15-440: Project 1
Remote File Storage and Access Kit (FileStack)
Using Sockets and RMI

Assigned Date: 1 Sep 2014
Design Report Due Date: 13 Sep 2014
Final Due Date: 1 Oct 2014

Table of Contents

1. Project Objective

The objective of this project is to apply the knowledge of **client-server communication** and **Remote Method Invocation (RMI)** to build a distributed file system, which we refer to as Remote File Storage and Access Kit ***(FileStAcK***, or simply ***FileStack)***. RMI involves the creation of stubs and skeletons at client and server sides, respectively which allow for transparent location, reading and writing of files maintained at networked computers. In RMI, the underlying details are generally hidden from users, whereby a calling object can invoke a method in a potentially remote node as if it is local.

2. FileStack

The following subsections will provide an overview of FileStack, followed by descriptions of its inherent functionalities, entities, architecture, communication between entities and the required RMI library and interfaces.

2.1 Overview

In this project, you will implement *FileStack*, a distributed file system that stores a vast amount of data (files) which typically do not fit on a single machine. Briefly, the files are physically stored on a set of servers called ***Storage Servers***. Users, referred to as ***Clients***, can create, delete, read, write, and list files (among others), all via using Remote Method Invocation (RMI). As a requisite step, Clients need to identify Storage Servers that host the required files. They do so with the help of a mediator. Clients contact a ***Naming Server*** (in this project we allow only one centralized Naming Server) which maps every file name to a Storage Server. The Naming Server is thus a repository of *metadata* or data about data.

2.2 Functionalities

The operations (or functionalities) that are available to the Clients of FileStack are:

1. *CreateFile(path)*: create the file referred to by *path*[1].
2. *CreateDirectory(path)*: create the directory referred to by *path*.
3. *Read(path, off, n)*: read *n* bytes of data from the file referred to by *path* starting at an offset *off*.
4. *Write(path, off, data)*: write *n* bytes of data to the file referred to by *path* starting at an offset *off*.
5. *Size(path)*: return the size, in bytes, of the file referred to by *path*.
6. *IsDirectory(path):* return true if *path* refers to a directory.
7. *List(path)*: list the contents of the directory referred to by *path*.
8. *Delete(path)*: delete the file or directory referred to by *path*.
9. *GetStorage(path)*: get the Storage Server (or more precisely a representing *stub*) hosting the file referred to by *path*.

In this project, you will implement all the above-listed functionalities except the *Delete(path)*

[1] *Path* is a string that refers to an absolute path (with "/" being the root) of a file or a directory in the Naming Server's directory tree.

function, which is left as a bonus to implement.

2.3 Entities, Architecture, and Communication

The main entities in FileStack are: **Clients**, **Naming Server**, and **Storage Servers**. A *Client* is the end-user of FileStack who wishes to perform operations on files. We assume that a Client knows the path of a file it wishes to manipulate. The *Naming Server* is a critical entity in FileStack because it is the means by which Clients locate files stored at Storage Servers. It runs at a pre-defined address that is known by both, Clients and Storage Servers. A Storage Server physically stores files in its local file system. The project assumes that a file cannot be partitioned across Storage Servers (i.e., no file striping is applied) and that a Storage Server can host multiple files.

FileStack is based on a client-server architecture as shown in Figure 1. In this architecture, a client is a service requester and a server is a service provider. Servers also behave as clients when requesting

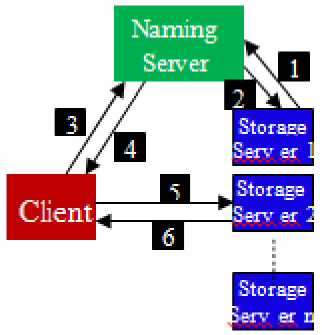

Figure 1: Architecture of File Stack

services provided by other servers. For example, the Naming Server behaves as a client when requesting the services of a Storage Server. In Figure 1, all forward arrows (arrows 2, 3, and 5) represent requests originating from clients to servers, and the corresponding backward arrows represent the services provided by the servers in response.

It is evident from the arrows in Figure 1 that distributed systems entail a lot of communication between its entities. FileStack is no exception. We will now discuss the purpose of communication in FileStack and the communicating entities.

