Blackbox: Distributed Peer-to-Peer File Storage and Backup

Payut Pantawongdecha
Isabella Tromba
Chelsea Voss
Gary Wang

Instructor: Ron Rivest

May 14, 2014

Abstract

*This paper presents the design of **Blackbox**, a distributed P2P file storage and backup that ensures security, fairness and reliability. **Blackbox** works as a client on a computer that has an access to the internet but doesn't need a centralized server. The client assumes that a fraction of users are adversaries who might abuse the system, but still promises a secure and uninterrupted operation.*

*Key aspects of **Blackbox**'s design include applications of cryptographic schemes to ensure security, network structure among clients to ensure reliability of connections, and rules for fairness.*

Introduction

Blackbox is a peer-to-peer distributed backup file system. Backing up files is an important task for any computer user, to make sure that data is not lost in the event of disk crash or other accident. Services like Dropbox, Carbonite and CrashPlan offer backup space, but often at monetary cost. On the other hand, each computer user usually has plenty of free disk space. A survey conducted by Butt et. al. [5] before 2004 shows that there are 8GB to 60GB available disk space on 500 instructional machines at Purdue University. Moreover, for systems with more than 40GB disk space, over 90% of the local disk space is unused. Since disk space has become cheaper over the years, the amount of free disk space today is proportionally higher. In this project, we design a backup file system that allows users to store files on their peers extra disk space, achieving fault tolerance without requiring users to pay a third-party service.

We begin by describing the set of goals that we want our client to achieve. These goals will ensure that ***Blackbox*** is safe and reliable for users to backup their crucial and secret data. We utilize a wide range of concepts from common cryptographic schemes in order to ensure the security and confidentiality of users data while using ***Blackbox***. Our system hopes to achieve the following:

1. **Confidentiality**
   When files are stored in another computer, other users must not be able to obtain any information about the file except some primitive data such as the file size, even though the file is on their computers.
2. **Integrity**
   Files should not be able to be modified by others. If they are, then the owner must be able to detect it and will not accept the files. This feature can prevent malware to enter another user's computer.
3. **Reliability**
   Users should be able to obtain the backup of their files with reliability. Practically, peers are not online indefinitely and adversaries exists, but users should be able to obtain their files within reasonable time window.
4. **Fairness**
   The amount of space that users can backup their data to should be the same as the amount of space they allow others to store data on. This problem goes beyond just specifying the amount of space. It also involves analyzing individual online time and resolving potential problems such as hard disk drive failure where a user lose all data from their peers, but still want to retrieve his backup.

There are related projects and products that are related to ***Blackbox*** such as *Freenet [2]*, a peer-to-peer platform for censorship-resistant communication, *BitTorrent Sync* [1], a client for syncing data between multiple trusted peers who share a secret key, *Tahoe-LAFS* [3], a distributed file system which stores files on multiple computers to protect against hardware

failures, and *PAST* [4], a large scale, persistent peer-to-peer storage utility. However, we still cannot find a peer-to-peer distributed backup file system that satisfies all of our goals. For example, *Tahoe-LAFS* has an unclear policy on “fairness.” In response to the shortcomings of previous systems, we propose ***Blackbox*** as a secure, fair, and fault tolerant distributed peer-to-peer file system.

First, we will describe the methods for file storage and encryption. Next, we describe the structure of the decentralized network of peers. Finally, we conclude with protocols and tactics for preventing leeching of space by untrustworthy users.

References

[1] http://www.bittorrent.com/sync
[2] http://en.wikipedia.org/wiki/Freenet
[3] https://tahoe-lafs.org/trac/tahoe-lafs
[4] http://research.microsoft.com/en-us/um/people/antr/past/hotos.pdf
[5] Butt, A. R, Johnson, T. A., Zheng, Y., Hu & Y. C. (2004). Kosha: A peer-to-peer enhancement for the network file system. ACM/IEEE SC2004: High Performance Computing, Networking and Storage Conference, November 2004. (http://people.cs.vt.edu/butta/docs/sc04.pdf)

File Storage

File Redundancy Scheme (Reliability)

In order to achieve a reliable file distributed file system, we need a way in which to split files in such a way to ensure that not all nodes that are storing our data are required to be online at the time we request a file. There are two main ways in which to achieve this goal. The first is the most straightforward which is a simple replication scheme. The second scheme involves a concept known as erasure coding. We begin by describing erasure coding.

Erasure Coding

The benefit of erasure codes is that they provide redundancy without strict replication. Erasure coding works by taking a file and splitting it first into m chunks. Then, those chunks are used to create *n* total chunks (*n* >*m*). The resulting erasure coding scheme is an m of n scheme which means that only m chunks are needed in order to recover the entire file. Because we do not want to rely on the peer having enough space for a large file, we always have a constant chunk size. So m, the number of chunks needed to recover a file, depends on the size of the file.

