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

Design Report Due Date: 14 Sep 2011

Final Due Date: 3 Oct 2011

Table of Contents

Learning Objective

The objective of the project is to apply the knowledge of networking, and Remote Procedure Calls (RPCs) to write a program that enables transparent method calls. Such calls allow transparent invocation of methods on processes that reside on a different system, and is a widely used concept in providing location transparency in distributed systems. This project is mainly in-line with learning objective ILO 1.2: "*Recognize the right communication architecture and remote calls for a particular application, and apply the knowledge to design simple communication mechanisms*".

Project Objective

The project will implement a Distributed File System, which we refer as *Remote File Storage and Access Kit (File StAcK*, or simply, *File Stack).* In this project, local clients access remote files using Remote Method Invocations (RMIs). You will provide clients with functions that enable accessing files that are stored on remote machines. In particular, you will provide the following functionalities in distributed file systems:

1. Create/Read/Write files: Enable clients to call local methods to create, read or write files. You will support binary and ASCII reading and writing of files.
2. Create/list-files in directories: You will write functions for supporting meta-file operations. You will implement RMIs for creating directories (createDirectory), listing files (list) and a function to return if the provided path points to a directory (isDirectory). The create function will take only absolute paths (no relative paths) in this project. The list function will list the files in a directory, provide detailed information about the files, such as file-sizes.

Conceptual architecture

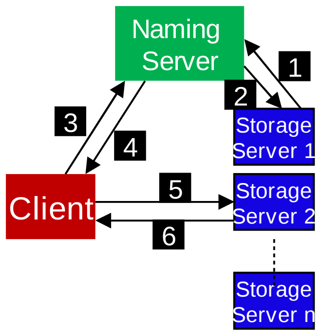

Figure 1: Architecture of File Stack

File Stack system is described in Figure 1. The system consists of multiple file *storage servers*. Each storage server is responsible for storing a set of files. There is a single registry server, called *Naming Server*, that has meta-data about file storage (which files are stored in which servers). A naming server maps file to a storage node using hashes. Each storage server will first register to the naming server about its presence during its boot-up procedure (Arrows marked 3 and 4).

Clients are users of the file system. *Clients* can create, read and write files and directories. For any file operation the client first resolves the storage server where the file resides by contacting the naming server (Arrows marked 5 and 6), and then directly contacts the storage server (through RMI) to perform the file operation.

System Description

In this section, we describe the functionalities supported, the main entities, and the communication mechanisms between various entities.

File operations

You will implement the following file operations for distributed file systems:

1. *Create file* call*:* Create a file given a *path* (path is a absolute path string for the file).
2. *Read* call: The client should be able to read "n" bytes of data from an offset "offset" for a file given by a *path*.
3. *Write* call: Each write call will write the given "n" bytes of data to the file, starting from the offset "*offset*".

4. *Create directory* call: The create call creates a directory on a respective file storage server. The create function will take only absolute path (no relative path).
5. *Size* call: Given a file, the call will return the size of the file.
6. *List files*: The list call will input a directory or a file name (full path), and then list the directories and files under it (if present).
7. *Get Storage server* call: Get the storage server in which the file is stored.

Entities

The three main entities are the naming server, storage server and the client.

Naming Server

The naming server is responsible for storing the meta-information of the file system. It stores information about which files are stored on which storage servers: it maintains the meta-information about the common directory tree.

Naming server has four important functionalities:

1. Storage Server registration: Upon startup, the storage servers inform the naming server of which files they are initially able to serve. This process is known as *registration*. There is no more direct communication between naming server and storage servers after this.
2. Client redirection: All clients are assumed to know the identity (IP address and port) of the naming server – this is the only information they have about the file-system components. Hence, if the client has to read/write a file, it first contacts the naming server and requests for the storage server information that will process its read/write function. The naming server looks-up the storage server that physically stores the file, and passes the information of the storage server to the client.
3. Client service for file meta-information: Since the naming server has all the meta-information about the file-system, the meta-operations (createFile, createDirectory, isDirectory, list and delete) can be handled directly by the naming server. Note that the "delete" command has to be enabled. However, delete implementation is a bonus-question for project 1; it is not necessary to implement the logic for delete in project 1. Other operations, such as file reading and writing, require information beyond the meta-information. Hence, they are handled by the storage servers where the files are physically stored.
4. Empty file/directory creation at storage: The naming server does not have to redirect all file commands to storage server. Empty file creation is generally the first operation that happens when the client tries to open a non-existent file. In this project we take a simple approach for file creation. The naming server records (in its book-keeping) that the client had requested to open a file and maps the file to the storage server. It then sends a "create" message to the storage server. This ensures that clients do not create new files on the storage servers without the meta-information being filled at the naming server. Similar operations have to be taken care during a delete operation. You have to create a design and an empty implementation that supports delete operation; no need to implement the logic for project 1 (this framework will be necessary for project 2). However, implementation of delete is a bonus-question for project 1, and we highly encourage you to attempt to implement it.