<u>Storage Servers – Naming Server Communication:</u> Upon start-up, each Storage Server sends a list of paths (representing the files that it currently hosts) to the Naming Server, a process we denote as *registration*. The Naming Server then traverses this list and adds the paths to its directory tree where each leaf node is a (filename, Storage Server) tuple. During the traversal, the Naming Server remembers paths of existing files and ultimately replies back to the Storage Server with a list of duplicate paths (see Section 5.3 for details on that). Figure 1 depicts this communication with arrows 1 and 2. After registration by all Storage Servers, the Naming Server is deemed to be capable of locating all files stored at each Storage Server.

<u>Client – Naming Server Communication:</u> Arrows 3 and 4 in Figure 1 illustrate this communication. A Client contacts the Naming Server whenever it needs to perform an operation on a file. While some operations cannot be directly handled by the Naming Server, in which case it replies back with the Storage Server that hosts the file, other operations can be directly handled by it.

Operations requiring the content of a file namely read, write and size, cannot be directly handled by the

Naming Server. When a Client wishes to perform any of those operations, it first contacts the Naming Server to get the Storage Server (or more precisely a representing *stub*) that hosts the file. It does so using the getStorage operation and, subsequently, communicates with the respective Storage Server to handle the operation. In Figure 1, arrows 5 and 6 mark this communication.

All the other operations namely createFile, createDirectory, isDirectory, list, and delete, can be handled by the Naming Server without involving the Storage Servers. This is because the Naming Server merely leverages its tree to handle such operations without requiring file contents. It also ensures the integrity of the Naming Server's directory tree. Clients cannot create/delete files/directories on the Storage Servers without the awareness of the Naming Server. Therefore, the Naming Server first updates its directory tree by adding/deleting a file/directory and then instructs the respective Storage Server to perform the physical creation/deletion.

You have to create a design that supports the delete operation without actually implementing it. The implementation of the delete operation is a bonus-question in this project but a requirement in Project 2.

Naming Server – Storage Servers Communication: This communication is illustrated in Figure 1 by arrows 1 and 2. Upon start-up, each Storage Server recursively lists its hosted files and sends the resultant list of paths to the Naming Server. As described earlier, we denote this process as *registration*. In response, the Naming Server replies with a list of duplicate files (if any) which a Storage Server deletes from its local file system. In addition, it also deletes any directories that are rendered empty, a process known as *pruning*.

Besides their communication upon boot-strapping FileStack, the Naming Server sends create and delete operations to Storage Servers on behalf of Clients after changes to its directory tree have been successfully committed.

Client – Storage Servers Communication: Arrows 5 and 6 illustrate this communication that occurs when a Client wishes to perform a read, write, or size operation, after retrieving the respective Storage Server from the Naming Server (see Client – Naming Server Communication above).

2.4 RMI Library

As explained earlier, Clients request the Naming and the Storage Servers to perform operations (methods) on files. The Naming Server also behaves as a Client when it requests the Storage Server to perform create/delete operations. The recipient server, in return, fulfills requests by executing the operations' logic and returning results. This implies that servers can transiently act as clients when requesting the services of other servers. We use the terms *client* (with a lower-case *c*) and *server* (with a lower case *s*) to denote any service requester and provider respectively. For the rest of the document, we will also use the term "*invoking a method*" when we actually mean "requesting to perform an operation".

When a client invokes a method, it essentially invokes a remote method, hence, the name *Remote Method Invocation* (since the method's logic resides on a server). The client is only aware of the method's name, not where it actually resides. To enable a client to execute a remote method, we would require an RMI library. RMI library takes care of initiating client connections to the appropriate servers and forwarding method invocations to them, thereby, making them appear to clients as if the methods are implemented locally. At the recipient servers, the RMI library receives client connections, invokes the requested methods, and returns results. The servers execute the methods while being totally oblivious to the fact that the invocations were initiated by remote Clients. Thus, the RMI library helps masking the client-server communication.

