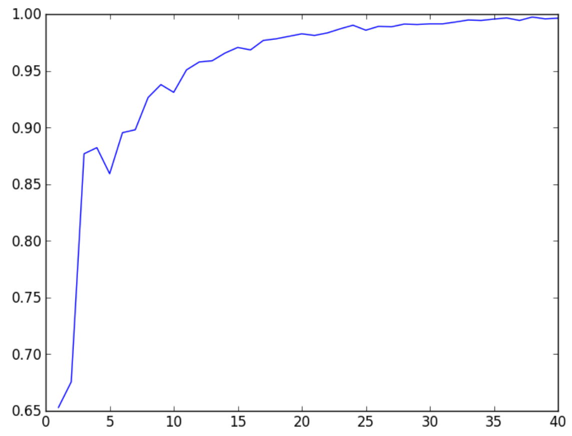

Fig. 1. Simulated probability of recovering file as a function of the number of nodes: 10 parts, 20 codes, 66% uptime

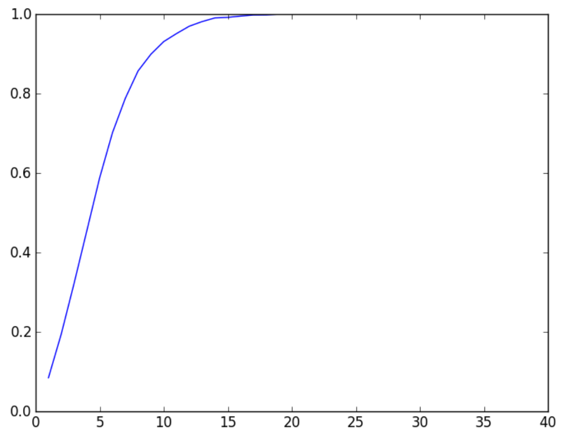

Fig. 2. Simulated probability of recovering file as a function of the number of codes: 10 parts, 66% uptime, 100 nodes

## C. *Erasure Coding*

Erasure codes (such as Reed-Solomon codes) allow for generating $n$ parts and $m$ error codes and for reconstructing the file if any $n$ of the $n + m$ parts are fetched from the network. This greatly increases the probability of retrieving a file since, for sufficiently large $n$, the replication ratio $n/(n + m)$ only has to approximate the probability of any given node being online.

We use the Jerasure implementation of Cauchy Reed-Solomon codes, which seem to be the fastest open-source implementation of general purpose erasure codes[12].

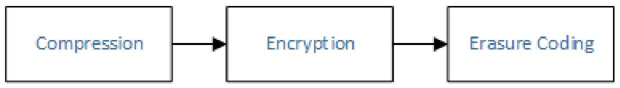

Fig. 3. Our chosen pipeline

### D. *Pipeline*

Several pipelines are possible for this system. The elements of the pipeline would be compression, erasure coding and encryption. Clearly encryption cannot take place before compression because encrypted data almost never compresses at all. It is possible to compress Cauchy Reed-Solomon codes to some extent if they weren't created from encrypted data, but experiments we made show that they compress significantly less well than the original file, so clearly compression should be the first step. There is no significant difference between encrypting after creating erasure codes or before that, apart from the fact that the total amount of data to encrypt will be significantly higher if erasure coding is done first. This leads us to our final pipeline (shown in Figure 1): compression, then encryption, then erasure coding.

## III. Threat Model

A major issue in a decentralized system like this one is that of security. Here we consider the security of our design against three types of attackers.

### A. *Eavesdroppers*

Eavesdroppers want to access user data, i.e. steal user files, steal user accounts, get some knowledge of exactly what files a given user has etc.

We simply use the typical way to protect oneself against eavesdropper: cryptography is used extensively in the system. User are each assigned a RSA private key which identifies them. The vast majority of operations require a signature from the user. Users also have a single AES256 key which is stored in the distributed hash table, encrypted using the user's public key (this is because AES cryptography is much faster than RSA). All files are encrypted using this AES key and a separate random IV. Some private user data is stored that way as well. Note that a separate key could also be used for each file, though this would incur a slight performance penalty.

