Eren Limon
Mediha Meltem Günay
Şafak Tüfekçi

DriveCloud: Development of a cloud storage protocol

Google Drive & Master Synchronization

In order to build the "Google Drive & Master Connection", we've firstly enabled the Drive API and saved "credentials.json" file to our computer. Then, we prepared the project according to the instructions from the recommended resources from project's website [1] and run the following commands to create a new project structure:

```
$ gradle init --type basic

$ mkdir -p src/main/java src/main/resources
```

By using the code given on Google Drive API's website, we've authorized the access to drive and created a command-line application with gradle. From that point, our application was always build with the following command:

```
$ gradle -q run –Pmode='master'[1]
```

However, we've modified the application given on the website and basically created a constructor with the contents of the application's *main* method. Our constructor both authorize the access and automatically create *DriveCloud* folder on the user's Google Drive.

After building the initial connection, we've created *getFileList()* and *getIDFromFileName()* methods to return the files and and their IDs given their names. Then, we've created *createFolder()* [2] method to create a DriveCloud folder on the Drive, which returns the folder's ID. We've created *deleteFile()* [2] method to delete a file from the Drive, given a file name. We've created *uploadFile()* [2] method to uploads a file to DriveCloud folder on Google Drive, given a file name and its full path on the master's folder.

```
public static void uploadFile(Drive service, String uploadFolderId, String name, String fullPath) throws java.io.IOException {
    String folderId = uploadFolderId;
    File fileMetadata = new File();
    fileMetadata.setName(name);
    fileMetadata.setParents(Collections.singletonList(folderId));
    java.io.File filePath = new java.io.File(fullPath);
    FileContent mediaContent = new FileContent(getMimeTypeFromFileName(name), filePath);
    File file = service.files().create(fileMetadata, mediaContent)
            .setFields("id, parents")
            .execute();
    System.out.println("File with ID uploaded to DriveCloud: " + file.getId());
}
```

Figure 1: uploadFile() Method

[1] If mode is needed to be set to "Follower", the following command line could be entered:
gradle -q run –Pmode='follower'

As can be seen above, *uploadFile()* method gets four different parameters as input: an authorized API client service, target folder's ID, full path and name of the uploaded file. It creates a new file, setting its name and setting its parents to the target folder. It then initializes a FileContent class with the given path and a MimeType. Then it combines both of them and uses API client to create a folder with the given parameters on the drive folder. Then, we rint its ID to check whether its unique or not.

Besides these, we've also created *getMimeTypeFromFileName()* method which returns different string including various MimeTypes, since we needed MimeType of a file for downloading a file from the drive. We finally implemented *downloadFile()* method, and at that point all the method for synchronizing DriveCloud folder on Drive and DriveCloud folder on master was ready.

```
public static void downloadFile(Drive service, String fullPath, String fileName) throws IOException, GeneralSecurityException {
    FileList result = null;
    try {
        result = service.files().list()
                .setPageSize(10)
                .setFields("nextPageToken, files(id, name)")
                .execute();
    } catch (IOException e) {
        e.printStackTrace();
    }
    List<File> files = result.getFiles();
    String fileId = "";
    for (File file : files) {
        if (file.getName().equals(fileName)) {
            fileId = file.getId();
        }
    }

    if (fileId.equals("")) {
        System.out.println("File not found!");
    }
    OutputStream outputStream = new FileOutputStream(fullPath);
    try {
        Drive.Files.Get request = service.files().get(fileId);
        request.getMediaHttpDownloader().setProgressListener(new CustomProgressListener());
        request.executeMediaAndDownloadTo(outputStream);
    } catch (IOException e) {
        System.out.println("Error on Download!");
        e.printStackTrace();
    }
}
```

Figure 2: downloadFile() Method

As can be seen above, *downloadFile()* method gets three different parameters as input: an authorized API client service, target file's path and name. First of all, it lists all the files in our DriveCloud folder on the drive. Then, it matches the file, comparing its name with the given name in the method and returns its ID. As it returns its ID, it creates a new FileOutputStream with the given path and creates an API client request. API client request works with CustomProgressListener and automatically downloads the file to the given path.

On the other hand, synchronization of both folders was fully handled in *MasterMode*[2] class. *MasterMode* class has four different array lists to control whether folders are synchronized or not.