> For instance, a 10MB file could be split into 500KB size chunks. We want to split the file into a chunk size that reduces the number of superfluous ‘padding bits’ needed. This means that the chunk size will depend on the file size. So with a 10MB file and “chunks” of 500KB results in a 20 chunks for a particular file. Alice will then specify the amount of redundancy that they would like for her file. The higher the redundancy, the less peers she needs online to recover an entire file, but also the less space that she has available to store other files on the system. For instance, if the 20 chunks are then converted into

50 chunks and stored throughout the system, only 20 of these peers must be online at a particular time in order for Alice to recover her entire 1GB file.

Note: Nychis et. al. [2] pointed out that a file cannot be decrypted until all *n* out of *m* chunks of the file have been recovered. The reason for this is that the file is encrypted before the chunking phase. So in order to decrypt, the chunks need to be reassembled (via an interpolation algorithm that works similar to polynomial interpolation as presented in Adi Shamir's secret sharing scheme).

Replication versus Erasure coding [2]

A study by Rodrigues and Liscov [3] compared erasure coding and replication schemes in peer to peer networks and found that replication was a better scheme if peer availability is high, whereas in networks where peer availability is low, erasure coding is preferred.
In a study by Weatherspoon and Kubiatovicz [4], they found that erasure coding results in an order of magnitude less bandwidth and storage as compared with replication.

Overall, the academic literature seems to agree when it comes to the advantages of using erasure coding as opposed to replication in P2P file systems where nodes are unreliable (unavailable), whereas replication is preferred in networks with high node availability.

For these reasons, we chose to go with an erasure coding scheme. This will allow our system to store more data at a lower storage cost. Because our system relies on available disk space as opposed to a distributed file system in a data center which has SSD dedicated to file storage, lower storage costs are crucial for the usability of ***Blackbox.*** The main complication with erasure coding comes into play when a node fails. We discuss that in the next section.

Node failure and redistribution of the Erasure Coded Chunks [1]

Suppose we have 10 chunks of a file distributed across the network and only 8 of these chunks are needed to recover the file. Now, 2 nodes leave the network. We now rely on all 8 of these nodes to recover our files. Even worse, suppose 3 nodes leave the network. Now, we do not have enough redundancy to recover our files. The way in which our system detects nodes leaving the network will be described in later sections. Here, we describe the 'chunk' node failure protocol to ensure that we maintain the invariant that 8 chunks are needed to recover a file.

The main drawback with erasure codes is that in order to regenerate a chunk when a node leaves the network, you need the entire file. There are several approaches that we considered:

1. Alice- whose file chunk was lost is notified that one chunk was lost and now she must recompute a chunk to send out to a new peer

2. The nodes in the network handle the re-chunking. i.e. when a node leaves the network, the remaining nodes are responsible for chunk redistribution

The first approach is the naive approach. Alice simply receives a notification that she must re-generate a chunk. Assuming that ***Blackbox*** operates a file backup system as opposed to a remote file store, she can easily recompute a chunk and send it out to the new peer.

The second approach does not require Alice to store the file on her own computer. This allows for more flexibility in our design. As mentioned previously, the entire file needs to be recovered in order to generate a new chunk. When a node leaves the network, a notification is sent to all nodes that this file has disappeared. Then, nodes with available storage agree to store a chunk from a file that is now below its necessary chunk count.

The following occurs on the node that agrees to store an additional chunk:

1. The node will broadcast a request for all other chunks that combine with the chunk it has, to reconstruct the file.

2. Once the node reconstructs the file, the node computes a new chunk for themselves and then deletes all other chunks it collected.

A notable problem with this strategy is the huge overhead in bandwidth. All chunks of the file must be sent over the network for all files that a particular node was storing. Consider the following scenario:

For a node that stores 10GB (a reasonable assumption given that average computers have 10-60GB available storage space) lets say that the average chunk size is 100KB, so the node stores 100,000 chunks (each chunk corresponding to a file). So now 100,000 files need to be recovered. Lets assume a file is 500KB (all text files). The loss of this node causes an additional 6GB of data to be sent over the network.

Because we assume that node failure is infrequent, the increased bandwidth caused by re-chunking is not crippling for the system.

