Sharing and Revocation

In this section, you'll design three instance methods to support sharing files with other users and revoking file access from other users.

EXAMPLE

This example scenario illustrates how file sharing occurs.

- EvanBot calls `StoreFile("foods.txt", "eggs")`.
  - Assuming that foods.txt did not previously exist in EvanBot's file namespace, this creates a new file named foods.txt in EvanBot's namespace.
  - Because EvanBot created the new file with a call to `StoreFile`, EvanBot is the *owner* of this file.
- EvanBot calls `CreateInvitation("foods.txt", "codabot")`.
  - This function returns a UUID, which we'll call an *invitation Datastore pointer*.
  - The invitation UUID can be any UUID you like. For example, you could collect/compute any values that you want to send to the recipient user for them to access the file. Then, you could securely store these values on Datastore at some UUID, and return that UUID.
- EvanBot uses a secure communication channel (outside of your system) to deliver the invitation UUID to CodaBot. Using this secure channel, CodaBot receives the identity of the sender (EvanBot) and the invitation UUID generated by EvanBot.
- CodaBot calls `AcceptInvitation("evanbot", invitationPtr, "snacks.txt")`.
  - CodaBot passes in the identity of the sender and the invitation UUID generated by EvanBot.
  - CodaBot also passes in a filename (snacks.txt here). Note that CodaBot (the recipient user) can choose to give the file a different name while accepting the invitation.
- CodaBot calls `LoadFile("snacks.txt")` and sees "eggs".
  - Note that CodaBot refers to the file using the name they specified when they accepted the invitation.
- EvanBot calls `LoadFile("foods.txt")` and sees "eggs".
  - Note that different users can refer to the same file using different filenames.

- EvanBot calls `AppendToFile("foods.txt", "and bacon")`.
- CodaBot calls `LoadFile("snacks.txt")` and sees “eggs and bacon”.
    - Note that all users should be able to see modifications to the file.

Design Requirements: Sharing and Revoking

File access

- The owner of a file is the user who initially created the file (i.e. with the first call to `StoreFile`).
- The owner must always be able to access the file. All users who have accepted an invitation to access the file (and who have not been revoked) must also be able to access the file. These users must be able to:
    - Read the file contents with `LoadFile`.
    - Overwrite the file contents with `StoreFile`.
    - Append to the file with `AppendToFile`.
    - Share the file with `CreateInvitation`.
- If a user changes the file contents, all users with access must immediately see the changes. The next time they try to access the file, all users with access should see the latest version.
- All users should be reading and modifying the same copy of the file. You may not create copies of the file.

User.CreateInvitation

```
CreateInvitation(filename string, recipientUsername string) (invitationPtr UUID, err error)
```

Generates an invitation UUID `invitationPtr`, which can be used by the target user `recipientUsername` to gain access to the file `filename`.

The invitation UUID `invitationPtr` can be any UUID value you like. For example, you could collect/compute any values that you want to send to the recipient user for them to access the file. Then, you could securely store these values on Datastore at some UUID, and return that UUID.

The recipient user will not be able to access the file (e.g. load, store) until they call `AcceptInvitation`, where they will choose their own (possibly different) filename for the file.

If the target user already has access to the file, or if the target user has already had their access to the file revoked, then this function has undefined behavior and will not be tested.

Returns an error if:

1. The given `filename` does not exist in the personal file namespace of the caller.
2. The given `recipientUsername` does not exist.
3. Sharing cannot be completed due to any malicious action.

AcceptInvitation

```
AcceptInvitation(senderUsername string, invitationPtr UUID, filename string) (err error)
```

Accepts an invitation by inputting the username of the sender (`senderUsername`), and the invitation UUID (`invitationPtr`) that the sender previously generated with a call to `CreateInvitation`.

You can assume that after the sender generates an `invitationPtr` UUID, the sender uses a secure communication channel (outside of your system) to deliver that UUID to the recipient. Using this secure channel, the recipient receives the UUID and the sender's username, and can input them into `AcceptInvitation`.

Allows the recipient user to choose their own `filename` for the shared file in their own file namespace. The recipient could choose to give the file a different name than what the sender named the file.