In the reference framework provided to you by the test-code, the above four functionalities can be split into two interfaces at naming: naming.Registration and naming.Service. The naming.Registration interface handles the registration messages from storage servers. The naming.Service interface handles the service calls from clients to (a) get information about storage server, and (2) service file meta-informations. Please refer to the javadoc of the naming package for further details.

Storage Server

Storage servers provide actual storage space for files in the File Stack. The storage server stores the files of File Stack in a temporary storage directory in its local file-system. For simplicity, we call this directory as “storage directory” of a storage server. The storage server has the following functionalities:

1. Registration: As described in the “Storage Server Redirection” in the previous section, the storage server has to first register with the naming server. During registration, the storage server recursively lists the files in its storage directory. It transmits the resulting directory tree to the naming server. After registrations from all stroage server, the naming server, hence, knows the files stored on each storage server.
2. Client service for file information: Once the client obtains the information about the storage server from the naming server, it contacts the storage server requesting for the file operation. The file operations supported by the storage servers are read, write and size (to obtain the size of the file) commands.
3. Commands from naming server: The naming server can issue the command “create” and “delete”. For project 1, you have to only handle “create” command.

The above commands are exposed as storage.Command and storage.Storage interfaces. The storage.Command interface of the storage server listens to the commands from naming (e.g. create and delete). The storage.Storage information allows the client to read, write and query the size of the file. Please refer to the javadoc of storage package for details.

Client

The client is any entity that wants to access files on File Stack. The client can run on any host that is connected to the network. The client should posses generic stubs that allow it to make any call on File system. These stubs are small pieces of code that act as wrappers for providing functionalities that may be actually implemented on other hosts. Please see more about creating stubs and remote calls in the next section.

Communication between entities

As shown in Figure 1, the following communications should be supported: Communication between

1. Storage Server and Naming: This has one functionality, i.e. registration of storage servers. The storage server will initiate this conversation, and the naming server will acknowledge the registration. In the reference framework provided to you in the starter code, the “naming.Registration” interfaces handles this communication.
2. Naming and StorageServer: The naming server may initiate a create or delete operation on storage server. This is serviced through “storage.Command” interface.
3. Client host and naming: The code on the client host (stub) will first interact with naming for any file operation. This is exposed through “naming.Service” interface. This involves commands to

get the storage server information, and file meta-operations.

4. Client host and storage server: After the client obtains an handle for the storage sever on which the file resides, the code on the client host (stub) will contact the storage server to perform the operation. Operations to read/write and querying size are performed through this communication. This functionality is exposed through storage.Storage interface.

While the code on the client host contacts naming and storage server, it is necessary to allow simple APIs for clients for file operations. The clients should transparently perform file operations without the knowledge of how or where the files are stored. You will enable implementing your own Remote Method Invocation (RMI) library.

In the reference framework provided to you, RMI library is implemented in the package *rmi*. This package contains two important classes, Skeleton and Stub, which can be used to implement RMI library. In RMI, a client possesses a *stub object*. The stub object purports to implement a certain functionality. In fact, the functionality is implemented remotely by a server. The stub object merely marshals[1] the arguments given to its methods and transmits them over a network to the server. It then waits for the server to provide the result, which is returned to the client. RMI hides the network communication from the client. Network requests and responses appear to the client as regular method calls on the stub object.

The *skeleton* is an object on the server that is responsible for maintaining network connections and unmarshalling arguments. It is the server's counterpart to the client's stub. The skeleton may itself implement the server functionality directly, or the functionality may be implemented by other objects.

The Skeleton base class includes a multi-threaded server which should communicate with stubs over TCP connections. The Stub base class includes several helper methods useful in writing stubs. Each Stub object, when created, is given the network address of the skeleton with which it is to communicate.

