PeerBox
Secure Decentralized Dropbox

Nada Nasr 19-3881
Yousra Hazem 19-4248
Nisma El-Nayeb 19-1169
Yahia El Gamal 19-1023
Ahmed El Safty 19-2180
Omar Nada 19-4332

Department of Computer Science and Engineering, German University in Cairo, Cairo, Egypt

Abstract. PeerBox is a decentralized, P2P version of Dropbox, a cloud file sharing and storage application. The aim was to create a more secure and scalable application for file storing and sharing, where files are split and stored on the users' devices.

1 Introduction

1.1 Motivation

Our main motivation was the vulnerabilities that come with cloud file sharing and storing solutions. Having your personal files and data stored on a server can potentially allow non-authorized access, particularly if the server has been compromised or if the service provider is untrusted.

1.2 Summary

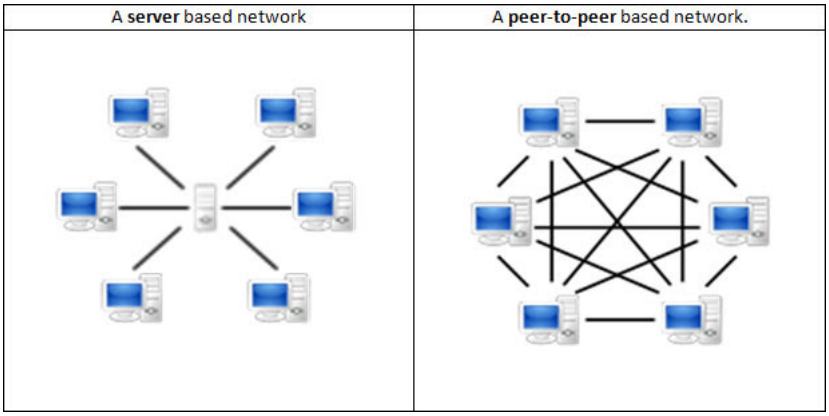
In a centralized network, the server is the focus of control, and in the case of Dropbox, that means having all users' uploaded data stored on a server and having the possibility of non-authorized access through it. Decentralization, however, means that there is no server, just the users' devices. With the idea of decentralization in mind, we developed PeerBox. PeerBox is a decentralized, P2P version of Dropbox, that does not store the users' data or files on a server, and instead stores it on other users' devices. Users send and receive files and information in a secure P2P manner. PeerBox is written in Java. How it works will be further discussed throughout this report.

2 P2P Architecture

We will begin by introducing three types of P2P architectures.

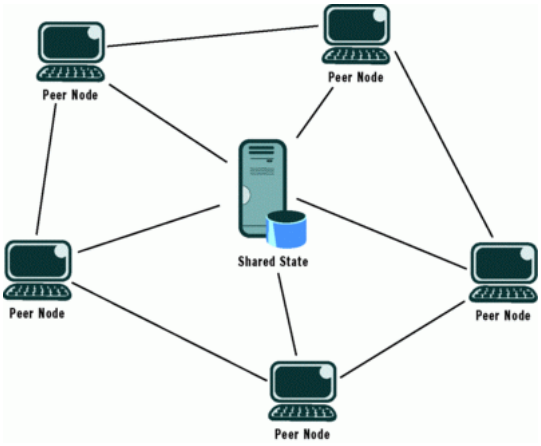
2.1 Pure P2P

The quest to create and successfully manage a pure peer-to-peer system has been elusive. They are hard to maintain, and with no single point of control, tracking the activity of each peer is almost impossible. A pure peer-to-peer system has to be free of hierarchy and centralism. In other words, all peers are identical, all nodes are peers in a total democratized peer group nodes. Such that each of them has equal amount of control over the network, acting both as a client and a server. In a file sharing system using this architecture, the peer searching for a file has to broadcast a request to as many peers as possible.



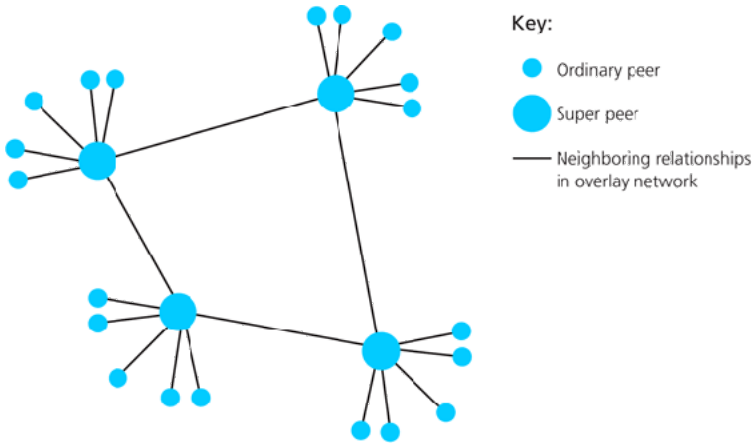
2.2 Hybrid P2P

Hybrid peer-to-peer networks introduce central servers holding information about peers like their IP addresses and files they are sharing. This helps peers lookup files and the clients sharing them. A central server also facilitates monitoring the network and maintaining it. As much as it "kills" the principle of decentralized p2p, a centralized server helps in building the system security.