Because we want to provide the abstraction of a user account rather than require users to transfer their private keys across their devices, private keys are actually store in the DHT, encrypted using an AES256 key that is derived from the user ”password” using PBKDF2[13] (note: a better alternative for a production system could be scrypt[14], but PBKDF2 implementations are more widely available). This clearly makes the system vulnerable to dictionary attacks on passwords, so a real system should probably have a password strength verification module and should educate users about the importance of a strong password, especially in this context.

Note that an alternative would have been to generate the private key directly from the password using PBKDF2, but that would have made it extremely hard for users to change their passwords later.

Additionally, the DHT we chose to use uses public key cryptography to ensure that, for instance, nodes which do not have a user's private key are unable to delete or modify values that have been set by that user.

### B. *Cheaters*

Cheaters are economically motivated attackers.

An important concern in such a system as ours is the possibility of having users that use the file synchronization service without actually sharing the right amount of space to the network. The two main ways to do this would be to only connect to the network to perform actions such as storing and downloading files, or simply not actually storing the file blocks one is in charge of.

*1) Karma:* Misbehaving nodes accumulate karma. When a node finds another node is misbehaving, it adds a karma record to the node's public karma set. Karma records exist for a limited amount of time (30 days in our current implementation) and contribute to diminishing the node's contribution to the user's overall allowed amount of storage.

For instance, if a node is found not to be connected at a given time, a NotConnected karma record will be added to it. These NotConnected records are then used to estimate the uptime of the node during the last 30 days, and the node's contribution to the overall allowed storage space for the user will be proportional its uptime.

As another example, if a node is found to be connected but fails to prove that it does indeed store a good version of a given file part, a DeletedPart karma record will be associated with it. While it is normal for a node to be disconnected at some point, deleting a file part is considered explicitly malicious behavior and is thus harshly punished. In our implementation, three DeletedPart records within a month bring the user's allowed storage space down to 0. Unlike with disconnected nodes, the specifics of the punishment don't matter too much here, as long as they create a strong incentive to keep the file blocks.

An issue with this system is that these errors are ordinarily detected only when a client tries to download a file, which is a fairly uncommon event. For this reason, every online client will regularly keep a check on their file blocks by doing the following: when a file is added, the client generate hashes for each file block using different random salts. Then the client will regularly require nodes that store blocks from its files to compute the hash of one of their blocks using a salt they have never seen before. If the node computes a correct hash, it can be assumed that it still has the file block. If the client runs out of precached hash challenges it can always recreate a few from the files it still has locally.

### C. *Byzantine Attackers*

Byzantine attackers could exploit issues with the system for no apparent reason. One such issue is the Sybil attack[**?**], but it is definitely out of the scope of this project.

Another thing a Byzatine attacker might do is trying to delete data from the DHT. Fortunately, Maidsafe-DHT will allow such operations to be performed only by nodes which have the same private key as the node which initially set the value in the DHT, so this would be impossible.

Yet another is to add karma records to other nodes to reduce their maximum storage space. Unfortunately we haven't found a way to prevent this issue that would still make it possible to catch misbehaving nodes and that would be efficient, since any such method would require the help of a third party.

## IV. Future Features

### A. *Duplicate Merging*

### B. *File Sharing*

### C. *Tests*

## V. Conclusions

This paragraph will end the body of this sample document. Remember that you might still have Acknowledgments or Appendices; brief samples of these follow. There is still the Bibliography to deal with; and we will make a disclaimer about that here: with the exception of the reference to the LaTeX book, the citations in this paper are to articles which have nothing to do with the present subject and are used as examples only.

## VI. Acknowledgments

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

## References

[1] http://dropbox.com

[2] http://drive.google.com

[3] http://skydrive.live.com

[4] http://icloud.com

[5] Mager, Biersack, Michiardi, *A measurement study of the Wuala on-line storage service*, 2012

[6] http://labs.bittorrent.com/experiments/sync.html

[7] http://spacemonkey.com

[8] http://maidsafe.net

[9] https://code.google.com/p/maidsafe-dht/

[10] Houston, *Dropbox startup lessons learned 2011*, 2011

[11] http://bashitout.com/2009/08/30/
Linux-Compression-Comparison-GZIP-vs-BZIP2-vs-LZMA-vs-ZIP-vs-C
html

[12] Plank, *A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage*, 2009

[13] IETF, *PKCS #5: Password-Based Key Derivation Function 2 (PBKDF2) Test Vectors*, 2011

[14] Percival, *Stronger Key Derivation via Sequential Memory-Hard Functions*, 2009