Project 2: Distributed Filesystem
15-440 Spring 2012

Assigned: Thursday, 16 February
Due: Thursday, 8 March

Overview

In this project, you will implement a simple distributed filesystem using the RMI library you created in project 1. Files will be hosted remotely on one or more *storage servers*. Separately, a single *naming server* will index the files, indicating which one is stored where. When a client wishes to access a file, it first contacts the naming server to obtain a stub for the storage server hosting it. After that, it communicates directly with the storage server to complete the operation.

When completed, your filesystem will support file reading, writing, creation, deletion, and size queries. It will also support certain directory operations — listing, creation and deletion. It will be possible to lock files, and commonly accessed files will be replicated on multiple storage servers.

Logistics

You should work with a partner on this project. The starter code is found in the Labs section of the course Web page. After you have unpacked it, place your RMI library in the `rmi/` subdirectory. The code will not compiled until you do so. When you are finished, submit a single zip archive to:

`/afs/andrew/course/15/440-s12/handin/lab2/yourandrewid`.

where `yourandrewid` is the Andrew ID of one of the partners in the group. Mention the names and IDs of both partners in a comment at the top of every Java file that you modify, and add them to the `README`. Be sure to include all files necessary to compile and run the filesystem, including an updated version of your RMI library, divided between the packages `rmi`, `common`, `storage`, and `naming`. Your archive should contain a subdirectory for each of these packages. We will extract your classes from these packages and use them for testing. Your filesystem should work on Java 6 virtual machines.

Checkpoint

This project involves a large amount of work, and many features must be implemented before it is complete. To help you with this, we suggest an optional checkpoint, for which you should implement all file and directory operations except locking and replication. The starter code provides you with two sets of conformance tests. The default set includes tests only up to the optional checkpoint: none of the tests depend on locking or replication. Once you are ready to proceed on to the full version of the filesystem, simply replace the default `conformance/` subdirectory with the `conformance/` subdirectory distributed inside `conformance-final.zip` in the starter code archive. The checkpoint is not part of your grade, nor is there a deadline for it. It's simply a convenient intermediate goal for you to work towads.

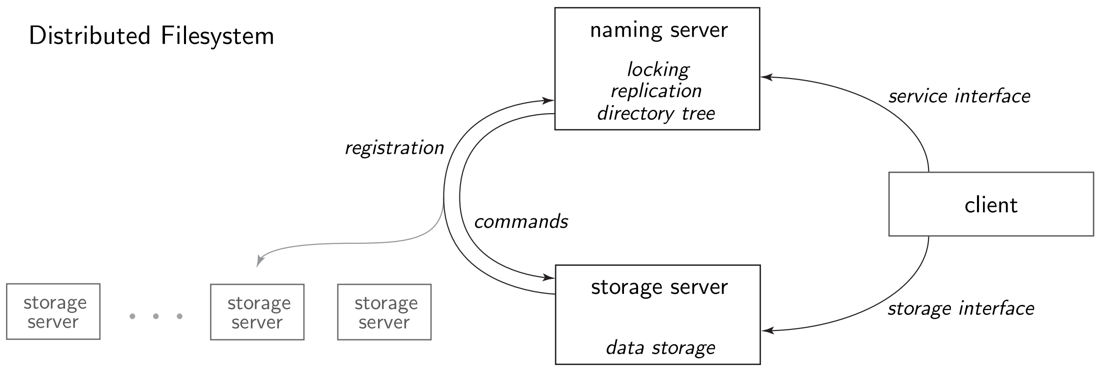
Detailed Description

The filesystem consists of several major components. The first of these is the RMI library, which you are already familiar with. All communication between servers and clients, and between naming and storage servers in the filesystem, is done using the RMI library. The RMI library is in the package `rmi`.

Next, servers and clients need a way to identify files. Each file is identified by its path in the distributed filesystem. Paths are represented by `Path` objects, which are defined in the package `common`. These path objects are transmitted through the various interfaces in the filesystem, and you may use them internally in your data structures, if you so choose.

The primary function of *storage servers* is to provide clients with access to file data. Clients access storage servers in order to read and write files. Since storage servers store the data, they are also, in this design, the entities that report file sizes. Storage servers also must respond to certain commands from the naming server.

Clients do not normally have direct access to storage servers. Instead, their view of the filesystem is defined by the filesystem's single *naming server*, for which clients have an RMI stub. The naming server tracks the filesystem directory tree, and associates each file in the filesystem to a storage server. When a client wishes to perform an operation on a file, it first contacts the naming server to obtain a stub for the storage server hosting the file, and then performs the operation through this stub. Naming servers also provide a way for storage servers to register their presence.