An RMI library is based on the concept of *Stubs and Skeletons*. When a client needs to perform an operation, it invokes a corresponding remote method via an object called the “stub”. The stub object, or simply the stub, is part of the client and is responsible for handling the invoked method, be it local or remote. If a method is local, the stub merely invokes a helper function that implements the “logic” of the operation. On the other hand, if the method is remote, the stub initiates a connection to the appropriate server (more precisely the server’s skeleton), *marshalls*[2] the method name and arguments, transmits the byte stream over the network, *unmarshalls*[3] the result and returns it to the client. Thus, stubs allow clients to invoke local and remote methods alike, leaving the underlying complexity associated with remote methods to the stub.

Skeletons are counterparts of stubs but reside reversely at servers. Each stub communicates to a skeleton. A skeleton is an object responsible for listening to multiple client connections, unmarshalling the byte stream, invoking the method implementing the logic of the requested operation, marshalling the results, and sending them back to the client.

2.5 Interfaces

A server usually declares all the methods it handles in interfaces. An interface contains a subset of the methods that can be invoked by a particular client. For example, the Naming Server declares two interfaces, one for methods that can be invoked by Clients and the other for methods that can be invoked by Storage Servers. Segregating declarations into multiple interfaces ensures that clients can only invoke their permissible methods. In FileStack, the Naming Server splits its method declarations across two interfaces:

- *Registration:* defines a single method, namely *register,* invoked by Storage Servers upon bootstrapping FileStack.

- *Service*: defines the methods that can be invoked by Clients and handled directly by the Naming Server. As described in Section 2.3, the methods are: *getStorage, isDirectory, list, createFile, createDirectory,* and *delete.*

Similarly, Storage Servers split their method declarations into two interfaces:

- *Command:* defines two methods *create* and *delete,* which can be invoked by the Naming Server whenever a Client requests any of the createFile, createDirectory or delete operations. For any of these operations, the Naming Server basically commands a specific Storage Server to alter its local file system accordingly.

- *Storage:* defines the methods that can be invoked by Clients and handled only by Storage Servers. As described in Section 2.3, these methods are: *size, read* and *write*.

[2] *Marshaling* is the process of converting a datum (e.g., an object) into a byte stream that can be transmitted over a network.

[3] *Unmarshaling* is the reverse process of marshaling, whereby a datum or an object is reconstructed from a byte stream.

For each interface, a stub and a corresponding skeleton are required. A stub uses an interface to determine if an invoked method is remote or local, and subsequently performs tasks as described in Section 2.3. The corresponding skeleton uses the interface to verify if an invocation is legitimate (i.e., the invoked method belongs to the interface). If so, the skeleton acts as described in Section 2.3.

To create a stub, a client requires the following information: an interface and the network address (IP and port) of the corresponding skeleton (or more precisely, the IP address of the Storage Server at which the skeleton exists). Similarly, to create a skeleton, a server requires the following information: an interface, a class that implements the logic of the methods defined in the interface, the server's IP address, and a port number. Essentially all skeletons belonging to a server possess the same IP address but different port numbers.

The Naming Server creates two skeletons, one for each interface. We call these skeletons **RegistrationSkeleton** and **ServiceSkeleton**, which correspond to the Registration and the Service interfaces respectively. In any distributed system, stubs that communicate with a known server, such as the Naming Server, are created by clients at boot-up. In other words, since the network addresses of RegistrationSkeleton and ServiceSkeleton are predefined, each Storage Server and Client, at boot-up, creates RegistrationStub and ServiceStub respectively.

Likewise, a Storage Server creates two skeletons, one for each interface. We call these skeletons **CommandSkeleton** and **StorageSkeleton**, which correspond to the Command and the Storage interfaces, respectively. A problem, however, is that a Storage Server may run on any machine and its address (and, thereby, the addresses of its skeletons) is not predefined. This implies that neither Clients nor the Naming Server can create the corresponding stubs to communicate with Storage Servers. To resolve this issue, each Storage Server creates its CommandStub and StorageStub. During registration, the Storage Servers transmit their stubs to the Naming Server along with their list of files. Hence, when a Client invokes the getStorage method, what the Naming Server actually returns is the CommandStub of the Storage Server which hosts the requested file. Similarly, when the Naming Server needs to communicate with a Storage Server, it uses the respective StorageStub.