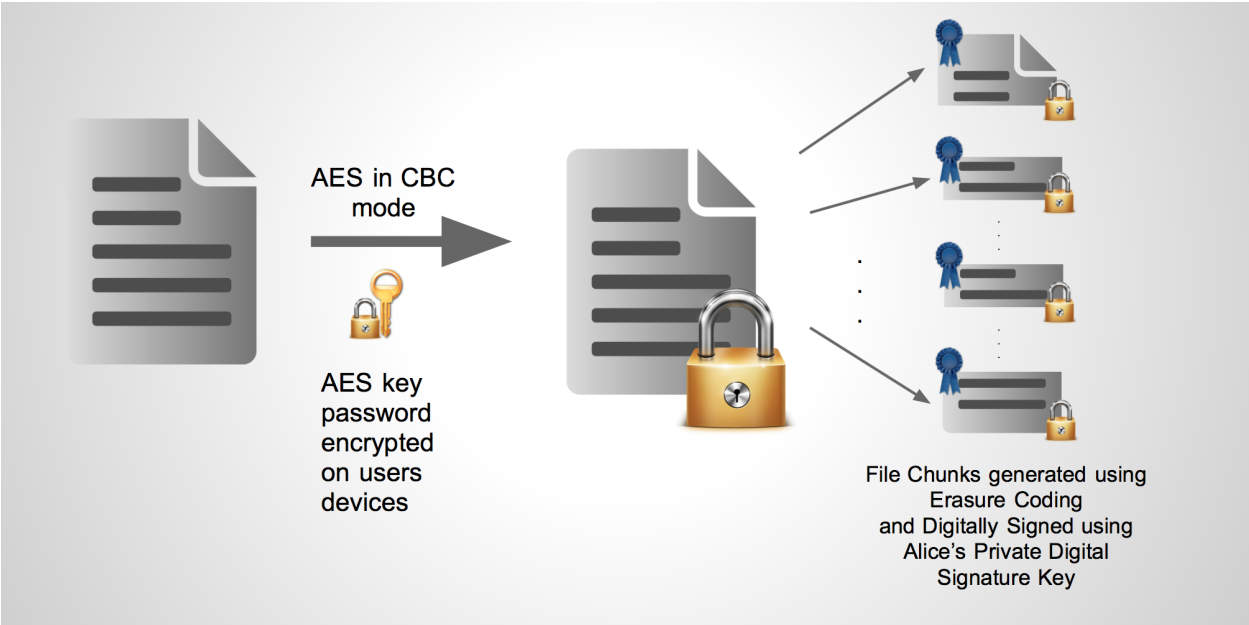

Figure 1. Encryption of a file and subsequent *chunking* using Erasure Coding

Chunk Validation (Integrity)

Each chunk is digitally signed by Alice to ensure that no malicious party has altered the contents of any chunk. Each peer that receives a chunk from Alice will verify that the chunk is in fact from Alice by verifying that the digital signature is correct.

In order to ensure that all chunks sent out were not modified in transit or maliciously changed by an adversary, we use a digital signature scheme. Each chunk that is sent from Alice's computer is digitally signed. Upon receiving a chunk, a peer will verify that the chunk is in fact from Alice by verifying that the digital signature is in fact Alice's. The public keys for each participant in ***Blackbox*** is publicly available. Of course the digital signature must be used to verify the 'encrypted' form of the chunk because your peers will never actually know what the original files contents are. This means that Alice will first encrypt the file and then sign it.

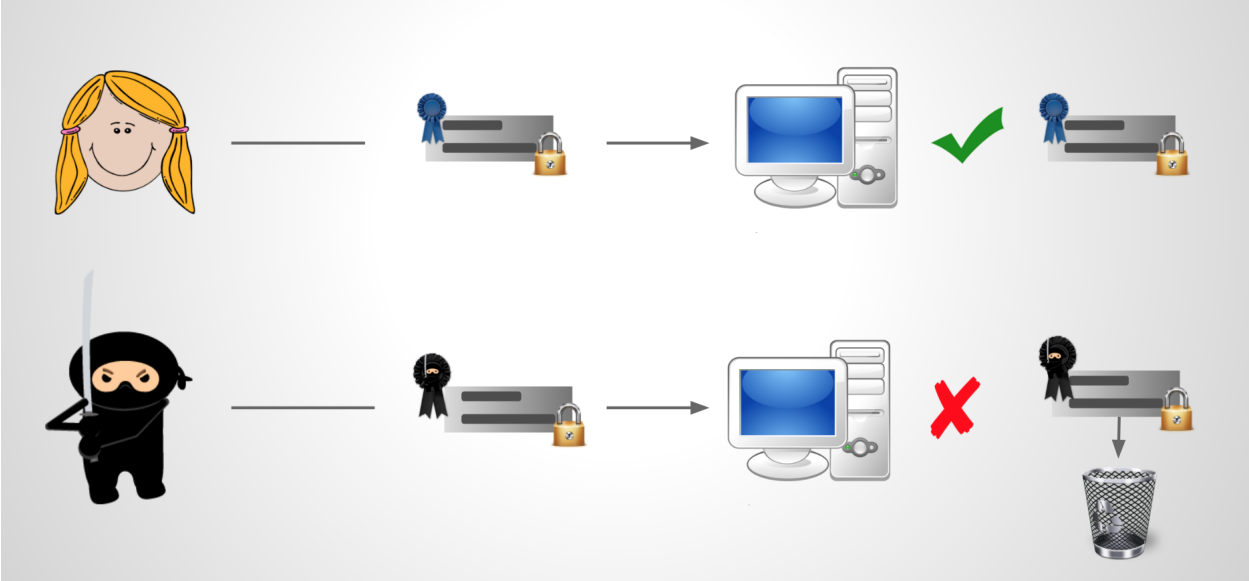

Figure 2. File Validation using a digital signature scheme.