The library is used by deriving a class from Skeleton and another class from Stub. The class derived from Stub will typically implement an interface listing the methods that the server provides. Typically, only this interface need be visible to the client at compile time. The derived classes implement the protocol for communicating arguments and return values.

In the typical case, the server will create both the skeleton and the corresponding stub. The stub will then be transmitted to any clients that wish to use the services provided by the server. Stubs may also be created directly by the client – this is provided primarily to bootstrap RMI. If all stubs were created by the server, then it would be necessary to use a means other than the RMI library to transmit an initial stub to the client. To avoid this, the RMI library allows the client to create an initial stub. In this case, the stub class must be visible to the client at compile time. The initial stub typically connects to a well-known service. That service may be all that the server provides, or it may be a means to obtain more stubs for other services. For more details, please refer to the “rmi” package in the java-docs.

Design and implementation guidelines

The project involves many modules such as RMI, naming server, storage servers and client code. Each component may in turn use the other modules. For example, RMI is used by both naming and storage

1 *Marshalling* the data is a process where you convert the input and output data such that they can be transmitted over the network as bits.

servers to communicate. In this section, we provide the sequence of modules that can be built and their possible design considerations for effective implementation of the project.

We recommend the students to build the modules in the following order:

1. RMI library
2. Storage Server
3. Naming Server
4. File access functionalities

Design of the RMI library

In order to implement the RMI library, you should create stubs and skeletons that talk to each other. We now explain the design of stubs, skeletons and the communication mechanism.

Communication mechanisms

Stubs and skeletons can possibly reside at different host machines, and have to communicate over the network. Java Sockets provide API to send and receive messages from one application to another. Use Java Socket programming for sending and receiving messages over the wire. Get started by reading tutorials, and writing simple socket examples (Search for Java Socket programming tutorials online).

Part 1: Connection test: First write simple code that connects and communicates with a socket on different machine.

Part 2: Simple protocol: Now its time to design your protocol. Remember, you will be using these sockets to send and receive general datatypes (any serializable objects) from stub to skeleton. At the stub, you should be able to send object, and you should receive objects at the skeleton. The skeleton might send any object (return values, exceptions, etc.) to the stub. Hence, design a simple basic socket server and client that accepts a connection from client, reads some object and sends back another object or exception. You should handle all exceptions ranging from connection exceptions (such as not able to connect to server).

Part 3: Remote invocation of methods: You will be using RMI for all types of communication (from client to naming, client to storage server, storage to naming server, etc). Hence, it is important to design a generalized RMI that can be used across all RMI communication. A generic way is to provide a “object”, a “method” and the “arguments” input to the skeleton, and the skeleton will search for the object and call the method on the object and return the results (Hint: use Java Reflection). As a starting code, it is recommended to write a sample code where: (1) the client passes an object, a method and the arguments to the server, and (2) the server executes the method and sends back the return values.

Generic stub-skeleton framework

Think back about the whole problem. We now know how to invoke a remote method, but we still have not created stubs and skeletons. So, how can you design stubs and skeletons? Ofcourse, you can write custom stubs and skeletons for each RMI call. But we don't want such custom stubs and skeletons; this will lead to hundreds of stubs and skeletons – one for each function call. We want to design RMI framework in a generic way that allows any module (naming, storage, client) to communicate with any other module. Think about the communication mechanism before designing generic skeleton and a stub.