3. Recap

Let us revisit the whole problem at a high level. The main issue is to enable Clients to perform operations on files stored on remote servers in a distributed file system. Technically, we refer to the act of performing an operation as invoking a *remote* method. We denote a method as remote because the "logic" of the method is implemented on remote servers, namely the Naming Server or the Storage Servers. In order to trigger a method at a server and return its results, we need *stub* and corresponding *skeleton* objects. A client may either possess a stub (created at boot-up) or may have acquired it from the Naming Server. A server creates its skeletons at boot-up using its pre-defined interfaces.

4. A Working Example

Consider the situation in which a Client wishes to perform a read operation on a file named "abc" (assume the name “abc” is of type String). To do so, the Client requires StorageStub to communicate with StorageSkeleton of the Storage Server hosting the file "abc". So as a first step, it must contact the Naming Server to get StorageStub. As such, it invokes getStorage(abc) using ServiceStub. ServiceStub at the Client side connects to ServiceSkeleton at the Naming Server, marshalls the method name (i.e., getStorage), argument types (i.e., String) and argument values (i.e., “abc”), transmits the byte stream, unmarshalls the result (after finally received from the Naming Server) and closes the connection. ServiceSkeleton listens for incoming connections, accepts the new connection, unmarshalls the incoming stream to identify the requested method, calls the locally defined getStorage(abc) function, marshalls StorageStub object and returns it to ServiceStub.

Now that the Client has StorageStub, the second step is to contact the Storage Server. The Client invokes read(abc,0,10) using StorageStub which in return communicates with StorageSkeleton. StorageStub and StorageSkeleton interact in a similar manner as ServiceStub and ServiceSkeleton, and the Client eventually acquires a buffer containing 10 bytes of data read from the file "abc". The entire example is demonstrated in Figure 2.