Implementation of Digital Signature Scheme [6]

If we were to implement ***Blackbox***, we would use one of the digital signature algorithms provided by the OpenSSL open source library. For instance, OpenSSL provides an implementation for DSA (digital signature algorithm) which is approved by the US Federal Information Processing Standard FIPS 186 (Digital Signature Standard, DSS), ANSI X9.30.

References

[1] Dimakis, A. G., Godfrey, P. B., Wainwright, M. J., & Ramchandran, K. (2007). The benefits of network coding for peer-to-peer storage systems. In Third Workshop on Network Coding, Theory, and Applications.

[2] Nychis, D. G., Andreou, A., Chheda, D., & Giamas, A. (2008). Analysis of erasure coding in a peer to peer backup system. Information Networking Institute, Carnegie Mellon University.

[3] R. Rodrigues and B. Liskov, High availibility in DHTs: Erasure coding vs. replication. Proceedings of the 1st International Workshop on Peer- to-Peer systems (IPTPS), March 2002.

[4] H. Weatherspoon and J.D. Kubiatowicz, Erasure Coding vs. Replication: A Quantitative Comparison. Peer-to-Peer Systems: First International Workshop, IPTPS 2002, LNCS 2429, pp. 328–337, 2002

[5] http://en.wikipedia.org/wiki/Erasure_code

[6] https://www.openssl.org/docs/crypto/dsa.html

File Encryption

Encryption Method (Confidentiality)

Recall our goal of confidentiality: when files are stored in another computer, other users must not be able to obtain any information about the file except some primitive data such as the file size. In order to achieve this goal, every file is encrypted by the owner of the file before it is distributed to peers in the network. Each user will store their AES key on their own computer and/or on a USB drive in the case that their computer's hard drive is compromised. It is essential that the user does not lose their AES encryption/decryption key (symmetric cryptosystem where encryption and decryption keys are the same). If they do, they will no longer have the ability to access their files. Instead of storing Alice's AES key in plaintext on her computer, we further encrypt her key using a password that Alice specifies. This insures that if someone gains access to Alice's computer, they would need to know her password for her AES key in order to recover the key. This additional encryption step adds another layer of security to ***Blackbox.***

Background on AES

AES stands for Advanced Encryption Standard and is based on the Rijndael cipher. It was adopted by the U.S. government in 2001. AES is a symmetric key encryption.[1] This is good for our use purpose because we want only the person whose file it is to be able to encrypt/decrypt it. Our application has no need for asymmetric encryption.

Security of AES

As of 2009, the only successful attacks on AES so far have been side-channel attacks[1]. AES relies on a block cipher mode of operation. We chose to use AES in Cipher Block Chaining mode.

Performance of AES

AES performs well on devices ranging from 8-bit smart cards to high performance processors[1]. The following is a figure showing the performance metrics of AES encryption in CBC mode with varying key sizes.