2.3 Mixed P2P

The third and final type of p2p networks is the Mixed peer-to-peer, it has super nodes acting as small servers spreading all over the network. It's more centralized than the Hybrid p2p systems. More hierarchical as well, As only the Super Peer communicates with each other in a pyramid-role-like manner.



3 Design Choices

3.1 TomP2P vs. openChord

TomP2P was our initial choice to use for PeerBox. We later decided to switch to openChord. This comparison might shed some light on why we thought TomP2P was a good choice and why we went for openChord in the end.

TomP2P TomP2P is a distributed hash table which provides a decentralized key-value infrastructure for distributed applications. TomP2P stores key-value pairs in a distributed hash table. To find the peers to store the data in the distributed hash table, TomP2P uses an iterative routing approach. The underlying protocol for all the communication with other peers uses state-less request-reply messaging. Moreover, It supports many features like extended DHT operations (multiple values, same key), non-blocking IO, managing Future connections, Iterative Kademlia-like routing system, and more. However, the application is poorly documented with weak tutorials and official sample codes. Not to mention that it's too powerful for the project. We therefore decided to go with openChord.

openChord Open Chord provides the possibility to use the Chord distributed hash table within Java applications by providing an API to store all serializable Java objects within the distributed hash table. It has meaningful and descriptive documentation, profound code examples, and simple features for our security-oriented project.

3.2 Security Techniques

For our project, we used Java libraries for hashing and encryption[2]. We found them to be the most efficient and reliable implementation of the algorithms we wanted to use.

Public Key Encryption We use **RSA** encryption for P2P sharing. It is widely used for secure data transmission and believed to be one of the most successful, asymmetric encryption systems today. It is also suitable for sending small chunks of data.

Private Key Encryption AES is one of the most commonly used and most secure symmetric key encryption algorithms. Up till now, no known attacks on AES exist.

Hashing Function Since we will need to compute a large amount of hashes for big files, a faster hash function is better. We therefore chose **MD5**, which is known to be faster than its well-known counterpart SHA.

4 Implementation Details

4.1 Uploading Files

The steps for uploading a file are as follows:

1. Split the chosen file into *pieces* of equal size.
2. Hash each piece.

3. For the n pieces encrypt each piece with keys $K_{P1} \dots K_{Pn}$.
4. Store the information from steps 1 and 2 in $DHT1$.
5. Generate the *torrent file*.
6. Generate a key Key_{torr} which we will use to encrypt the torrent file with and store in $DHT2$.

4.2 The DHTs and The Torrent File

DHT1 $DHT1$'s purpose is storing the pieces of uploaded files. For one file the table might look something like this:

Key	Value
Key_{DHT1_1}	$E_{K_{P1}}(Piece_1)$
Key_{DHT1_2}	$E_{K_{P2}}(Piece_2)$
$\vdots$	$\vdots$
Key_{DHT1_n}	$E_{K_{Pn}}(Piece_n)$

where $Key_{DHT1_i} = hash(Piece_i)$ and $Piece_i$ is a piece encrypted with key K_{Pi}.

Torrent File The torrent file stores information about an uploaded file. For each piece we store the information in the following format $\langle hash(Piece_i), Key_{DHT1_i}, IV_i \rangle$, where $hash(Piece_i)$ is the ith hashed piece, Key_{DHT1_i} is the DHT1 key where this piece is stored, and IV_i is the initialization vector used in encrypting $Piece_i$.

DHT2 Here we store the torrent files for all uploaded files. For one file this might look something like this:

Key	Value
Key_{DHT2_1}	$E_{Key_{torr1}}(torrent_1)$

where $Key_{DHT2_i} = hash(torrent\_data_i \| current\_time)$[1] and $torrent_i$ is the torrent file encrypted with Key_{torri}.

In case of editing a file by one of the users with has access to it, the table might look like this:

Key	Value
Key_{DHT2_1}	$\langle Key_{temp} \| Key_{DHT2_2} \rangle$
Key_{DHT2_2}	$E_{Key_{new}}(torrent_1)$

How this works will be further explained in section 4.4.

[1] *current_time* is a time stamp for when the file was encrypted.

DHT3 This table is used for storing users' public keys and can be represented as follows:

Key	Value
MAC_i	$\langle Modulus_i, Exponent_i \rangle$

where MAC_i is the unique MAC address of a device, and $Modulus_i$ together with $Exponent_i$ re-generate the original public key.