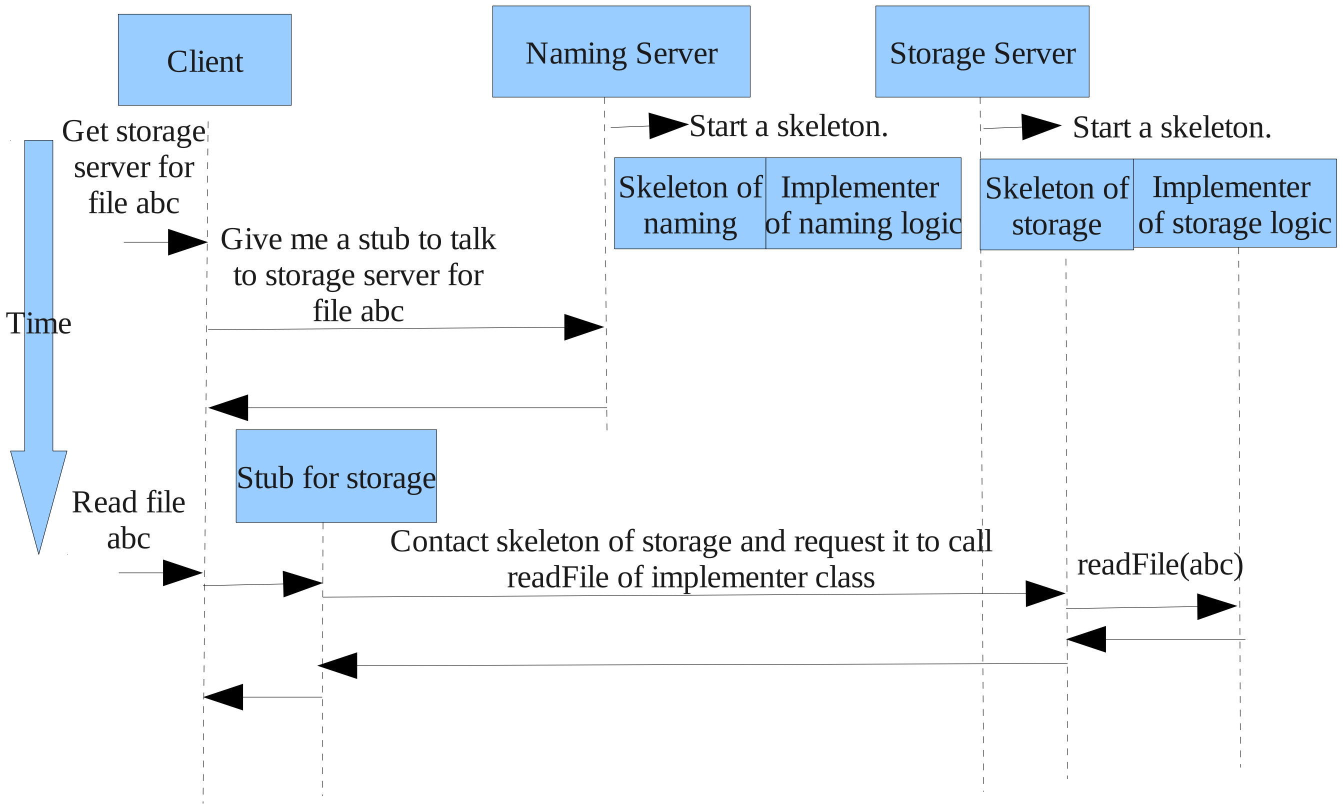
Client
Naming Server
Storage Server
Start a skeleton.
Start a skeleton.
Skeleton of naming
Implementer of naming logic
Skeleton of storage
Implementer of storage logic
Get storage server for file abc
Give me a stub to talk to storage server for file abc
Time
Stub for storage
Read file abc
Contact skeleton of storage and request it to call readFile of implementer class
readFile(abc)

Figure 2: Example of client contacting naming server to obtain stub for storage server

Consider an example of how client obtains a stub for naming server (a first step) as shown in Figure 2. The client – in order to execute any functions on a distributed file – has to first contact the naming server. Naming server will provide the information of the storage server (IP address and port of storage server skeleton) where the file will be served. In order to contact the naming server, the client uses an RMI and calls a function, say, getStorageServerInfo(abc) to get the storage server where file abc resides. However, it cannot *connect* to the naming server by a simple function call. It needs a stub for naming server that connects to the skeleton of the naming server. How does it obtain the stub?

An example of how to obtain stubs is shown in Figure 2. Each server (naming and storage server) will create its own “skeleton” object. This object is generic in the sense that it is only capable of handling one type of request: it can read method name and arguments, and call the method in an “implementer” object. The logic of what to do is inside an “implementer” object that is passed to the skeleton as a parameter. Skeleton just unmarshalls the implementer method and arguments, calls the method, and return the values to the client. The skeleton is also capable of creating stubs and passing the stubs to the requesting client. For example, the skeleton on the naming server can create stubs for the next redirection entity – the storage server for a file – and send it to the client. Using this storage server stub, the client can now contact the storage server using RMI.

Simple? But this is a chicken and an egg problem: you need to contact the skeleton to get the stub of the entity, but we cannot contact the skeleton without a stub! Hence, you need some bootstrapping entity what will issue stubs for first-level entities. And this is... you find it out. Once you get this information, you can contact the server and ask for stubs.

Once you have the RMI module designed, implemented and tested thoroughly, other modules (storage, naming and clients) are easy to implement. Spend most of your time getting this part correct.