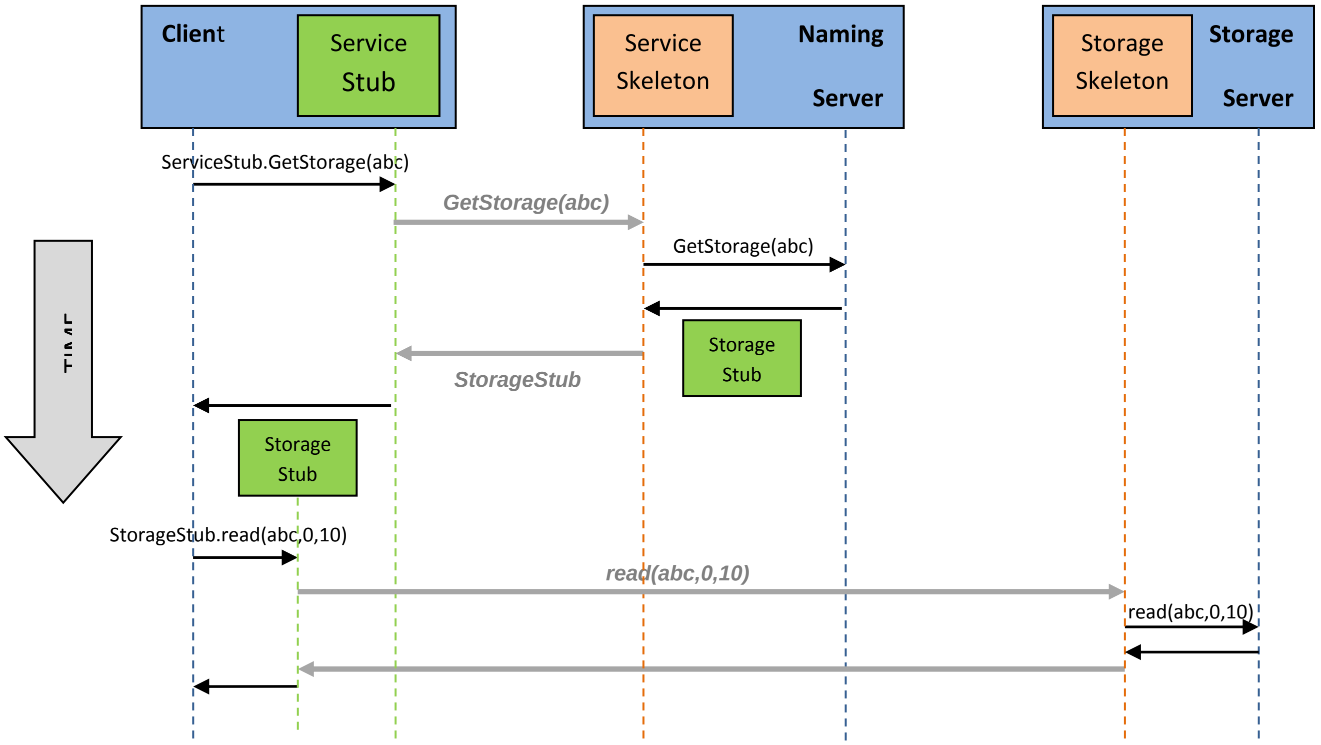

Figure 2: An example of a Client performing a read operation on file 'abc'.

5. Implementation

In this project you will be using the Java Programming Language. You are provided with a starter code *P1_StarterCode.zip* that contains five primary packages namely *rmi*, *naming*, *storage*, *client*, *and common*, which you must implement to create a fully-functional distributed file system (i.e., FileStack). We recommend that you implement the packages in the following order, (1) rmi, (2) common, (3) naming, and (4) storage. The client package has been fully implemented for you.

In this section, we provide you with design considerations for successfully implementing each package. In addition, it is important that you read the package-info document under each package prior to implementing
the package.

5.1 The RMI Package

The RMI package is your RMI library. The RMI library consists of two generic (paramterized) classes:

Skeleton and Stub. Both, the Skeleton and the Stub classes take a *remote interface*[4] as a parameter. They define and implement methods that are common to all skeletons and stubs in FileStack (e.g., constructors to instantiate skeleton/stub objects as well as start() and stop() methods to start and stop skeletons/stubs, respectively).

The connection and communication between stubs and skeletons are carried out using Java API for TCP socket[5] programming. The skeleton is multithreaded. When it is started using the start() method, its main thread creates a listening socket (see Figure 3) which waits for incoming client connections. Once a client's request is received, the skeleton accepts the request, creates a new thread (or what we call a *client thread*) to service the request, and instantiates a new service socket within the client thread to handle further communication with the client. Figure 3 illustrates the concept of stubs and skeletons using multithreaded socket programming.

A stub is implemented in Java as a *dynamic proxy* (java.lang.reflect.Proxy). A proxy has an associated *invocation handler*. When a method is invoked on a proxy (stub) object, the method name and parameters are encoded and dispatched to the *invoke* method of the invocation handler. For instance, in Figure 2, when the client invokes the method *getStorage* on the proxy *ServiceStub,* the method name and arguments (i.e., *getStorage* and *"abc"*) are encoded and dispatched to the *invoke* method of the

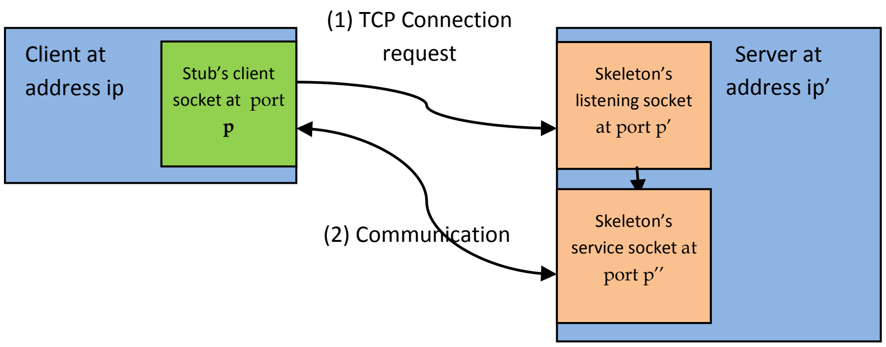
Client at address ip
Stub's client socket at port p
(1) TCP Connection request
(2) Communication
Skeleton's listening socket at port p'
Skeleton's service socket at port p''
Server at address ip'

Figure 3: Sockets and Multi-threading.