4.3 Sharing Files

For someone to share a file they have to go through two steps; first they have to upload the file to the peer network as mentioned earlier and then they have to share it to give access to the content of the file shared. The files on the network are encrypted with the owner's private key, so if owner A wants to share file F with peer B then A sends the decryption information of F to B. Of course, in order for B to know which file has been shared with them, A sends the file's index in DHT2 namely Key_{DHT2_i} along with the file name. So all in all we send $E_{Key_{Public}}(File\_name, Key_{DHT2_i}, Key_{torr}, IV_i)$. On the other side B gets the option of accepting the file by pressing y or n and if they press y they sync the file from the network and decrypt it using the information they just received.

4.4 Updating Files and Syncing - The Hobba Technique

The steps for updating a file after having edited it are as follows:

1. Do steps 1-5 from section 4.1 with the edited file.
2. Generate Key_{temp}.
3. Generate Key_{new} by XORing Key_{temp} and Key_{torr}, where Key_{torr} is the key with which the old torrent file was encrypted.
4. Encrypt the newly created torrent file with Key_{new}.
5. Insert into DHT2 as such:

Key	Value
Key_{DHT2_1}	$\langle Key_{temp} \| Key_{DHT2_2} \rangle$
Key_{DHT2_2}	$E_{Key_{new}}(torrent_1)$

where Key_{DHT2_2} serves as a pointer.

For each time an authorized user edits a file, this process is repeated. So if another user further edits the file, then the DHT2 entries for this file will look like this:

Key	Value
Key_{DHT2_1}	$\langle Key_{temp1} \| Key_{DHT2_2} \rangle$
Key_{DHT2_2}	$\langle Key_{temp2} \| Key_{DHT2_3} \rangle$
Key_{DHT2_3}	$E_{Key_{new}}(torrent_1)$

where Key_{new} can now be obtained by first XORing Key_{temp1} and Key_{torr}, and then XORing the result of that with Key_{temp2}.

If a user wishes to sync his file with the other users who have access to that file, the following steps will accomplish that:

1. The user already has the original Key_{torr} and the Key_{DHT2_i} for $file_i$, which is used to access DHT2.
2. If at Key_{DHT2_i}in DHT2 the value is the torrent file, then download(as in section 4.5) the file.
3. Else, if the value is $\langle Key_{tempi} \| Key_{DHT2_i} \rangle$, keep "*hobbing*" by going to step 1 using the pointer(s) .

4.5 Download

A file, given its torrent info, can be downloaded as follows:

1. Decrypt the torrent file.
2. Obtain, from the torrent file, the needed information about all pieces of the file.
3. Obtain all the needed pieces from DHT1.
4. Decrypt each piece.
5. Re-assemble the pieces to obtain the file.

5 Attack Scenarios

5.1 Gaining Access to Unshared File

One possible attack that the system handles properly is when a malicious user tries to gain access to an unshared file. There are a number of obstacles that will face the attacker such that it will ultimately secure the access to files.

1. In DHT2 (where the torrent information is stored) it is exponentially hard to know exactly the key of the desired torrent file. Even if the attacker was lucky enough to know the key of the hash, the value of the cell is encrypted by a key Key_{torr} which is only exchanged with devices that received a sharing message (legitimate devices) through another layer of encryption (in the next point).
2. Assuming the attacker was in the middle of sharing. It will be impossible for the attacker to get the Key_{torr} unless the attacker has the private key of the receiver (which should never happen), the sharing messages are encrypted by the public key of the receiver of sharing so the attacker (man in the middle) will never gain access to the info because it doesn't know the private key of the receiver.

5.2 Decrypting a stored piece

This will be quite hard for the attacker to decrypt the user pieces stored on its local storage as it is encrypted by the Key_{torr} secured by DHT2 and an initialization vector (also secured by DHT2 as well).

5.3 Flooding the network

One attack that the network isn't ready for yet is flooding the network with huge files. Unfortunately, at the current state is the network is vulnerable to this kind of attacks. One approach we thought of to avoid this vulnerability is having a kind of reputation system. Where you gain more reputation by storing more data on your local disk and be available more to the network. The more reputation a node has, the more storage it can use of the network. Reparation propagation can be done through a gossip like method.

6 Future Work

In PeerBox, certain points need to be reviewed and improved, some of which are as follows:

1. Whenever a node gets a request to delete a new piece, it needs to verify that the requester has the permission to do so. This can be done through a centralized access control database, that stores which node has access to edit which files.
2. In DHT1, we do not delete old versions of a file when we update, in order to be able to go back to the history of a file. However, this will need a pointer to previous versions in the most recent file version, which is currently not implemented.
3. As mentioned in the previous section, PeerBox has a vulnerability when it comes to flooding the network with huge files. This needs to be further looked into and fixed. We mentioned an approach for avoiding this problem in section 6.3 as well.
4. PeerBox currently has no GUI. A proper GUI is, of course, essential and needs to be implemented.

References

1. (http://sourceforge.net/projects/open-chord/)
2. http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html
3. Urdaneta, G., Pierre, G., and Van Steen, M. 2009. A Survey of DHT Security Techniques. VU University, Amsterdam, The Netherlands.