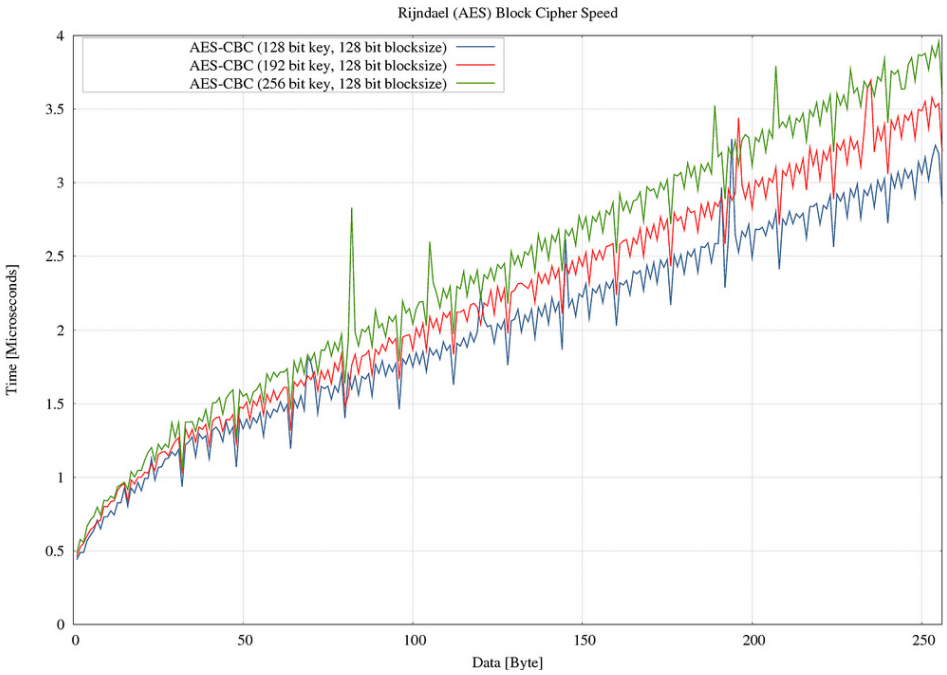

Figure 3. AES in CBC mode performance analysis for varying key sizes

Implementation of AES

Implementation of AES can be done using the OpenSSL library. OpenSSl provides an implementation of AES. The following is an excerpt from the OpenSSL documentation [2]:

```
aes-[128|192|256]-cbc  128/192/256 bit AES in CBC mode
aes-[128|192|256]      Alias for aes-[128|192|256]-cbc
aes-[128|192|256]-cfb  128/192/256 bit AES in 128 bit CFB mode
aes-[128|192|256]-cfb1 128/192/256 bit AES in 1 bit CFB mode
aes-[128|192|256]-cfb8 128/192/256 bit AES in 8 bit CFB mode
aes-[128|192|256]-ecb  128/192/256 bit AES in ECB mode
aes-[128|192|256]-ofb  128/192/256 bit AES in OFB mode
```

Available modes of operation in the OpenSSL AES implementation include CBC, ECB, and CFB modes. The ECB mode of operation does not provide the security we want, namely, identical plaintext blocks are encrypted as identical ciphertext blocks. CBC is encryption parallelizable but not decryption parallelizable whereas CFB is not encryption parallelizable but decryption parallelizable. [1][2]

References

[1] AES http://en.wikipedia.org/wiki/Advanced_Encryption_Standard]
[2] OpenSSL AES https://www.openssl.org/docs/apps/enc.html

File Identification

Each file chunk stores in its metadata the hash of its parent file (fileID). This way, when Alice broadcasts a request to recover a file with a specific fileID, all nodes that receive the request will look at the chunks they are storing and check if they have chunks for that particular fileID.
Each client stores a hash table mapping the fileID to the chunks location on their file system. This way, when Alice requests a fileID, the peers can quickly refer to this hash table to check if they are storing any of Alice's chunks. If they are, they will retrieve the chunk and send it out on the network.

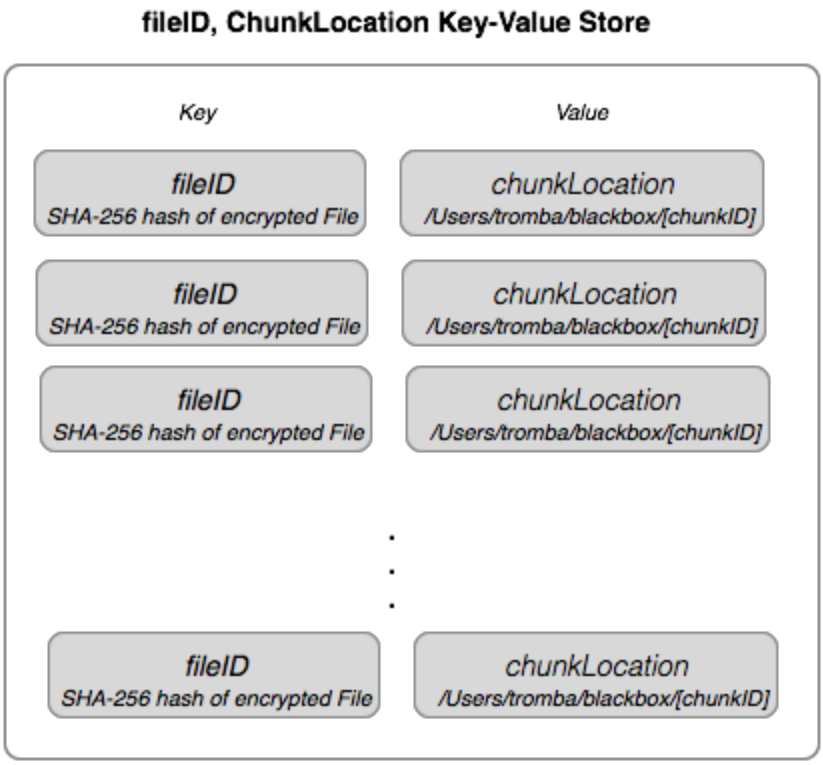

Figure 4. Dictionary storing mapping between fileIDs and chunkLocations

Security

If we choose an appropriate hash function $h : \{0,1\}^* \rightarrow \{0,1\}^d$, it will have the following properties [2]:

1. **One-way (pre-image resistance)**
   It is hard for an adversary, given y=h(x) (where x is picked uniformly at random from $\{0,1\}^n$ to find any x' such that h(x') = y.
2. **Weak Collision-Resistance**

Infeasible given $x \in \{0,1\}^*$ to find $x' \neq x$ such that $h(x) = h(x')$.

3. **Pseudo-Randomness**
   Hash function h is indistinguishable under black-box access from a random oracle. For this property to be satisfied, we need a family of hash functions.
4. **Non-malleability**
   It is infeasible given h(x) to produce h(x') where x and x' are related. (e.g. x' = x+1).

In choosing a hash function, we must be careful to verify that it is secure (according to the security requirements provided above). The following summarizes the security of widely-used hash functions:

Table color key
No known successful attacks
Theoretical break
Attack demonstrated in practice

Hash function	Security claim	Best attack	Attack date	Comment
MD5	2^{64}	2^{18} time	2013-03-25[1]	This attack takes seconds on a regular PC. Two-block collisions in 2^{18}, single-block collisions in 2^{41}
SHA-1	2^{80}	2^{61}	2005-08-17[2]	Attack is feasible with large amounts of computation power.[3]
SHA256	2^{128}	24 of 64 rounds ($2^{28.5}$)	2008-11-25[4]	
SHA512	2^{256}	24 of 80 rounds ($2^{32.5}$)	2008-11-25[4]	

Figure 4. Hash Function security summary[1]. For more information on the available hash functions refer to the Appendix.

We chose to use SHA-256 as there is no theoretical break yet compromising the security of this hash function.

References

[1] Hash Function Summary, Wikipedia
http://en.wikipedia.org/wiki/Hash_function_security_summary
[2] 6.857 Course Notes
http://courses.csail.mit.edu/6.857/2014/files/L04-hash-functions-introduction.pdf

Network Structure

We use a decentralized network structure where each node maintains information about the state of the network. We use a decentralized network because individual nodes may go offline at any time. Each node keeps track of the identities and properties of every node in the network.

Nodes

Nodes in our network are computers running our program that are willing to share hard drive space. Nodes form two-node “partnerships” with other nodes where each node in the partnership agrees to let the other node use some amount of space on its hard drive.

Each node in the network has an RSA public/private keypair. The public key is used to identify nodes, and the keypair is used when sending messages (see next section).

Each node must be added to the network by an existing node in the network. The node that added the new node vouches for the trustworthiness of the new node. If a large number of nodes added by the same node are found to be malevolent or uncooperative, the network will distrust the node that added these nodes. This prevents an adversary from creating new nodes whenever his existing nodes are found to be untrustworthy.

If a prospective user does not know any nodes that are willing to add him to their network, the user can start a new network and invite other nodes to join it. By limiting networks to users who know each other in real life, we prevent networks from becoming too large and running into scaling issues.

An existing node adds a new node to the network by signing his public key. The signed public key is distributed to all the nodes in the network. Under this system, the public key of each node in the network is signed by the node that invited it, so any node in the network can check that the public keys of every other node are correct, as long as they have the public key of the first node in the network. In addition, every node can determine which node invited any given node, so nodes that invite too many malicious nodes can be punished.

Each node in the network keeps track of the state of the network. This state includes the public keys and signatures described above, in addition to various statistics about each node in the network. These statistics include uptime, amount of space available, and how often each node is online at the same time as this node.

Message Passing

Sometimes, two nodes in the network need to send messages to each other when they are not online at the same time. This might be necessary to recover files or to verify that a node has the

files that it promised to store. Our network accomplishes this by sending the message to intermediary nodes to pass on to the destination.

Suppose Alice needs to send a message to Bob, but Bob is not currently online and Alice is about to go offline. Alice picks an ordering of the nodes in the network, and sends the message to the first *MESSAGE_DUPLICATES* nodes in the ordering. Whenever less than *MESSAGE_DUPLICATES* nodes have the message, a node with the message copies it to the next node in Alice's ordering. When Bob eventually comes online, Bob polls all the nodes in the network for messages addressed to him. When Bob receives the message, the copies of the message are deleted.

To prevent other nodes from reading the message, Alice encrypts the message with Bob's public key. Since only Bob has access to his private key, only Bob can read the message. In addition, Alice signs the encrypted message with her public key, so adversaries cannot forge messages without Alice's private key. The message destination and the ordering of backup nodes are signed along with the message. Messages are encrypted using RSA-OEAP. Messages are signed using RSA-PSS.