[2] In order to handle synchronization in MasterMode class, DriveConnection class is called in its instructor and all the codes below were used, by its courtesy.

```
private void synchronize() {

    if(addedToMaster.size() > 0) {
        System.out.println("----------------case 1");
        addToDrive();
        extraFileList.clear();
    } else if(deletedFromMaster.size() > 0){
        System.out.println("----------------case 2");
        deleteFromDrive();
    } else if(driveFileList.size() < (hashOfAllFilesAtMasterDriveCloud.size() == 1 ? hashOfAllFilesAtMasterDriveCloud.size() :
        System.out.println("----------------case 3");
        deleteFromMaster();
    } else if(driveFileList.size() > hashOfAllFilesAtMasterDriveCloud.size())  {
        System.out.println("----------------case 4");
        addToMaster();
    }
```

Figure 3: synchronize() Method

Synchronize() method as can be seen above checks four different cases for four possible actions to synchronize.

1. **First Case:** A file is added to the master.

For this case, we have an *addToDrive()* method that we've run a loop that we go over the added files' array list. We only enter this case if our *addedToMaster* array list is greater than zero and this only happens when a new file is added to master's DriveCloud folder. As only one action can be handled in a one synchronization frame, our array list always has one element. For this reason, we may directly set the full path and its name while running the loop (at the first iteration). After that, we give the updated the variables to the *uploadFile()* method as inputs that we run.

2. **Second Case:** A file is deleted from the master.

For this case, we have an delete*FromDrive()* method that we've run a loop that we go over deleted files' array list. Also, we only enter this case if a file is deleted from Master's DriveCloud folder and its hash is added to the *deletedFromMaster* array list. As again only one action can be handled in a one synchronization frame, our array list always has one element. For this reason, we can directly set the deleted file's name while running the loop (at the first iteration). After that, we give the updated the variable to the *deleteFile()* method as inputs that we run.

3. **Third Case:** A file is deleted from the drive.

For this case, we have an *deleteFromMaster()* method that we've run a loop that we go over the all the files at both Drive and Master's folder in order to find the file that makes the difference. It's not possible that there's no difference between the folders as we only enter this if clause if the folders' sizes are different from each other (Master's is greater than Drive's). After we find the specific file, we create a new file with that specific file's full path and use *delete()* method to delete it. While doing that, we also update our array list (*hashOfAllFilesAtMasterDriveCloud*) that keeps track of the files in Master's folder. We used SHA_1 as the required cryptographic hash function and the related code in *MasterMode* to generate hashes was found from an online source. [4]

4. **Fourth Case:** A file is added to the drive.

For this case, we have an *addToMaster()* method that we've run a loop that we go over the all the files at both Drive and Master's folder in order to find the file that makes the difference as above. Because of the same reasons above, It's also not possible that there's no difference between the folders as we only enter this if clause if the folders' sizes are different from each other (Drive's is greater than Master's). After we find the specific file, we give that specific file's full path and name to *downloadFile()* method as inputs to download it. While doing that, we also update our array list (*hashOfAllFilesAtMasterDriveCloud*) that keeps track of the files in Master's folder.

On the other hand, we've implemented a *checkMasterAndDrivePeriodically()* method that checks whether Master and Drive folders are synchronized periodically. In this method, we update all the array lists mentioned above and run *synchronize()* method. It works with a scheduler and runs in every 30 seconds to update array lists and synchronize the folders accordingly. In order to update the mentioned array lists, we've several methods such as: *getDeletedHashes(), getAddedHashes(),* and etc.

Master & Follower Synchronization

When the user selects the FollowerMode option while building the project, a FollowerMode object is created in the main class. The constructor of this object first creates a directory on the desktop named DriveCloud. If we are running multiple followers on a single computer, the name of the newly created directory follows a sequence as "DriveCloud1", "DriveCloud2" …etc.

After the completion of the DriveCloud directory properly, the constructor initializes a property named "hashOfAllFilesAtFollowerDriveCloud". This is an array list of hashes of all files in the drive cloud folder of the follower at any time. This property is used to check if a new file is added in the DriveCloud folder or a file is deleted from the DriveCloud folder. After that, the constructor creates a connection object with the method of connectToMaster() and calls the Connect() method. This connection object is responsible for the connection to the master via both command channel and data channel and its constructor takes the server address, the server and the data ports numbers to be able to create both channels. Firstly, we only create the command connection and the data connection is created on demand by some specific methods. MasterMode accepts this command connection and creates a new thread named CommandConnectionThread which allows multiple followers to connect to the master at the same time. This class and the Connection class will be examined in detail in the following parts.

After the connection is set successfully, the follower starts to check the changes in the master and in itself periodically. To check these changes, it asks the master the added and the deleted files in the master via command channel using a Command object which will be discussed

separately, and the master sends the hashes of those files to the follower. The changes that the follower might encounter can be due to 4 actions:

1. A file is deleted[3] from the master.
2. A file is added to the master.
3. A file is added to the follower.
4. A file is deleted from the follower.

