Project: Dropbox - Part 2

CS131: Fundamentals of Computer Systems

Thu., March 14, 2013

1 Introduction

The second part of this project will involve including some features of your synchronizer that will tie back to the lectures on virtualization and performance. You will be making your server multithreaded, allowing it to open connections with multiple clients simultaneously. You will also have to create some sort of internal structure on the server to make sure files are modified by one thread at a time. This process will also be used to store hashes of files, which will be used to check for updates to files. To improve performance, you will only send over files that have been changed. We will not be dealing with conflicts (more on this later).

2 Specification

2.1 Multithreading

Your server should now be able to handle multiple threads concurrently, which means that your server will spawn a new thread for each client it is handling. There is a fixed limit on the number of threads your server can spawn (this should be a constant defined somewhere), which can be implemented easily using a thread pool.

2.2 Concurrency concerns

Now that you can have clients concurrently communicating with the server, you need to make sure that they do not mangle their results when syncing at the same time. To do this, you can use either the `synchronized` keyword or a mutex. If you are using objects to represent each file on the server, then using the `synchronized` keyword is an easy way to ensure that only one server thread is accessing an object at a time. If you are not using objects to represent each file, then you will need to use mutexes or some other kind of locking mechanism to ensure that a file never contains half of one client's data and half of another client's data.

2.3 Incremental updates

Your client and server should only send each other files when necessary. You can use the modification time or the hash of a file (or a combination of the two) to tell whether it has been modified or not, either way will be accepted as valid, regardless of the actual differences between using the two (for instance, the modified time may not always mean what you think it means, and there is always the possibility that the hash of a file will not change even if its contents change – but we are ignoring these). You cannot use the last modified time of a directory to tell whether the contents of any of the files in the directory have changed, though, since this time is modified when the directory itself is modified (a file is added or removed to the directory's list of entries). This feature is intertwined with the next, which is two-way synchronization, like real Dropbox.

2.4 Two-way synchronization

Your client and server should now have two way synchronization. This goes hand-in-hand with incremental updating. Your guidelines for synchronizing should be as follows:

- If a file is present in the client and server and...
  - has been updated on neither the client nor the server, do nothing.
  - has been updated on one of the client and server, synchronize the updated file so that the client and server have the updated version.
  - has been updated on both the client and the server, leave the client and server as is and print out a message on the client saying that there is a conflict.
- If a file is created in one directory, add it to the other.
- If a file is removed from one directory, remove it from the other.
  - If a file that is marked as a conflict is removed from the client, it should not affect the server.
  - If a file is removed from the server and is marked as a conflict on the client, it should not affect the client.
  - If a file is removed from the client and is marked as a conflict, then the client should then receive the server's version of the file, now putting the client back into sync with the server.

You do not need to...

- worry about empty directories. It is acceptable for there to be extra empty directories on your server (if a file is removed from a directory) or for there to be a lack of empty directories (if an empty directory is created). The synchronization rules above apply to files. The directory structure should be maintained, but the focus is on the files.
- incrementally update the files themselves. You should just send the whole file over if you have detected a change, rather than trying to send a piece of the file over.
- make the client or server crash-resistant.

Note: While handling the edge cases for synchronization is important, the emphasis will be on keeping a design that is clean and sustainable for future work (there are two more parts left, after all).

3 README and Commenting

You will have to update your README for this part of the project, again documenting any high-level design decisions made. The README should contain any high-level documentation you may have, and the rest of your documentation should be in the form of comments. These should serve to explain the function of pieces of code that are particularly long or cryptic.

4 Notes

- Making your server multithreaded and implementing some kind of locking on the server side should not be that hard, so don't overthink it!
- Implementing two-way synchronization (and incremental updates) is the trickier part, espeically since there are so many possible cases for the states of the files on the client and server. You should have to store some extra state on the client and server to implement this.

- If you implemented incremental updates for Part 1, there still may be a bit of work to do before it works for Part 2, because there are now multiple clients to deal with, and the contents on the server may change between the times that a client synchronizes a file.

5 Threads and Thread Pools

The standard way of creating a thread in Java is to subclass `java.lang.Thread`, and create an object that implements run(). Then, instantiating an object of this class and calling start() on it will start the new thread. Note that calling run() on the thread will simply execute that function like a normal function, and start() will cause a new thread to be spawned which is running the function run(). For this part of the assignment, however, we want to use a thread pool. A thread pool holds a fixed number of threads. When we want to spawn a new thread, we then pass that duty to the thread pool, which will launch a new thread if there is a new thread that is free, otherwise it will wait for one of the threads in the thread pool to finish before starting a new thread. Fortunately for us, Java makes it extremely easy to create and use thread pools. To create a thread pool, all that needs to be done is to call `java.util.concurrent.Executors.newFixedThreadPool` with the desired size of the thread pool. We can call submit on the returned `java.util.concurrent.ExecutorService` object to submit a `java.lang.Thread` object to be run using a thread from the thread pool.

Here is a small example, to give an idea of what this means:

```
ExecutorService service = java.util.concurrent.Executors.newFixedThreadPool(2);
// MyThread is a class that extends java.lang.Thread and implements run()
MyThread myThread1 = new MyThread(param1, param2);
MyThread myThread2 = new MyThread(param1, param2);
MyThread myThread3 = new MyThread(param1, param2);

// The first two threads should immediately start executing
service.submit(myThread1);
service.submit(myThread2);
// This thread will have to wait for one of the others to finish before it can start
service.submit(myThread3);
```

For more information on threads and other concurrency methods, and their use specifically in Java, the Oracle documentation contains more information (http://docs.oracle.com/javase/tutorial/essential/concurrency/).

6 Locking and the `synchronized` Keyword

Using the `synchronized` keyword in Java is fairly intuitive. It can be used with an object in a statement, or be used on methods. The synchronized keyword, in this way, can be used the same way a mutex would. This is demonstrated below.

```
class Counter {
    private int i = 0;
    private Object lock = new Object();

    // While we are in this method, the instance of Counter
    // that increment() is being called on will be locked
    public void synchronized increment() {
        i++;
    }

    // While we are in this method, the instance of Counter
```

```
    // that decrement() is being called on will be locked
    // This means that increment and decrement are both locking
    // the same object and are thread safe.
    public void synchronized decrement() {
        i--;
    }

    public void incrementThenStartThread(Thread t) {
        // While in this block, this object will be locked
        synchronized (this) {
            i++;
        }
        // And now the object is no longer locked
        t.start();
    }

    public void lockAndStartThread(Thread t) {
        // This is essentially the same as a mutex
        synchronized (lock) {
            t.start();
        }
    }
}
```

For more information on other concurrency features Java has, the Oracle documentation contains more information (http://docs.oracle.com/javase/tutorial/essential/concurrency/).

7 Handin

To hand in your database implementation, run

```
cs131_handin dropbox-part2
```

from your project working directory. Make sure you include all .java files and your README.

If you wish to change your handin, you can do so by re-running the handin script. Only your most recent handin will be graded.