Assignment 3

Team

James Juan Whei Tan - 40161156

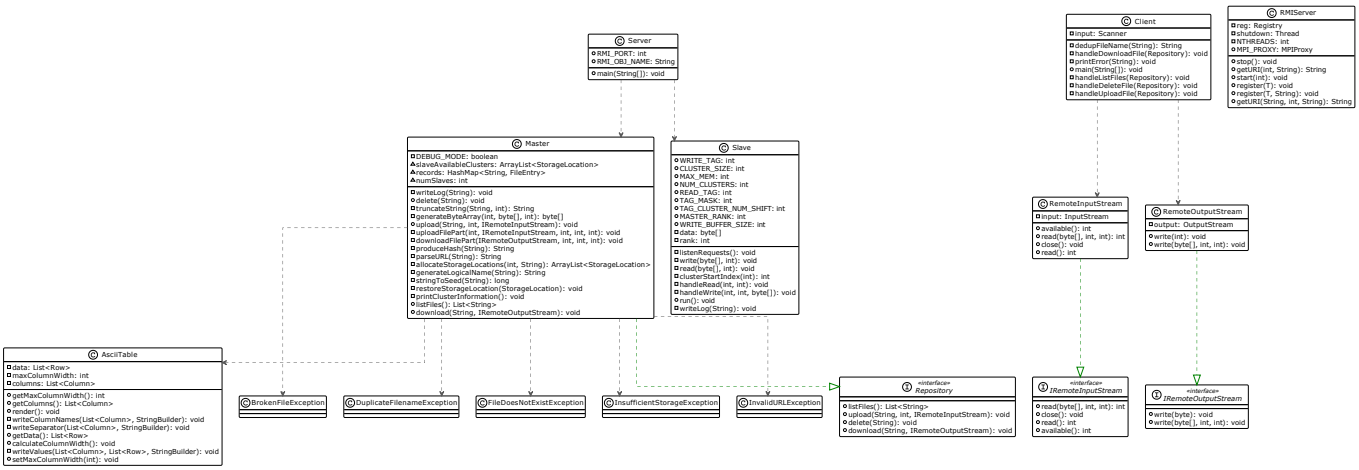
Vithushon Maheswaran - 27052715

VCS

Development of the project was carried out in a Github repository.

Class Diagram

Class diagram source



Application Design

Server

Upon launching the server, at least two processes are spawned to form our cluster (these processes communicate with each other via MPI). The first node always assumes the role of the **Master** node, the other nodes assume the role of **Slave** nodes. The functionality involved in each of these roles are encapsulated in the `Master` and `Slave` classes respectively.

Master class

The Master class also doubles as the interface that we expose to clients, i.e. clients communicate with the Master class in order to interact with the file system. This is achieved by exposing certain interfaces via RMI. As we introduce each functionality of the master node, we will also briefly describe the interfaces that are exposed and how they work.

Upload files

The Master class coordinates the transfer of files from the clients to the other slave nodes. When the client decides to upload a file to the file system, the Master node first checks if there is enough space in the Slave

nodes to store this file. It then divides up the file into parts of constant sizes and randomly decides how to distribute them among Slave nodes. Slave nodes contain a fix number of clusters, the cluster contains exactly enough space to store one file part. The Master node has an internal data structure to keep track of free clusters on each Slave node. The Master node then sends the parts one-by-one to the appropriate Slave nodes. This communication between Master and Slave nodes is achieved via MPI. Tags can be attached to messages that are sent via MPI, we leverage tags to achieve two objectives, i.e. tell the Slave node of the objective of the message (e.g. to read or write to a cluster) and to specify which cluster number we are interested in. Tags are plain Java integers, we define 2 constants in the Slave class, i.e.

```
public static final int READ_TAG = 0b001;
public static final int WRITE_TAG = 0b010;
```

We shall refer to these tags as `objective tags`. For now, they shall always be assumed to be 3 bits, which leaves us 32 - 3 = 29 bits to store the cluster number. In other words, the lower 3 bits of the tag are used to store the `objective tag`, and the the upper 29bits are used to store the cluster number. For the purposes of uploading files, the `objective tag` used is the `WRITE_TAG`.

In the message itself, the Master node sends a byte array the size of a cluster on the Slave node. Upon a successful upload of a file part, the Master node expects the Slave node to send a confirmation message (a byte array of length 0) with the same tag.

When all file parts have been successfully uploaded to Slave nodes. The Master node stores in an internal data structure the details of the file along with the location of each of its file parts on the Slave nodes. It also marks the clusters that it has just used as no longer available.

The interface that we expose to clients for the purposes of uploading files is

```
public void upload(String filename, int filesize, IRemoteInputStream
input) throws RemoteException, IOException, BrokenFileException,
InsufficientStorageException, DuplicateFilenameException,
NoSuchAlgorithmException;
```

The client is expected to create a class that extends `UnicastRemoteObject` and implements the `IRemoteInputStream` interface to pass the file to the server.

List uploaded files

Externally, the Master node expects each file to be referred to by a special URL of the following format `//magical-file-system/<file-name>`, e.g. if a file was uploaded to the system with a filename of `file.txt`, its special URL shall be `//magical-file-system/file.txt`. When listing files, the Master node returns a list of file URLs corresponding to all the files that have been previously uploaded to the system. To retrieve the list of uploaded files, the Master node simply inspects its internal data structure.

The interface that is exposed to clients for this purpose is

```
public List<String> listFiles() throws RemoteException;
```

Delete files

In order to delete a file, the Master node simply deletes the entry that corresponds to a particular file in its internal data structure. At the same time, it also marks the clusters that were previously occupied by this file as free. It does not actually wipe out the data on the Slave nodes, all it does is lose the reference to the file parts.

The RMI interface is as follows:

```
public void delete(String url) throws RemoteException,
InvalidURLException, FileDoesNotExistException;
```

The Master node expects clients to refer to the file via its special URL.

Download files

In order to download a file, the Master node has to fetch each individual file part of the Slave nodes. For the purposes of downloading files, the Master node uses the `READ_TAG` (as illustrated before) and specifies the cluster number using the tag (like before). The Master node communicates with each Slave node one-by-one in order to reassemble the original file. It expects each Slave node to send it a byte array the size of the clusters on Slave nodes.

The RMI interface exposed is as follows:

```
public void download(String url, IRemoteOutputStream output) throws
RemoteException, IOException, FileDoesNotExistException,
InvalidURLException;
```

Once again, the client is expected to provide a URL to the file and an instance of class that extends `UnicastRemoteObject` and implements the `IRemoteOutputStream` interface to receive the output file from the Server.

Slave class

Each instance of a Slave class contains a single byte array to store the data. Given a particular cluster number, the Slave node will calculate the appropriate offset into this byte array. Upon the start up of the Slave node, it begins listening to requests sent to it via MPI from the Master node. When it receives a message, it inspects the lower 3 bits of the tag in order to find out if it is a **read** or **write** request.

In the case of **read** requests, the Slave node fetches the data in the appropriate cluster and replies with a byte array containing the requested data. It uses the same tag that was sent to it by the Master node in this request.

In the case of **write** requests, the Slave node writes the data sent to it to the appropriate position in its internal byte array based on the given cluster number. At the end, it responds to the Master node with the

same tag number in order to indicate a successful write.

Client

A simple client application is provided to illustrate the functionalities of the file system. It is a command line application that invokes the methods on the Master node via RMI.