The network tries to ensure that at least *MESSAGE_DUPLICATES* online nodes have the message, to decrease the probability that all nodes with the message go offline at the same time. In addition, the order in which nodes are chosen to get copies of the message is determined by Alice, so an adversary can't try to copy the message to his friends until all *MESSAGE_DUPLICATES* copies are in his control and delete the copies to prevent the message from being delivered.

Since this method of sending messages uses much more bandwidth compared to sending the message directly (by a factor of

$$\frac{MESSAGE\_DUPLICATES \times (\text{amount of time Alice and Bob are both offline})}{\text{amount of time a node is online after getting a copy of a message}}$$

), our system tries to avoid using this method. In particular, nodes prefer to share space with nodes that tend to be online at the same time as itself. In addition, messages have a maximum lifetime of one day and nodes will refuse to deliver messages from nodes that send too many messages.

Partnerships

Files are stored in the network via "partnerships". A partnership consists of two nodes that agree to store each other's files. Each node may be part of multiple partnerships. We picked

this design because it simplifies the problem of ensuring fairness, because each node will only share space with other nodes that promise to share a similar amount of space with the node.

When a partnership is formed, the partners decide how much space to share with each other and how often they expect to be online. These numbers can be renegotiated later, in case a node gains or loses hard drive space or change usage patterns.

The amount of space partners share with each other is not necessarily symmetrical, if they have different expected uptimes. We define the expected benefit a node A receives from a partnership (A, B) to be the amount of space B is willing to share multiplied by B's expected uptime. In other words, it is the expected value of the amount of files that A can recover from B immediately, in the event of A losing his files. When a partnership is formed, the partners decide how much space each partner shares so that the expected benefit each node receives from the partnership is equal, i.e.

$$\begin{aligned} &(\text{space shared by } A) \times (\text{expected uptime of } A) \\ = &(\text{space shared by } B) \times (\text{expected uptime of } B) \end{aligned}$$

When a node wants to save a file to the network, it decides how important the file is and picks a "replication factor" accordingly. Then, it picks partners with whom to share the file until the sum of the expected uptimes exceeds the desired replication factor. This ensures that the average number of nodes with the file that are online at any time is greater than the desired replication factor. If the node does not have enough partners such that this is possible, it forms new partnerships.

Each node in the network that is willing to form new partnerships broadcasts how much unshared space it has and its expected uptime. Each pair of nodes in this set keeps track of how often they are online at the same time as each other. When a node needs to form new partnerships, it goes through the nodes in descending order of how often they are online at the same time, and negotiates partnerships with those nodes until its needs are met.

Ending Partnerships

Nodes in partnerships keep track of the uptime of their partners. Every day, each node sends a message to its partners. Since messages have a lifetime of 1 day, this allows the node to know if its partner was online at any point in a 24-hour period. When two nodes enter a partnership, they decide what proportion of days each partner should be online. If the proportion of days a node's partner is online falls below this ratio (plus a grace period of a few days or weeks to allow a user with a broken computer to get a new one), the relationship is terminated and the node will refuse to enter more relationships with that node.

Since new relationships are only formed between nodes that are currently online, a node that has left the network forever will not form any new relationships. Its existing relationships will eventually be terminated, so it will eventually not be in any relationships.

Nodes in relationships can negotiate new minimum uptimes or end the relationship without prejudice. This could be useful if a node is going on vacation but will be back.

Anti-Leeching

Leeching occurs when a malicious user attempts to manipulate the layout of the network in order to unfairly glean backup space on others' computers without providing any in return. In a large network, we must take some steps against leeching in order to ensure that hard drive space is used optimally and distributed fairly.

Although there is little that we can do in the short term against a user that is capriciously or spontaneously malicious – deleting copies of peers' files when those peers wish to restore backups, for example – we can still try to prevent against long-term leeching by instating a system of etiquette, and punishing users who deviate from that system of etiquette.

Partnerships

For an individual user, the process of trading storage space on their hard drive for storage space on others' hard drives is reduced to the problem of forming 1-on-1 partnerships with other users of the network. Each user might have many 1-on-1 partnerships with other users, which when combined create enough space to store all of the user's files. If one partner proves to be uncooperative, a user can abandon that partnership and pair up with someone else instead. Partnerships thus provide a useful abstraction for thinking about the logistics of leeching.

Quizbeats

Potential leeching attack: An untrustworthy user claims to be backing up others' files, earning backup space in return, but is actually storing nothing on their computer.
Counter: **"Quizbeats."** In order for you, a user, to check that one of your partners is storing the files that you think they are storing, you sends them periodic quizzes (which we call 'quizbeats'). You send a random range of bits that you want your partner to prove they are storing. Your partner responds back with those bits. Note that your partner is storing an encrypted version of your file, so in order to check that the bits they reply back with are accurate, you must encrypt the file.