After calling this function, the recipient user should be able to perform all operations (load, store, append, create invitation) on the shared file, using their own chosen `filename`.

This function has undefined behavior and will not be tested if the user passes in an invitation UUID that has already been accepted.

Returns an error if:

1. The user already has a file with the chosen `filename` in their personal file namespace.
2. Something about the `invitationPtr` is wrong (e.g. the value at that UUID on Datastore is corrupt or missing, or the user cannot verify that `invitationPtr` was provided by `senderUsername`).
3. The invitation is no longer valid due to revocation.

DESIGN QUESTION

Design Question: 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?

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.)

RevokeAccess

```
RevokeAccess(filename string, recipientUsername string) (err error)
```

Revokes access to `filename` from the target user `recipientUsername`, and all the users that `recipientUsername` shared the file with (either directly or indirectly).

The owner of the file must be able to call this function. This function has undefined behavior and will not be tested if the user calling it is not the owner of the file.

The owner can only call this function on a user the owner directly shared the file with. This function has undefined behavior and will not be tested if the target user is not someone the owner directly shared the file with.

Note: This function could be called either before or after the target user calls `AcceptInvitation`. Your code should be able to revoke access either way.

After revocation, the revoked users should not be able to access the file. They should get an error if they try to access the file using your system (e.g. by calling `LoadFile`, `AppendToFile`, etc.). Note that calling `StoreFile` on a revoked file is undefined behavior and will not be tested.

The revoked users should also be unable to regain access to the file, even if they try to maliciously bypass your system and directly access Datastore.

Non-revoked users should be able to continue accessing the file (e.g. by calling `LoadFile`, `StoreFile`, etc.), without needing to re-accept any invitations.

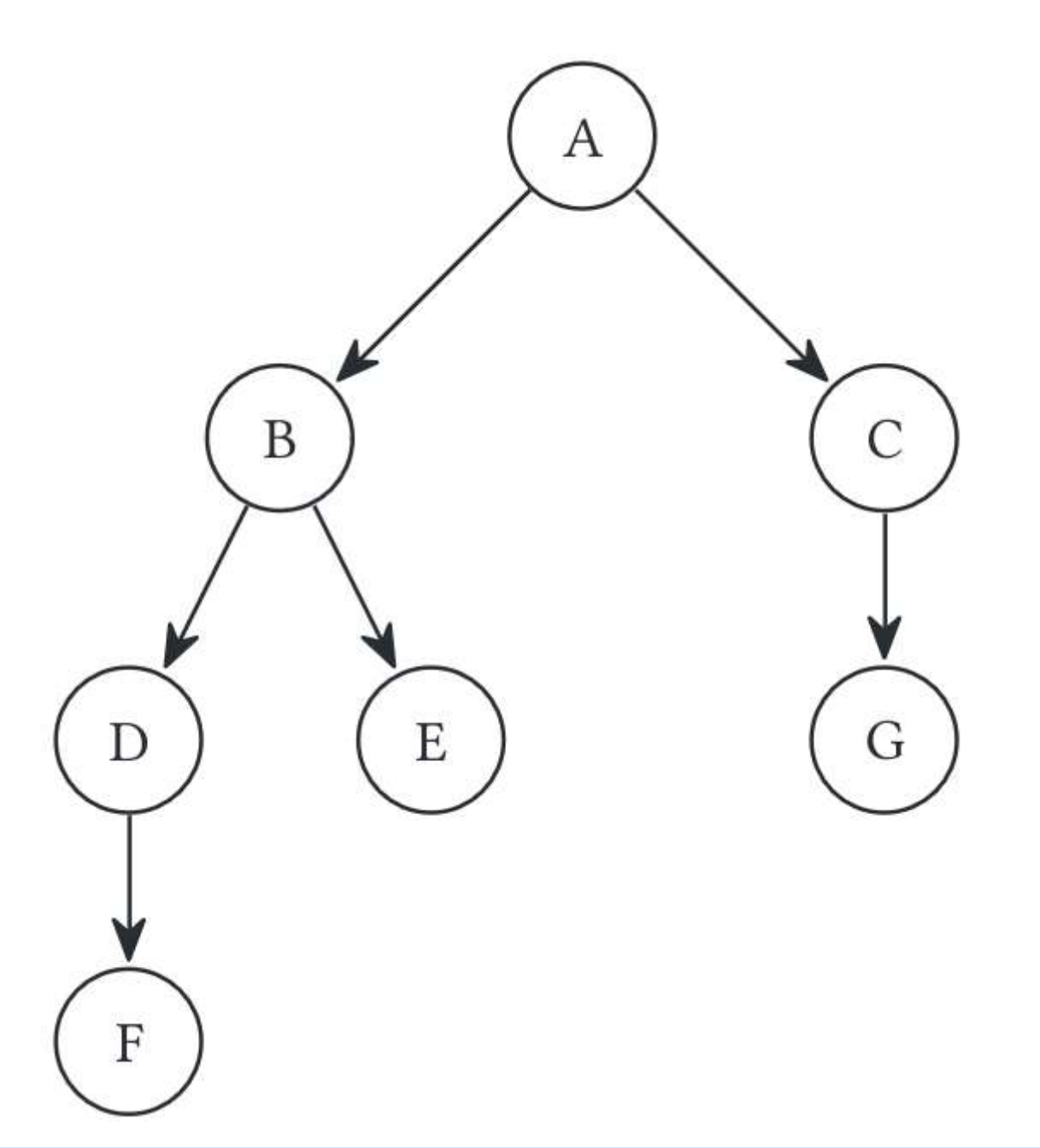
Returns an error if:

1. The given `filename` does not exist in the caller's personal file namespace.
2. The given `filename` is not currently shared with `recipientUsername`.
3. Revocation cannot be completed due to malicious action.

EXAMPLE

This example scenario illustrates revocation behavior.

Consider the following sharing tree structure. (An edge from A to B indicates that A shared the user with B.)



- A calls `RevokeAccess(file, B)`.

    - This call is defined because A is the owner. Any other user calling revoke is undefined behavior and will not be tested.
    - This call is defined because A directly shared the file with B. A can only call revoke on B and C; revoking on any other user is undefined behavior and will not be tested.
- Users B, D, E, and F should all lose access to the file.
    - If any of these users try to access the file (load, append, create invitation), the function should error.
    - These users can now be malicious: they can use values they've previously written down, and access Datastore (without listing out all UUIDs). However, they should still be unable to read the file, modify the file, or deduce when future updates are happening.
- Users C and G can continue accessing the file, without re-accepting any invitation.
    - For example, `C.LoadFile(file)` should work without re-accepting an invitation.

Design Requirements: Revoked User Adversary

Once a user has their access revoked, they become a malicious user, who we'll call the Revoked User Adversary. The Revoked User Adversary will not collude with any other users, and they will not collude with the Datastore Adversary.

The Revoked User Adversary's goal is to re-obtain access to the file. The revoked user will not perform malicious actions on other files that they still have access to. Their only goal is to re-obtain access to the file that they lost access to.

The Revoked User Adversary might attempt to re-obtain access by calling functions with different arguments (e.g. calling `AcceptInvitation` again).

The Revoked User Adversary may also try to re-obtain access by calling `DatastoreGet` and `DatastoreSet` and maliciously affecting Datastore. However, unlike the Datastore Adversary, they do not have a global view of Datastore (i.e. they cannot list all UUIDs that have been in use).

The Revoked User Adversary will not perform any rollback attacks: Given a specific UUID (that they previously had access to), they will not read the value at that UUID, and then later replace the value at that UUID with the older value they read. They will also not perform any rollback attacks on multiple UUIDs.

Prior to having their access revoked, the Revoked User Adversary could have written down any values that they have previously seen. The Revoked User Adversary has a copy of your code running

on their local computer, so they could inspect the code and learn the values of any variables that you computed.

Your code should ensure that the Revoked User Adversary is unable to learn anything about any future writes or appends to the file (learning about the file before they got revoked is okay). For example, they cannot know what the latest contents of the file are, and they should be unable to make modifications to the file without being detected. Also, they cannot know when future updates are happening (e.g. they should not be able to deduce how many times the file has been updated in the past day).

DESIGN QUESTION

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

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?

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?