Paths

As stated above, paths are represented by `Path` objects, which are transmitted through all interfaces in the filesystem. Notionally, each path is a string of the form `/directory/directory/directory-or-file`, but you may choose any internal representation you want — for example, you may represent a path internally as an array of path components. Paths are always taken to be relative to the root of the filesystem. Because paths are transmitted over the network, path objects must be serializable. Finally, the filesystem locking scheme requires paths to be comparable. See the section on locking for details.

Storage Servers

Storage servers provide two interfaces: the *client interface*, through which clients perform file operations, and the *command interface*, through which the naming server may issue file management commands to the storage server.

The client interface is specified by the interface `storage.Storage`. It provides the client with three operations: file reading, file writing, and file size query. The command interface is specified by `storage.Command`. It allows the naming server to request that a file on the storage server be created, deleted, or copied from another storage server, as part of replication. The job of the storage server is simply to respond to these requests. These requests may come in concurrently.

A question arises — where does the storage server *actually* store the data for the files it is hosting? In this design, the storage server is required to put all its files in a directory on the machine that it is running on — this will be referred to as the storage server's *local* or *underlying* filesystem. The structure within this directory should match the storage server's view of the structure of the entire distributed filesystem. For example, if a storage server is storing its files in the local directory `/var/storage`, and it is hosting a file whose *distributed* filesystem path is `/directory/README.txt`, then that file's full path on the local filesystem should be `/var/storage/directory/README.txt`. If a storage server is not aware of the existence of a file in the filesystem (because it is hosted by another storage server), it need not store anything for the file. This scheme provides a convenient way to make data persist across storage server restarts.

Naming Server

The naming server can be thought of as an object containing a data structure which represents the current state of the filesystem directory tree, and providing several operations on it. The primary interface that the naming server is required to implement is the *service interface*, specified by `naming.Service`. The naming server also provides a *registration interface*, specified by `naming.Registration`, which storage servers use to inform the naming server of their presence and join the filesystem.

For the optional checkpoint, the service interface allows a client to create, list, and delete directories, create and delete files, determine whether a path refers to a directory or a file (or neither), and obtain stubs for storage servers. For the final version, the service interface also allows clients to lock and unlock files and directories.

The registration interface is used once by each storage server on startup. When a storage server is started, it contacts the naming server and provides the naming server with two stubs: one for the storage server's client interface, which the naming server will later provide to clients, and one for the storage server's command interface, which the naming server will use to maintain the storage server's view of the filesystem in a consistent state. The storage server also lists all files that are present in its directory on its underlying filesystem. If any of those files are not yet listed in the distributed filesystem, they are added. The rest are considered duplicates, and the naming server will request that the storage server delete them.

In the final version of your filesystem, the naming server transparently performs replication of commonly accessed files, causing multiple storage servers to maintain copies of the same file. This is not under the direct control of the client. The details are given in the section on replication.

Coherence and Thread Safety

Each server in the filesystem must individually be thread-safe: attempting to perform operations concurrently on a single server should never cause that server's state to become inconsistent. The consistency requirements across the whole filesystem, however, are much more relaxed. The design is fairly fragile and depends strongly on having well-behaved clients. For the version up to the optional checkpoint (and therefore without locking), consistency also depends on luck.

At the optional checkpoint, without locking, clients maintain no special state over a file that would allow them to consider the file "open" or reserved for their own purposes. This means that, while a single read or write request should complete correctly once it arrives at the storage server, any other client may interfere between requests. Files currently being accessed by a client may be overwritten by another client, deleted, moved to another storage server, and re-created, all without the client noticing. The final version of the project allows a client to lock a file in order to prevent other well-behaved clients from performing any of these operations until the lock is released.

When implementing the storage and naming servers, you must decide when is the appropriate time for the naming server to command each storage server to create or delete files or directories, thus maintaining the servers' views of the filesystem in a consistent state. As much as possible, it is preferable to avoid having to rigidly and synchronously maintain all storage servers in the same state. However, the interfaces are highly simplified and do not provide good ways to implement complex schemes for lazy file creation or deletion, so code accordingly. As an example, a file that the naming server has been successfully asked to delete should not remain accessible for subsequent requests to the storage server.

We ask that files that have been deleted from the storage server be deleted from the underlying filesystem (as opposed to merely unlinked from some internal data structure), and that the storage server eagerly remove underlying filesystem directories that have become empty. This allows us to test the behavior of your storage server in response to requests.

Locking

For the final version, you must implement a custom lock type and a particular locking scheme, which well-behaved clients can use to ensure consistency across multiple requests. Each file and directory may be locked for *shared* (reading) or *exclusive* (writing) access. Multiple clients may lock the same object (file or directory) for shared access at the same time, but when a client locks an object for exclusive access, no other client can lock the same object for any kind of access.

