15-440: PROJECT 2
SYNCHRONIZATION AND REPLICATION IN DISTRIBUTED FILE SYSTEM

IMPLEMENTATION NOTES

SUMMARY OF CHANGES WITH RESPECT TO PROJECT 1

The naming server supports file and directory locking. The Service interface therefore has two new methods: lock and unlock. Each node in the directory tree on the naming server now has a read-write lock. Locks for a path are always taken in order from root to final node. Threads requesting the lock are granted it in first-come first-serve order, with the exception that threads requesting shared access are granted the lock at the same time (if there is a block of threads in the queue all requesting shared access, then when one of them gets shared access, they all do, until the next thread requesting exclusive access, or until the end of the queue).

The naming and storage servers support replication. Read and write accesses are noted on the naming server when file lock requests occur. The storage server Command interface provides the new copy method to support replication.

Path objects now implement Comparable. This is meant to allow an application to choose a locking order if multiple locks need to be taken. All applications must take locks in the order defined by Path.compareTo. See the big comment by that method for further explanation.

StorageServer now has an additional constructor StorageServer(File, int, int), which takes two port numbers to force the client/storage and command interfaces to use the given ports.

LOCKING

Naming server methods should not all be synchronized anymore. The read-write locks now provide most of the mutual exclusion needed. There are a few exceptions, however. A student that leaves all methods synchronized is not relying on the read-write locks.

Locking must be done carefully. For example, it is not acceptable to traverse a path and get a list of directory tree objects along that path, and then lock each one. This is because if the parent of an object remains unlocked, another thread can unlink the object from the tree after the path is traversed, but before the object is locked. Then, the locking thread has locked an object that no longer exists. When locking, it is necessary to lock an object, then consult its child list, and then attempt to lock the next object. Locking an object should prevent its child list from being modified (at least by well-behaved clients).

It is also important to unlock all objects that have been locked when locking fails. For example, if three components are locked before it is discovered that the fourth does not exist, then those three components must all be unlocked.

It is desirable, but not necessary, to make the locks interruptible. This is because when the naming server is shut down, a large number of service threads may still be pooled in lock queues. It is better to stop these threads as soon as possible, instead of permitting them to take the locks and continue performing operations on naming server data structures.

External control of locks is deliberate. This allows an external tool to take a lock, perform several operations atomically, and then release it. Without explicit external locking, complex atomic operations cannot be performed by a client.

Attempting to take the same lock twice for reading can result in deadlock if another client tries to take the lock for exclusive access in between the two attempts.

Taking a lock for shared access on a directory ensures that its child list will not be altered, but does not ensure that the children will not be changed. A subdirectory's child list can be altered, and a file can be written. Taking a lock for exclusive access on a directory ensures that neither it nor the entire subdirectories under it are being accessed in any way by any client.

All the above statements concerning locking assume that it has been implemented correctly, and that all clients are well-behaved. It is the implementor's responsibility to ensure this. Operations that modify a file or directory should lock that file or directory for exclusive access. Operations that work on an entire directory tree at once should lock the root for exclusive access. Operations that read the state or contents of one file or directory should take the lock on that object for shared access.

REPLICATION

Since the naming server has no good way to measure actual read and write requests on the storage servers, it considers a lock for shared access to be a read, and a lock for exclusive access to be a write.

During replication, the naming server should allow threads other than the current reader to read existing copies of the file. It is acceptable to make the reader that caused the replication wait for the replication to complete. However, the gold standard in this is to make replication asynchronous with all readers. Caution must be used if this is attempted however - if the asynchronous replication thread has to take the lock on the object for shared access, it is important that it will not go to the end of the lock queue when attempting to do this, but take the lock together with the current thread. It would be a mistake if the replication thread went into the queue after an exclusive access thread, which will invalidate the copy it has not yet created.

The students should be careful about race conditions related to replication. A second reader should not be able to access a copy of a file that has not yet been completely downloaded by the server making the copy. On the other hand, if this other reader causes another replication, then it should not go to the same server that is still currently downloading a copy.

Threads reading the same file may still access it concurrently, even though there are locks, because these locks allow shared access. It may still be necessary to make a few synchronized statements when these threads access shared server or per-file data structures. Depending on the design, this may be especially important in the code for replication.

The storage server copy method should support large files - up to 2^31 bytes in size (file sizes in the file system are reported as longs). However, it is not practical to copy that many bytes at a time, in great part because the JVM cannot support an array whose size is not an int, or even one whose size is just very large. Therefore, the copy method should download one block at a time, where a block can be 1MB or some other size chosen by the implementor.