ServiceStub's invocation handler. The invoke method checks whether the invoked method is local or remote. In the case of a local method (i.e. a method that the proxy implements), the local method is simply invoked and passed the arguments. In this project, local methods are *equals(), hashCode() and toString().* The equals method determines if two proxies were created for the same skeleton. HashCode returns the hash code of a proxy object. ToString prints information about a proxy. In the case of a remote method, the proxy connects to the corresponding skeleton at the server side, marshalls the method name, parameter types and values, and sends the entailed byte stream. You can read more about implementing a dynamic proxy at http://tutorials.jenkov.com/java-reflection/dynamic-proxies.html.

[4] A *remote Interface* is a Java interface in which each method declared in it throws an exception of type RMIException.

[5] A *socket* is an end-point in a bidirectional communication between two processes running typically on two separate computers on a network. Each socket is bound to a different port number which is utilized by TCP to identify the data destined to the socket.

The RMI library is best implemented in two phases:

Phase 1: learn more about the two Java socket APIs (Socket and ServerSocket) and dynamic proxies (java.lang.reflect.Proxy) to implement a basic stub-skeleton communication. Use ObjectOuputStream and ObjectInputStream which allow writing and reading primitive data types of Java objects (or referred interchangeably to as *serializing* data) to output and input streams, respectively. At the end of this phase, a stub should be able to connect to a skeleton, and send and receive objects. Similarly, a skeleton must be able to accept several client connections, read objects and send back objects (including exceptions).

You should make the server as robust as possible via handling various types of exceptions and displaying meaningful messages whenever needed, so as to avoid abrupt crashes. For example, if the stub fails to create an input stream, it should throw an RMIException with a meaningful message. You should also clean-up (i.e. close sockets and streams) whenever required.

Phase 2: Now that you are able to send and receive serialized objects (using ObjectInputStreams and ObjectOutputStreams), it is time to send and receive more meaningful data. The stub should now check whether the invoked method is local or remote using its defined interface. When the invoked method is local, the stub simply invokes a corresponding locally implemented method (or what we call an *implementor* method). On the other hand, when the invoked method is remote, the stub sends the method name and parameter types and values to the respective skeleton. The skeleton should receive them, invoke the corresponding method, which implements the necessary logic, and send back the generated result to the stub. Exceptions that arise due to unsupported methods during unmarshalling, and the ones that are thrown by the implementor methods should be communicated back to the client.

5.2 The Naming Package

The naming package contains the Registration and Service interfaces, as well as the NamingServer class that creates the necessary skeletons and stubs and implements the logic of all the operations handled by the Naming Server. For the create operation, we leave the strategy of selecting a Storage Server to host a new file or directory up to you.

The Naming Server creates and maintains the FileStack directory tree, with the top-level directory being the root represented by the path "/". While the inner tree nodes represent directories, the leaves represent files (or more precisely file,stubs tuples). The Naming Server gradually builds its tree during registration. After registration, the Naming Server uses its tree to handle operations. It is important to design the directory tree in a way that allows the Naming Server to easily look-up, traverse and alter the tree, as well as detect invalid paths (e.g., a path that denotes a non-existing file).

5.3 The Storage Package

The storage package contains the Command and Storage interfaces, as well as the StorageServer class that creates the necessary skeletons and stubs and implements the logic of all the operations handled by the Storage Server.

Each Storage Server has its own local file system. The files hosted by a Storage Server are stored in its local file system in a directory denoted as *temporary directory*. A temporary directory and all its sub-directories and files are part of FileStack i.e. part of the Naming Server's directory tree. Each file/directory at a Storage Server has an absolute ***local path*** with respect to the root directory of the Storage Server. In other words, a local path dictates the location of a file/directory at the local file system of a Storage Server. Paths we have been referring to so far in the document are relative paths that dictate the locations of files/directories on FileStack. We call these paths ***FileStack paths***. FileStack paths refer to the directory tree of the Naming Server and are prefixed with the root directory of FileStack (i.e., "/") followed by the temporary directories of Storage Servers. To exemplify, consider a file *f1.txt* stored in a temporary directory, *tmp*, at Storage Server, *SS1*, with an absolute path of */home/SS1/Public/tmp/sub_dir1/f1.txt*. Therefore, the local path at SS1 will be */home/SS1/Public/tmp/sub_dir1/f1.txt*, while the FileStack path at the directory tree of the Naming Server would be */tmp/sub_dir1/f1.txt*. Clients and the Naming Server know and use FileStack paths only. The job of resolving (or mapping) FileStack paths into local paths (and vice-versa) is the responsibility of Storage Servers.