Shared access permits multiple well-behaved clients to perform operations such as reading files and listing directories simultaneously. Such operations do not interfere with each other: for example, two clients may saefely read the same file at the same time. Exclusive access permits well-behaved clients to write to files and modify the directory tree.

When a client requests that any object be locked for any kind of access, all objects along the path to that object, including the root directory, must be locked for shared access. Otherwise, for example, a file that is locked for reading (shared access) can still be deleted when another client removes its parent directory, because the parent directory was not locked by the reading client. Be careful about the order in which you take these locks on parent directories. If locks are taken in haphazard order, it is possible to end up with a deadlock where two clients are each holding a lock, and both need to also take the lock held by the other client in order to proceed.

The locking scheme has a further constraint. Some clients may need to lock multiple objects simultaneously. Doing this in arbitrary order on each client can also result in deadlock for the same reason. Therefore, the path library (`common.Path`) requires that path objects be comparable. Clients must take locks on path objects in order from least path to greatest path, according to the results of comparison. This requirement interacts with the requirement to lock subdirectories: when an object is locked, it is not only the object itself whose lock is taken, but the lock on every object along the path to it. Great care must be taken that the order defined by the comparison does not lead to deadlocks due to this interaction. Be very careful about how you compare path objects. We ask that you describe your locking scheme and comparison scheme, and their interaction, in comments by `common.Path.compareTo` and `naming.Service.lock`.

In addition to all of the above, the locks must also provide some measure of fairness. It should not be the case that a client which is waiting for a lock is continuously denied it, and the lock is given to other clients that requested the lock later. This is especially important for clients requesting exclusive access. In the absence of fairness constraints, a large number of readers will make it impossible for any client to write to a file. The writing client will wait for the lock as new readers keep arriving and sharing the lock with current readers.

In order to avoid this, in this project, we require that you give the lock to clients on a first-come, first-serve basis. It must never be the case that a client which requests a lock later is granted access before a client which requested earlier, unless both clients are requesting the lock for shared access. In the latter case, however, if two clients are requesting the lock for shared access, and there are no intervening waiting clients, both clients must be able to take the lock at the same time.

Replication

The final version of the project must support replication according to the following simple policy: during a series of read requests, the file is replicated once for every 20 read requests, provided there are enough storage servers connected to the naming server to maintain additional copies. At a write request, the naming server selects one storage server to keep a copy of the file, and all other copies are invalidated (removed) before the remaining copy is updated.

Since the naming server has no way of directly tracking read and write requests, or the amount of traffic associated with each file, it makes the simplifying assumption that taking a shared lock on a file is tantamount to a read request, and taking an exclusive lock is a write request.

Be careful about how replication interacts with locking. Well-behaved clients should not be able to interfere with the replication (copy) operation and cause the results to become inconsistent. Well-behaved clients should, however, be able to read from existing copies of a file, even as a new copy is being created.

Implementation

Create three classes, `common.Path`, `storage.StorageSever`, and `naming.NamingServer`. Incomplete versions of these classes are provided in the starter code, with documentation comments by each method giving the details of how the method must behave.

The interfaces `storage.Storage`, `storage.Command`, `naming.Service`, and `naming.Registration` are provided for you. Some of these interfaces have extensive comments describing method semantics, which you should, of course, read. To generate browsable documentation, run `make docs`. The documentation will be available in the `javadoc/` subdirectory.

The interface `naming.Service` declares the methods `lock` and `unlock`. If you wish to code up to the optional checkpoint, either comment out these method declarations, or create empty implementations for the methods in `naming.NamingServer`.

There are helper classes provided with the starter code, such as `naming.NamingStubs`. You may create whatever other additional helper classes your implementation requires.

Copy your implementation of the RMI library into the `rmi/` subdirectory. Since this project is done in groups of two, you may use either partner's implementation. You may also splice these together, or otherwise improve your RMI library as you see fit.

As mentioned above, the conformance tests distributed in `conformance/` test only up to the checkpoint. If you wish to run all tests, use the replacement `conformance/` directory from `conformance-final.zip`, which is contained within the starter code archive.

Using

Three methods of using the filesystem are provided together with the starter code. Programmatic access is possible through the `client` package. This package provies two classes, `DFSInputStream` and `DFSOutputStream`, which extend `java.io.InputStream` and `java.io.OutputStream`, respectively. This allows you to create streams connected to files in the filesystem and treat them as any other Java I/O streams.

