**Project- 1**

**Design Report**

*-Kaustubh Iyer*

1. **Description of project:**

* **Communicating entities:**

We establish 3 distinct communicating entities: Naming Servers, Storage Servers and Clients.

* **Communication Paradigm:**

The entities communicate via a Remote Method Invocation (RMI) library, which is layered on TCP.

* **Roles of entities:**

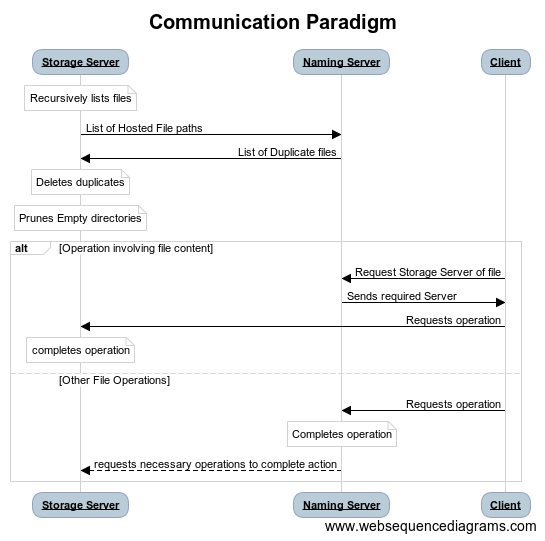
The **Naming Server** is a system that interacts with both the Clients and the Storage Servers. It performs naming related operations passed by the Client, maintains the structure of the file system with the Storage Servers and has information about the files and their locations on each Storage Server.

The **Client** is the user that needs to interact with the file system, both to make changes to it and to get files or information on files from either the Naming or Storage Servers. It does this using the RMI library to perform operations on files or the file system, interacting with the Naming or Storage Server depending on the function. If its operation does not involve the contents of the file itself, the Naming Server handles the request. If it does, then the Client asks the Naming Server for the Storage server corresponding to the required file. It then remotely invokes a function to carry out the operation on the specific storage server.

The **Storage Servers** are the entities that actually host the files. They serve information about the files to the Naming Server for organizational purposes at the beginning of the execution, and with the Clients if the Clients request for a read or write operation on a contained file.

* **Requirements of each entity over a network:**

In a typical file system, there is one Naming Server and several storage servers. The Naming servers must handle the most requests, since every operation involves them in some way. Hence the Naming Server needs to have good communicational power, as well as some computational power to be able to perform searches on the files, but not much storage power. The Storage servers need to have a lot of storage power and computational power since it needs to be able to store several files, as well as find the files and perform operations on them.



1. **Project Implementation:**

* **THE RMI PACKAGE:**

The RMI Package is a library for remote function invocations written in the TCP protocol. The package consists of two generic classes:

Skeleton and Stub, which are generically defined for any of the servers or clients to use.

* **The Skeleton Class:**

Skeletons are defined using 3 parameters: An object of the interface that handles the remote calls, an object that executes this interface, and a port number (if applicable). On occasion we would like to start a skeleton with a defined port number, but if we choose not to, then the class automatically allocates one when the start() function is called. Initializing the Skeleton with no interface or implementation of the interface results in an Exception being thrown. Initializing the skeleton with an invalid interface of remote invocation also results in an Exception.

The start() function opens the port for listening if it is provided in the construction of the object, else opens any system allocated port for listening. Whenever it accepts a connection, it creates a new thread to interact with the client (stub). The thread is created using a service that implements Runnable, and the run function of this class opens a service socket in a new port to interact with the stub. The function call object it receives from the stub is broken into the method name, the argument types, and the arguments, and this method is invoked from the implementation interface. The resulting object from this call is written out to the client.

The stop() function simply stops the listener’s loop. This does not stop the threads of the clients that are being served currently, which end by themselves after having served.

* **The Stub Class:**

There is only one function to implement in stub, which is create(). We implement create by creating an object of the implementation type as an instance of a dynamic Proxy. This is all we need to do, and the user can use this defined data member to invoke any function. An invocation leads to the call going to the invocation handler that we must define and pass to the dynamic proxy. The invoke function takes in a method name, arg types (array of classes) and the arguments (array of objects of these classes). It starts by checking if the method is local, in which case it can simply call the function. If it is not local, then it must create a socket and connect to the Skeleton inputted in the class definition of the Stub. It then sequentially writes out the method name, argument types and arguments, and reads the result of the call from the skeleton.

Once it has received the output, it closes the connection, and returns the object it read from the skeleton.

* **THE COMMON PACKAGE:**

The Path functions are helpers used by both the Naming and the Storage Servers to create and work with the paths of the files they either send or receive. A path is a specialized string that specifies the relative location of the directory it represents in the file system. Iterator<String>() is a function that returns the separator of each member in the path to a directory. The list() function goes over the directory tree and returns all the paths of every file in the file system.

isRoot() checks if a given path represents the root directory. The parent() function returns, for a path, the path of its parent directory. The last() function returns the current folder name. This returns nothing in the case of a root, since it is not represented by a name. IsSubPath() checks if a path is a subpath of another. This checks if the current path exactly prefixes the other one.

* **THE NAMING PACKAGE:**

The Naming server consists of a registration and a service interface, which are implemented by the server. The main purpose of the Naming server is to build the directory tree. We build an object representing the directory as a tree containing children as an array of directories. Each file does not contain any children and each directory contains children. We keep some basic functionalities in this class, so we can identify if a directory is a file or a folder, what its overall path is, etc.

The Registration interface has a function register, which serves to register the files from the Storage servers into the Naming server. This is done by opening a Storage skeleton, and waiting for Storage stubs from the storage servers to pass their files in. Once we receive their files, we check for duplicates, and return a list of duplicates to the Storage servers to delete (this is all done within the thread that receives the files from the Storage server, in the run function). Using the files that we get from these storage servers, we build the directory tree, as the object of the class that we created.

The service interface’s functions are implemented keeping in mind that the registration is complete and assuming we have a complete directory tree as our attribute. isDirectory() checks if a given path exists in the directory tree or not. This is done by traversing the tree, searching the subtree at each step until we either find the folder or we don’t. We return true or false accordingly. List() lists the files and folders in the given directory. It does this by traversing the tree to that directory, and then returning the subtrees in that directory. createFile() creates a new file as the given path, and allocates it arbitrarily to a storage server. createDirectory() creates a new directory, adding it as a member in the parent directory’s array after traversing there. getStorage() return the storage stub associated with the given file so that the client can interact with it.

* **THE STORAGE PACKAGE:**

The Storage server consists of a command and storage interface, which are implemented by the server. The main purpose of the storage server is to contain and manipulate the files of the file system. The root File object is passed in when creating an instance of a storage server. This is the directory of the root of the local file system. The start() function starts up the Storage server. It takes in a Naming Server registration interface. A Registration Stub is created, which connects to the Registration Skeleton of the Naming Server. We then remotely invoke register, passing in the list of files that we get from our file system (which is done by invoking list on the file root). Now the resulting path[] that we get must be deleted from the file system, as these are the duplicate files. Once this is done, we purge the directories that are empty to avoid redundancies.

The Command interface contains a function create, in which we must create a new file using the file’s function, in the specified directory.

The Storage interface contains three functions: size, read and write. Read opens the file of the given path and reads it using the File object’s functions. Write and size do the same.