During registration, the Storage Server recursively lists the contents of its temporary directory and sends the list of local paths (of files only) along with its stubs to the Naming Server. The Naming Server maps the received local paths to FileStack paths and sends back a list of *duplicate files* for deletion. Duplicate files are files that have been already registered and therefore exist in the Naming Server's directory tree. For instance, during the registration of Storage Server *SS1*, if the Naming Server encounters *f1.txt* sent by SS1 in its directory tree, then *f1.txt* is deemed a duplicate file.

In Java, directories and files are represented as java.io.File objects. Peruse through the documentation of java.io.File to understand more the File’s constructors and methods.

5.4 The Common Package

This package contains the class Path that defines utility functions which manipulate paths. These functions are used as helper methods by the Naming Server and the Storage Servers.

6. Test Suite

We have provided four packages namely rmi, common, naming and storage, containing test files that test the corresponding package. We recommend that you test your packages as you move forward. The test cases test if your implementation conforms to the design specifications, and checks the correctness of the implementation. Please note that this is a service offered to help you design and test your code faster. You are solely responsible to ensure that your implementation is flawless. During grading, we may use other test cases to make sure that your project works as expected.

7. Tips

- Start early!

- Read about Java multi-threading and synchronization that is used to synchronize accesses to shared variables. When multiple threads invoke a method that alters shared variable(s), you must ensure that the method is declared as *synchronized.*

- Learn more about Java libraries used to access and manipulate a file system and its constituent files/folders.

- Read the package-info provided in each package to understand the package’s functionalities and how to use or integrate it with the other packages.

- For each package of the four packages (i.e., the rmi, naming, storage, and common packages), it is important to read the respective test files found under the *conformance* package. This will give you an idea on how to design and implement a package.

- Do not defer testing until the end. Test your packages as you go.

8. Q&A

We use Piazza as a platform for asking questions and receiving answers. Posting your questions on Piazza will help the whole class benefit and will certainly avoid redundancy. Find our Piazza page at: https://piazza.com/qatar.cmu/fall2014/15440/home

9. Deliverables

There will be two deliverables:

1. **Design Report Deliverable:** you have to submit the detailed design of the project in this report. The deadline is Sep 13, 2014. The design document should contain the following sections:
   - A brief design of the project deduced from the starter code: The starter code provides a framework for the project. Identify all the activities that occur in the project (e.g., Naming Server starts up, Client requests a file for reading its content, etc). Draw sequence diagrams of these activities (Note: A sequence diagram is a UML methodology for representing how objects communicate and in what order). Do not elaborately discuss the items that are already stated in comments (or java-doc).
   - Description of the logic of the unimplemented functionalities: The starter code has many unimplemented functionalities. The missing parts are spread over the starter code. You can identify them by searching for parts of code that throw an "*UnsupportedOperationException*". Discuss the logic that you plan to incorporate to implement these missing functionalities. The description should discuss the logic in detail (including trivial errors, such as the action taken if a null value is passed as an argument to a function). Do not elaborately discuss in text; use appropriate flow-charts and sequence diagrams, if and when needed.

2. **Final Deliverable:** a zip file containing the source code. Please adhere to the same package and directory structure as provided by the framework (i.e., RMI library, Naming Server, and Storage Server, and test cases in separate directories). If you want to alter this structure (for example, to improve the framework), please let the instructor know. You need a written approval from the instructor before modifying the framework. You are; however, free to add files within the existing packages.

10. Submitting your project

Submit your code using AFS (Andrew File System): /afs/qatar.cmu.edu/usr10/mhhammou/www/15440-f14/handin/project1/*userid*/, where *userid* is your andrew ID.

11. Late policy

- If you hand in on time, there is no penalty (duh!).
- 0 -24 hours late = 25% penalty.
- 2 4 -48 hours late = 50% penalty.
- M o r e than 48 hours late = you lose all the points for this project.

NOTE: You can use your grace-days quota. For details about the quota, please refer to the syllabus.