Collusion

Potential leeching attack: Two of a user's partners collude to save hard drive space by storing only a single copy of the user's file, sharing data with each other in order to respond to quizbeats and make it seem like they are storing two copies.
Counter: If you are storing multiple backups of the same file across multiple partners, do not give any two partners the same encryption of a file. Instead, use randomized encryption to ensure that those partners must store different data. For example, pad the beginning of each file with a prefix of random bits, which are discarded in decryption. Thus, a user will not be cheated out of the reliability that multiple copies provides. Note that this means that to verify a quizbeat, a user's computer must recompute the randomized encryption each time, and remember the prefix of random bits for each file and partnership; however, should the user's hard drive crash, files can still be decrypted without needing the random bits.

Recovery

When hard drives crash, users must be able to retrieve the files that were stored with their partnerships elsewhere on the network. Requesting a file works the same way that a quizbeat does, except that the range of requested bits is the entire file. Since a user's hard drive has just crashed, that user lost their copies of their partners' files -- ideally, their partners will not mistake the crash for uncooperativeness, and will not delete your files in retaliation. To prevent this, let the rules of etiquette decree there will be a window of forgiveness -- fifteen days -- after which partners delete your files.

Faking Recovery

Potential leeching attack: If partners are too forgiving, an uncooperative user could leech space off of the system by always pretending to have a crashed hard drive, and taking advantage of a window of leniency.
Counter: Any user who wishes to recover their files must first reinstate copies of their partner's files onto their new blank hard drive, and pass a few quizbeats to prove it, before the partner allows them to recover their files.

Punishing Unfair Users

If a user's partner fails any of the anti-leeching etiquette -- flunking a 'quizbeat' after fifteen days, or deleting another user's files before fifteen days are up – then they are broadcast as 'untrustworthy' to the trust graph. Anyone dealing with the untrustworthy user is responsible for forming new partnerships in order to ensure their files are backed up.

Conclusion

Blackbox presents a free alternative to commercial back-up services by creating a distributed peer-to-peer network. Our system guarantees confidentiality and integrity of stored files by encrypting and signing all files. Users are able to reliably access their files through our distributed network. Finally, our system ensures fairness of resource use through our system of partnerships and anti-leeching measures.

Appendix

Regenerating Codes

A study by Dimakis et. al. [1] studies a different kind of redundancy scheme in peer to peer networks called regenerating codes. These regenerating codes use slightly larger chunks than erasure codes but can reduce maintenance bandwidth by 25% or more by requiring a new node to download less data than the full k chunks needed to reconstruct the file.

[1] Dimakis, A. G., Godfrey, P. B., Wainwright, M. J., & Ramchandran, K. (2007). The benefits of network coding for peer-to-peer storage systems. In Third Workshop on Network Coding, Theory, and Applications.

Storage Location

Files that a user wants to be backed up will be stored in a special folder much like Dropbox's folder. All files in this folder will be encrypted and distributed to peers to save.

Files that the user is saving for others will also be stored in a database on the computer depending on the user's operating system.

The client application running on the users machine will allow a user to input how much space they want to allow ***Blackbox*** to store on their computer. They in turn will be able to store files in proportion to the amount of space they allocated to storing other people's files.

Hash Algorithms

Algorithm and variant		Output size (bits)	Internal state size (bits)	Block size (bits)	Max message size (bits)	Word size (bits)	Rounds	Bitwise operations	Collisions found	Example Performance (MiB/s)[14]
MD5 (as reference)		128	128	512	$2^{64} - 1$	32	64	and, or, xor, rot	Yes	335
SHA-0		160	160	512	$2^{64} - 1$	32	80	and, or, xor, rot	Yes	-
SHA-1		160	160	512	$2^{64} - 1$	32	80	and, or, xor, rot	Theoretical attack (2^{61})[15]	192
SHA-2	*SHA-224* *SHA-256*	224 256	256	512	$2^{64} - 1$	32	64	and, or, xor, shr, rot	None	139
	SHA-384 *SHA-512* *SHA-512/224* *SHA-512/256*	384 512 224 256	512	1024	$2^{128} - 1$	64	80	and, or, xor, shr, rot	None	154
SHA-3	*SHA3-224* *SHA3-256* *SHA3-384* *SHA3-512*	224 256 384 512	1600 (5×5 array of 64-bit words)	1152 1088 832 576		64	24	and, xor, not, rot	None	

Source: http://en.wikipedia.org/wiki/SHA-2

Message Replication

MESSAGE_DUPLICATES = 3