What data structures are you going to use to organize data on Datastore?

Struct User {rootKey, username, SigningKey, DecryptingKey}
Struct FileReference {RootKey (to derive symmetricKeys), fileName, MiddleManPtr , FilePtr, sharingMapPtr (computed on the fly)}
Struct MiddleMan {RootKey(derive symmetric keys for File), FilePtr}
Struct MiddleManEntry {RootKey (derive symmetric keys for accessing MiddleMan, MiddleManPtr}
Struct File {HeadRootKey, TailRootKey, HeadPtr, TailPtr}
Struct FileNode {RootKey, Next, ContentPtr}
Struct Invitation {EncryptedRootKey, Signature of Encrypted Root Key, MiddleManPtr}

How will you authenticate users?

When a user is created, use their username to create a UUID . This UUID and password will be used to create a rootkey. With the rootkey can derive two new keys for an encrypt-then-MAC scheme.

How will you ensure that multiple User objects for the same user always see the latest changes reflected?

User objects will never store state that can change in a local variable but rather point to volatile data on the datastore. For example multiple user objects can just point to the same file in the datastore and always retrieve the most up to date version.

File Storage and Retrieval: How does a user store and retrieve files?

EvanBot is logged in and stores a file. How does the file get stored in Datastore? What key(s) and UUIDs do you use? How do you access the file at a later time?

Storing: User creates a unique UUID for a file reference using the username and filename(username || filename). If the user is the owner they can directly access the file using the reference (The FilePtr will not be uuid.Nil and the RootKey will be associated with the File). Else if not the owner, file reference must be accessed through a MiddleMan obj. The RootKey on the reference will be associated with the MiddleMan.

Retrieving: Deterministically compute the UUID to fetch the data. Deterministically recompute the two symmetric keys by using the rootkey and two hardcoded purposes to get the file reference. Repeat above process but this time using the FileReference's RootKey to derive the symmetric keys and the FilePtr/MiddleManPtr. Check authenticity with MAC then decrypt with the other key.

Efficient Append: What is the total bandwidth used in a call to append?

Our bandwidth will be directly proportional to the size of the content being appended. Will use a linked list with head and tail pointers to point to the beginning and end of the list. To append, simply retrieve the last node through the tail pointer and append a new node with the content. Note Nodes only contain two pointers Next and ContentPtr. Thus we never download variable amounts of data.

File Sharing: What gets created on CreateInvitation, and what changes on AcceptInvitation?

EvanBot (the file owner) wants to share the file with CodaBot. What is stored in Datastore when creating the invitation, and what is the UUID returned? What values on Datastore are changed when CodaBot accepts the invitation? How does CodaBot access the file in the future?

Create an Invitation struct. Create a brand new middleman object, append the new middleman to the corresponding File References SharingMap. Create a new RootKey that is used to deterministically derive two symmetric keys for ensuring integrity and confidentiality on the MiddleMan object (Encrypt-then-MAC). Encrypt the middleman object with the keys and compute a MAC on it. Encrypt the root key. Finally provide a signature on the key for authenticity and integrity.

CodaBot (not the file owner) wants to share the file with PintoBot. What is the sharing process like when a non-owner shares? (Same questions as above; your answers might be the same or different depending on your design.)

Same idea. Instead no keys need to be generated and no middleman needs to be created. (All users who chain shares will have the same middleman, assuming no share came from an owner). Use the hybrid encryption process as mentioned above with the already existing data.

File Revocation: What values need to be updated when revoking access?

Using the diagram above as reference, suppose A revokes B's access. What values in Datastore are updated? How do you ensure C and G still have access to the file? How do you ensure that B, D, E, and F lose access to the file?

We need to create a new Root key for accessing the file object (Will copy over old contents to a new file). First get all the content. Next, re-encrypt the content using the encrypt-then-MAC scheme with the new root key. Update the root keys stored in all middlemen who are not being revoked access and their filePtr to point to the new object (This is done through the SharingMap). B,D,E, and F will lose access since they all share the same middleman and their middleman was never updated with the new keys and do not point to the new file object.

How do you ensure that a Revoked User Adversary cannot read or modify the file without being detected, even if they can directly access Datastore and remember values computed earlier? How do you ensure that a Revoked User Adversary cannot learn about when future updates are happening?

Ensure integrity and confidentiality with an encrypt-then-MAC scheme over the file contents. Remembered values will effectively become useless after revocation since the root key changed.

UUID	Encrypted	Key Derivation	Value at UUID	Description/
username	Yes	Symmetric derived from root key, username, and password	User Struct	new users at log-gin
username and filename	Yes	symmetric derived from rootket, username, filename	FileReference	Used to ref-er-ence a fileObj
Randomly	Yes	Symmetric created randomly	MiddleMan	Intermediary non own-ers must go for file ac-cess
Randomly	Yes	Symmetric, Randomly	FileStruct	Represents true source of con-tent for files
Randomly	Yes	Hybrid, Random	Invitation	Necessery