In the first action, the follower should delete the same file from itself. In the second action, the same file should be sent from the master to the follower. In the third action, the same file should be sent to the master and in the last action, the same file should be deleted from the master as well. How we do these are going to be examined one by one below:

1. **First Case:** A file is deleted from the master.

Follower periodically sends a command to the master asking whether a file is deleted or not [5]. Master compares its "hashOfAllFilesAtMasterDriveCloud" property to the current hash of all files in the drive cloud directory, decides which files are deleted, forms another command object as a result that contains the required information and sends it to the follower as a reply. Follower sees which files are deleted from the hashes and deletes them from itself.

2. **Second Case:** A file is added to the master.

Follower periodically sends a command to the master asking whether a file is added or not. Master compares its "hashOfAllFilesAtMasterDriveCloud" property to the current hash of all files in the drive cloud directory, decides which files are added, forms another command object as a result that contains the required information and sends it to the follower as a reply. Follower now has the hashes of files that needs to be added to itself.

Follower sends another command to the master asking for the full path of the added files to be able to create the file in itself with the same name.

Follower sends a third command again via the command channel, this time for the content of the file. This command establishes a data connection between master and follower to provide the data flow. Master creates a new data connection thread to handle this situation. Master sends the content of the files demanded by the follower through data connection. After data transfer, follower asks master to send the hashes of added files and compares them to the newly added files to itself. If there are files that do not match with the hashes from master, follower sends a new command to retake the files to keep the consistency of data. This data connection is closed just after the data transfer is completed successfully.

[3] Deleting file method was found from an online source. [6]

3. **Third Case:** A file is added to the follower.

Follower periodically checks whether a file is added into its local folder or not by comparing the current hashes to the hashes from the previous time frame. When it detects an addition, it first sends a command to the master to activate its listening on the data port. After the data connection is established, follower uploads its file to the master.

4. **Fourth Case:** A file is deleted from the follower.

Follower periodically checks whether a file is deleted from its local folder or not by comparing the current hashes to the hashes from the previous time frame. When it detects a deletion, it sends a command to the master saying which file should be deleted.

Connection Class

The constructor of this class takes three parameters as server address, data port and command port. This class is responsible for establishing the connection between master and follower to provide data and command transfer. When the Connect() method is called via an Connection object, the method creates a command connection with the given address and ports. Since data connection is only needed for necessary data flow (for added files), it is opened on a demand and closes just after the successful data transmission.

As mentioned in the previous section, there are 4 different cases that we can encounter. To handle these issues there are some methods in this class to send the appropriate commands to master. Since follower is the only responsible side in the synchronization with master, it uses commands to reach master's Drive Cloud folder.

CommandConnectionThread Class

The constructor of this class takes two parameters as a socket and a data port. It extends to Thread class to provide multiple command connection from different followers to the master. For every connection demand from a follower, MasterMode creates a new CommandConnectionThread object and starts it. Then this connection continuosly listens the incoming commands from followers. There are different methods available in this class to classify the incoming command type and response them accordingly.

DataConnectionThread Class

The constructor of this class takes two parameters as a socket and a file hash. It extends to Thread class to provide multiple data connection from different followers to the master. For every data connection demand from a follower, MasterMode creates a new DataConnectionThread object and starts it. This thread is only activated when there are files to transfer between the master and followers.

Eren Limon
Mediha Meltem Günay
Şafak Tüfekçi

Command Class

We created a different abstraction for sending commands. This class has two constructors, one of them only takes the type of command as an input. The other one takes type and hash of the files in an arraylist as its inputs. We created this class to use object input stream and object output stream in command transfer so that we can avoid from the issues that can be caused by buffered input stream and print writer. (e.g. when sending multiple hashes, all hashes need to be concatenated to form a single string to use buffered input stream and it becomes very hard to divide them again from the correct places.) Since the commands are shared with object streams, it becomes easier to extract the hashes and other information from the command by calling getter methods of the Command class.

Task Divison

We followed the proposed instructions in the PDF for the task division in our group. Main tasks divided to our members can be seen as below:

- Meltem: FollowerMode, Connection, MasterMode
- Eren: MasterMode, CommandConnectionThread, DataConnectionThread
- Safak: DriveConnection, CustomProgressListener, MasterMode

The report and the JavaDoc files were prepared all together.

References

[1] https://developers.google.com/drive/api/v3/quickstart/java
[2] https://developers.google.com/drive/api/v3/folder
[3] https://developers.google.com/drive/api/v2/reference/files/get
[4] https://howtodoinjava.com/java/io/how-to-generate-sha-or-md5-file-checksum-hash-in-java/
[5] https://stackoverflow.com/questions/11593725/sending-file-through-objectoutputstream-and-then-saving-it-in-java
[6] https://www.geeksforgeeks.org/delete-file-using-java/