Design Doc for CS161 Project2

In order to initialize a user, we decided to utilize usernames and passwords useful. Because HMAC can generate a key of 32bits, we will use a slice of the first 16 bits for the key for encryption.— has to be more detailed here.

Our User Struct will contain the following:
1. username as sting
2. KeyA as byte array for Symmetric MAC key
3. KeyE as byte array for Symmetric Encryption key
4. KeyN as byte array for Symmetric key for name confidential
5. PrivateRSAkey as RSA private key for sharing files

InitUser
We use each user's username and password to create a key for his own key so that it can be acquired when being decrypted. (symmetric key)
In order to avoid the attack, we will append one ASCII character, "/" in between username and password. For instance, when username, "Alice," has its password, "thisisAlice," if we store this to the server by encrypting "username+password," the server let an attacker be able to acquire information of "Alice," when the attacker inputs "Alic" and "ethisisAlice" as his username and password.
We use json.Marshal(userdata) in order to marshal the user data and cast it into a byte array.
We use CFB encryption to encrypt the user information to store to Data Server. We have to encrypt the data to be safe as Data Server is Untrusted server. And we are passing the encrypted data byte array into HMAC to create a tag. We will use this tag when calling GetUser.

GetUser
Now that we uploaded the user data using InitUser, we will be able to verify if there is a matching user when we have the user input; username and password.
We will follow the identical steps from InitUser in terms of making keys as it requires steps for verification.
We will check if there is stored information based on username and password input. If there is no user data stored in the server, we raise an error stating that there is no user data stored.
If there is a matching data, now we will have to check if the HMAC tags are matching before doing any decryption. If tags are not matching, it means that the malicious server has tampered with the user data. So we raise an error stating that data has been corrupted.
If everything is fine, we return the user data.

FileServer structure
we will creat user files for each user and make it point to "masterfile."

StoreFile
StoreFile works like InitUser in a sense that it's uploading encrypted data to Data Server. Because we have the "masterfile". Here we make all data files stored in the masterfile.

And also store data files to Data Server. Instead of using username, we will use filename as its key for the structure, Dictionary, so that we are able to find the following stored data from the server.

AppendFile
Now that we have the masterfile that points all user data stored. We can find each masterfile using the filename input. If there exists a previously stored file in the filename, we store the input data on DataServer and store its location to its master file.
If we can not find a file that is previously stored with the filename, we raise an error.

LoadFile
It works like GetUser. It will take the name of the file and try to retrieve the encrypted and stored data by using Authentication, Encryption and NameConfidential keys from File Server.

ShareFile
In order to share file, we will create a node, that contains two keys, one for the data encryption and the other one for MAC, and location of file. In order to verify for the recipient who the file is shared from, we will have to include the sender's signature. We include the signature with the file data, two new randomly generated keys that we will use and the location of the file and encrypt these with generated keys. And store it to a random location. And we will make a new byte array with those two keys and location by concatenating them and encrypt this with the recipient's public key.
And store it onto another random location to send this to the recipient.

ReceiveFile
As the recipient has his own private key and the sender's encrypting the sending file with the recipient's public key, only the recipient will be able to decrypt the file. When he decrypts the encrypted and stored file. He will see the concatenated byte array that has two keys and the stored location where the sender's signature and the node are stored. Because it includes the sender's signature, the receiver can assure who sent the file to him. It will count the authentication. And if anyone has tampered with the stored file, as we're encrypting all of these with HMAC. The receiver wouldn't be able to see correct information. Thus, this idea works.

RevokeFile
Since we only knows where the masterfiles are located, we can simply change the location of the masterfile. And delete the previous masterfile by using the given function, DatastoreDelete(key string). Since the file is gone we don't have to do anything for previous shared users because all they will get from accessing to the location of the previous masterfile would be gibberish.