15-440: Distributed Systems
Project 1: File Stack
Implementation Notes

Overview

The servers are written as libraries. They have no main methods on their own. Instead, separate applications that drive the servers are provided in the apps package.

All the code that needs to be implemented is located in the rmi, common, storage, and naming directories.

The common package contains the path library. This is probably the easiest part of the project to implement, and can be implemented first, especially if one needs a re-introduction to Java. After that it is necessary to implement the RMI library, because the storage and naming servers depend on it - although it is possible to implement the storage and naming servers directly using sockets without RMI. It is probably easier to implement most parts of the storage server before doing substantial work on the naming server.

The servers and RMI skeletons are implemented according to a common pattern: there are several constructors and public start and stop methods that are used to drive the servers. The servers also provide several protected methods, among them stopped, which should be overridden by a user such as an application to handle server events.

When starting to write code, it may be a good idea to comment out all test cases in conformance.ConformanceTests.java that are not testing what you are currently working on (or have already finished). This way, you can code the path library without worrying about all the other test cases. Then, when writing the RMI library, you may want to write simple unit tests to check that things are working as you go - before you are ready to subject the RMI library to the conformance tests. If you are not comfortable with unit tests, you can work on the RMI package in a separate project directory. However, a sample unit test is provided so that you can see how easy unit tests are to use. To add a unit test, create the appropriate class file (under, say, unit/rmi), and then add the class to the list of tests to run in unit/UnitTests.java.

As distributed, the all the filesystem components will be put into one large JAR file called "dfs.jar", if the user types "make jar". This is only one way to package the filesystem. The code is arranged in such a way that it is possible, for instance, to create a JAR file individually for each server, or for each application, or for the client library. For example, to make a JAR file for only the naming server, you would include the packages naming, rmi, and common, the Storage and Command classes from storage, and use NamingServerApp as the JAR entry point.

The interfaces provided with the starter code strongly suggest a certain design. It is not the intent, however, to force you to implement the project in a certain, single way. The client interfaces (naming.Service and storage.Storage) are fixed. However, the inter-server interfaces can be negotiable. Feel free to modify the filesystem design if you believe your modification is an improvement. Changing the inter-server interfaces will break a large number of test cases, so please consult the staff

regarding your modification. Likewise, some of the test cases make certain simplifying assumptions about how the servers operate. If you believe your design is an improvement over the design implied by the test cases, and this is the reason the test case is being failed, please inform the staff, and your modification will be graded on its merit and not on whether it passed or failed the test case. In special cases, if you present a coherent alternative design, the client interfaces may also be modified - but this will break the client utilities.

RMI Library

If your RMI library uses the classes ObjectInputStream and ObjectOutputStream to transmit objects over the network, be sure to initialize them in the proper order. The ObjectOutputStream must be constructed and flushed before the ObjectInputStream is constructed. This is because the ObjectInputStream constructor blocks until a stream header is received from the peer's ObjectOutputStream. Therefore, if you constructor two ObjectInputStreams first on both peers, both constructors will block waiting for stream headers, and the two peers will deadlock. The same can happen if you do not flush the ObjectOutputStream immediately after constructing it. This flush will force the stream header to be sent immediately.

Servers

Take care when using the read and write methods from Java's standard stream objects. The offset arguments to those methods are not offsets into the stream being used - they are offset into the array being passed by the caller (you).

If using the File.delete method to perform directory deletion on the storage server, be sure to make the deletion recursive. The File.delete method will refuse to delete a non-empty directory. The storage server, however, should be able to delete non-empty directories.

The File.mkdirs method returns false if it is not asked to create any directories. Depending on how you implemented directory creation on the storage server, overlooking this may cause your code to think an error has occurred when in fact there is no error.

Server code can be executed by more than one thread at a time. If this is not desirable in places, you should use synchronized methods, synchronized statements, or some other strategy of ensuring mutual exclusion. Your code will be read for race conditions.

If multiple storage servers are connected to a naming server, you should use some strategy for deciding which server should host new files. If your strategy uses numbers, you should be aware that Java's random number generator (class Random) is not guaranteed to be thread-safe.

The storage server uses a local directory as its root directory. This local directory may be specified as a relative path. In this case, if the storage server does not take care to convert the path to an absolute path, the process working directory may suddenly change later, and the storage server will suddenly be serving files from the wrong directory.

Be very careful with where you start storage servers. Storage servers may delete files from the

directories in which they are running, so these directories should not contain important files that you have not backed up, especially when you are still developing the filesystem. Do not start a storage server in the same directory as another storage server is currently using - this will probably cause all files in the directory to be deleted during server registration.

General coding

When writing thread classes, do not mark them as extending Thread. Instead, mark them as implementing Runnable, and then create threads from them. See the documentation of the class Thread.

Check the validity of arguments early, if possible. For example, it is wrong to modify a large amount of state in, say, the naming server register method, only to later discover that one of the arguments you have not used so far is null, and the method should throw NullPointerException. This argument should be checked before the state is modified.

If state is modified by a method, but the method later fails, an effort should be made to restore the state. This is not always easy. For example, this may be difficult in the delete method of the naming server. Where it is difficult, it is not necessary to do it - but do write a comment explaining why it is difficult.

Objects holding system resources should have the finalize method overridden to make sure those resources are freed when the garbage collector cleans up the object, in case the user forgets to free them manually. Consider adding a cleanup task using the method Runtime.addShutdownHook for especially critical system resources.

The filesystem code is highly multithreaded. Be sure you are comfortable with stopping threads gracefully on command. Do not use the Thread.stop method.