Frequently asked questions:

1. All the generic functions take in some form of arguments (method objects, primitive data types). Are you going to test for trivial parameters (such as null)?

   Ofcourse, yes. Not only for RMI module, but for all modules. Take a look at the test-suite.

2. Many sockets and files are opened. Do I have to cleanup after their use?

   Ofcourse, yes.

3. Can I have custom skeleton and stubs? Can I have the client instantiate a stub? Can I avoid getting the stub from the skeleton?

   No. We want you to design nice and elegant distributed systems.

4. Do we pass exceptions from skeleton to stub?

   Yes. In addition to return values, you should also be able to pass exceptions. This gives client a capability to understand what happened, and correct if necessary

5. Each skeleton is tied to an implementation object. What if the implementation object is faulty (forgot to handle exceptions, etc...)? Should the Skeleton crash?

   Skeleton should be robust enough to handle these failures. It should not crash.

6. What if the implementer object of the skeleton does not throw remote exception?

   Check and make sure that the relevant methods in implementer object throws RemoteException. How do you check? Java reflection.

7. What if we pass a method to the skeleton that is not implemented by the implementation object?

   Make sure you check. Throw an user-defined exception to client if the call is invalid.

8. What happens if a stub from client-1 calls the skeleton that takes a long time? Should other clients wait?

   Make the skeleton as a multi-threaded server.

9. When multiple clients are accessing the same variable, how do we make sure that the access to variables are synchronized?

   Read Java synchronization

10. How do you create a generic skeleton that can take in any generic implementor class?

    Search for Java Proxy and Java InvocationHandlers (e.g., http://tutorials.jenkov.com/java-reflection/dynamic-proxies.html). Read thoroughly. Spend some time thinking on such a design. Discuss with instructors and TAs before you start concrete implementation.

Design of Naming Server

Both storage and client will first contact the naming server to request. We recommend opening two server sockets at the naming: one for facilitating registration for storage servers, and one to deal with file-requests from clients. We describe the registration process in under the next subsection “Design of Storage Server”. We now explain how clients may query naming server.

1. The clients can ask for storage server information (storage stubs) to access certain files
2. The clients can ask for meta-statistics: list files/dirs, and create empty dir/file. Naming keeps a

record of which files are present (in which storage servers). Hence, it can easily answer meta-data related functionalities.

Design of Storage Server

File storage: Storage Server physically stores the distributed files in FileStack. Every storage server can be started with a base-directory for FileStack. Any file under this directory is a part of the distributed file system in FileStack. Clients request the file using absolute FileStack path. The storage sever has to resolve this absolute path into its local path. For example, let us assume that the base-directory for Storage-Server-1 is /home/dist-sys/fileStack/store/. If the client wants to create a file "/dir1/subdir/file.txt", then the storage server has to resolve it to /home/dist-sys/fileStack/store/dir1/subdir/file.txt

Registration: Every time a storage server starts, it first registers to the naming server. It communicates its base-directory and current set of files to the naming server. The naming server will update its book-keeping instructions about the availability of the file. During the registration process, naming server may dictate the storage server to delete some files. For example, if the naming sever finds that /file1.txt is at storage servers SS1 and SS2, then it may tell SS2 to delete file1.txt during registration. Note that all registration process between storage server and naming server should use RMI.

Command interface and Storage interface: Storage server may receive two type of requests:

1. Commands from naming server (e.g., create files) from naming server.
2. Requests for file read/write from clients.

Storage server should expose two interfaces (and, generally, two skeletons): the command interface handles the former requests, and the storage interface handles the latter.

Starter code

Please use the code in P1StarterCode.zip

Bonus question (10% bonus)

Implement the delete operation of files and directories in FileStack.

Test code

We have also provided test code. The test cases test if your code is conforming to the above design guidelines, and to check if the implementation is correct. Please note that this is a service offered to help you design and test faster. You are solely responsible to make sure that your code works perfectly. During grading, we will also use other test cases to make sure that your project is working as expected.

Deliverables

A zip containing the source code. Please adhere to the same package and directory structure as that 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 instructors know. You need a written approval from the instructor before modifying the framework. You are however free to add files within the existing packages.

Handing in your solution

Submit your code on AFS directory /afs/qatar.cmu.edu/msakr/15440-f11/handin/*userid*/p1/, where *userid* is your andrew user id.

Late policy

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

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