You may browse your filesystem from the command line using the `dfs` script in the project main directory. This script, together with the `apps` package, allows you to issue commands similar to `ls`, `cd`, `mkdir`, and `cp` (actually, `put` and `get`). The usage is described in detail in the file `README.apps`. The script requres a bash shell to work correctly.

Finally, if you have a Linux or Mac OS system, you may compile a FUSE driver and mount your filesystem directly. After this, it can be browsed from the command line, in graphical file browsers, and accessed programmatically in the usual fashion. The FUSE driver is located in the `fuse/` subdirectory. You will need to modify one variable in its `Makefile` which depends on your implementation. See details in the `BUIDLING` section of `fuse/README`.

Notes and Tips

Perhaps the easiest class to implement is `common.Path`, so you may want to start here if you want to get going quickly.

Both the naming server and the storage servers will run two skeletons each, one for each of the two interfaces each kind of server implements. The naming and storage servers must also stop gracefully in response to calls to `stop`. In order to ensure this, you will have to subclass `Skeleton` and override the `stopped` method to provide the servers with notification that their skeletons have, indeed, stopped. Just like the skeletons, both the storage and naming server have certain protected methods. You do not need to modify these methods. These are event notifications that you must call in response to the appropriate events. They will be overridden when someone subclasses your servers.

If you write an application that manually starts storage servers, or use the `dfs` script to do this, be careful about what directory you start them in. Storage servers register with a naming server on startup. Every file in the storage server's directory on the underlying filesystem which is already present in the distributed filesystem will then be deleted to eliminate duplicates. This means if you start a storage server in a directory, a large number of files could suddenly disappear. Make sure this is not an important directory. The test cases only start storage servers in temporary directories.

The servers must be thread-safe. This means, most likely, that at least some methods or code blocks will be marked `synchronized`. However, when implementing locks for the final version of your filesystem, you must not simply make every method of the naming server synchronized, and then take locks. This will decrease concurrency. Instead, you are expected to rely on the per-object locks to ensure thread safety as much as possible. Two clients should be able to traverse the same directories simultaneously within the naming server, provided they are locking the objects along the way for shared access.

When you were implementing the skeleton in project 1, you were not required to make any effort to stop service threads during shutdown. In the naming server, however, a large number of service threads may be queued up waiting for a lock. Consider if you can somehow interrupt the lock, so that all threads are refused access immediately, and the server can shut down in a more graceful fashion.

The `storage.Command.copy` method, which is used for replication, may be used to replicate very large files. Such files cannot be stored in the virtual machine's memory all at once. You must implement this method in such a way that files substantially larger than the virtual machine heap size can still be replicated.

The storage server constructors are passed a local filesystem directory path to use for storage. This path is not guaranteed to be absolute, so it may depend on the current working directory of the application which started the server. It is a good idea to convert this to an absolute path before using it.

The section on replication is very short, because replication is easy to specify. It is, however, not trivial to implement. Replication is a potentially long-running process that runs concurrently with many other processes, including accesses to existing copies of the file being replicated. Allow yourself adequate time to think through, design, and test the replication mechanism.

The storage servers can run their interfaces on any port, because storage server stubs are always provided either by the storage server itself (upon registration), or by the naming server (upon request by a client). Stubs for the naming server have to be created explicitly, however. The class `naming.NamingStubs` provides convenience methods for doing so, and defines the ports on which the naming server should make its interfaces available.

The class `common.Path` includes a method which returns an iterator. The purpose of this is to allow you to iterate over path components using Java's for-each syntax.

You may write your own unit and conformance tests. The testing library is the same as in project 1 and all the same directions apply — the only difference is that you are given more tests for project 2.

Be careful about the semantics of standard Java methods. The documentation is often terse and assumes a certain understanding (or willingness to test), and may act in somewhat unexpected ways in corner cases. For example, `File.mkdirs`, which might be useful for your storage server to manage the underlying filesystem, will return `false` if the directory it is asked to create already exists. This could lead to an annoying corner case if you always blindly call this method, taking it to mean "ensure that the directory exists."

If you use random number generators for deciding which server should store a new file, or for picking servers during replication or read accesses, be aware that Java's standard random number generator is not thread-safe.

As always, release all system resources eagerly. Maintain consistent state whenever possible, including in response to typical error conditions and exceptions. When raising an exception, include the cause, if available, and write a descriptive error message. Use good and consistent coding style. You may use a different coding style from the starting code, but please keep it readable and consistent in all the files you modify or produce. Stick to 80 character per line and four-space tabs (or better, replace tabs by spaces). Comment your code well.

References

File management (`File` object is really more like a path)
`http://docs.oracle.com/javase/6/docs/api/java/io/File.html`

Java synchronization
`http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html`