CSC 580 - Multithreaded Programming, Spring 2011, Assignment 3

Multithreaded server architecture using ExecutorService and CompletionService
Assignment is due by 11:59 PM on Saturday, April 2, 2011 via gmake turnitin.
Dr. Dale E. Parson, http://faculty.kutztown.edu/parson

This assignment consists of refactoring and testing a simulated networked server class to utilize thread pool and completion service infrastructure interfaces and classes studied in Chapters 6 and 7 of the textbook.

```
cp  ~parson/JavaLang/ThreadedServer.zip  ~/JavaLang/ThreadedServer.zip
cd  ~/JavaLang
unzip  ThreadedServer.zip
cd ./ ThreadedServer
gmake  clean  test
grep latency DEBUG.out
```

There are two Java source files in this project. **MP3Server.java** emulates a streaming media file server, and **ClientTestDriver.java** comprises the test driver for class MP3Server. Class MP3Server contains the following methods. Its full listing appears below.

MP3Server's constructor takes *threadPoolSize* and *bufferSize* as parameters. The threadPoolSize gives the number of threads used by an ExecutorService that you will apply from the java.util.concurrent library, and the bufferSize is the number of bytes of data transferred by a server task in each call to java.io.PipedOutputStream.write, transferring data to a client thread via java.io.PipedInputStream.read.

MP3Server method **makeRequest** takes a file name *request* parameter and returns an InputStream data connection that a client thread can read in order to obtain a media data stream. The interaction model consists of a client thread requesting a connection, which the server grants in the form of an InputStream, after which the client reads a media directory entry or a media content file via that particular request's InputStream.[1] MakeRequest sets up a connection.

MP3Server method **start** initiates the server's main service thread. (MP3Server is an *active class*.) Start must be called once before makeRequest invocations can proceed.

MP3Server method **shutdown** terminates the main server thread and any other threads within MP3Server, using controlled shutdown as explained in detail in the STUDENT code comments for shutdown. MP3Server will use a subset of the interruption and shutdown techniques of Chapter 7.

Class ClientTestDriver uses multiple client threads to send a set of requests to MP3Server. An ad hoc client thread pool reads work items from a work queue, where each work item is a file name to request from MP3Server. The client thread requests a file, receives an InputStream object, reads the contents of that InputStream object and stores them into a temporary file. It then compares the contents of the temporary file to a reference copy of the original file, reporting any differences to System.err, which the makefile redirects to file ClientTestDriver.out. The client and server also send debugging information via System.out to file DEBUG.out. This file came about in support of debugging the URL / URLConnection problem; it also houses a *latency* statement of central concern to this project.

[1] In the original design a request took the form of a java.net.URL object, which class MP3Server used in conjunction with a java.net.URLConnection object to read a file in the server address space, passing contents back to a client via a PipedOutputStream / PipedInputStream pair from the java.io package. Unfortunately, it appears that multiple instances of URL / URLConnection objects that refer to a common file interfere with each other when used by multiple concurrent threads. Most test runs succeeded, but occasionally tests would fail, with the server reading more bytes than were actually in a file. After much debugging, replacing URL / URLConnection with plain old FileInputStream objects within MP3Server eliminated the problem. Watch out for using classes URL / URLConnection within multithreaded servers!

All of your changes go into MP3Server.java. The handout, active class runs one main server thread, which your solution will continue using, with some code changes for interaction with an ExecutorService. It also uses one *ad hoc worker thread*, which your solution must replace with a *fixed thread pool ExecutorService* that it constructs via the *Executors* factory class using the *threadPoolSize* parameter passed to the MP3Server constructor. The main server thread dispatches work to individual **tasks** implemented as *Callable<Exception>* objects. These Callable<Exception> objects constitute a refactoring of the handout *AdHocWorker* Runnable class. Every variable and type using the name "AdHoc" within MP3Server must be refactored, replaced or eliminated in your design.

In addition to the fixed thread pool ExecutorService, MP3Server must construct and use an ExecutorCompletionService<Exception> object to manage completion of the Callable tasks. MP3Server must allocate one additional explicit thread, in addition to the main server thread. This second thread runs in a loop calling CompletionService.take in order to receive completion notification and completion status of the Callable tasks. When a task completes normally, it returns null, signifying successful completion. When a task encounters an error in copying file contents into its PipedOutputStream, it returns the Exception that identifies the problem. CompletionService.take returns this Exception to this second server thread via a Future<Exception> object; this thread reports any non-null Exception returned via a Future to System.err. The current test suite does not trigger any such errors.

I have performed extensive testing in the process of tracking down the URL / URLConnection problem. I also uncovered a problem with Networked File System performance on Ron that has been corrected.

Unlike the penny-dime puzzle, which was CPU bound, this project is I/O bound. The main bottleneck of interest in the original, handout code comes from the fact that there is only one ad hoc worker thread. You will replace this thread with Callable tasks that run within a fixed thread pool ExecutorService. I have found after much testing that adding threads via the ExecutorService makes limited improvement to the throughput, mostly because the throughput is limited by the I/O speed of networked file system (NFS). Your solution should run a little faster. The important time measurement in this project is *latency* between the time that a client thread requests a connection and the time that it begins to read data via its PipedInputStream. In the handout code there is only one ad hoc worker thread within MP3Server, and when concurrent connection requests arrive from client threads, each client thread must wait until the server worker thread has completed transferring one or more complete media files before that client thread sees any data arriving in its PipedInputStream. Running **grep latency DEBUG.out** after **gmake clean test** reveals the average latency between the time a client requests a connection and the time it actually receives data. The server grants each connection request very rapidly, returning a buffered PipedInputStream to the client via its own thread, but the data begin to arrive only when a server worker thread gets time to feed that data pipe.

Your solution should decrease this startup latency dramatically because, with as many server pool threads as client threads (8 each in the makefile test case), data transfer need not occur within the constraint of one file at a time. While overall throughput increases modestly, latency decreases because multiple server threads can interleave their I/O constrained access to NFS. On Harry I have found average startup latency decrease from around 2000 milliseconds to around 10 milliseconds in going to my solution to project 3. Hermione and Ron show similar improvement. This architecture is meant to mirror architectures such as web servers, in which *low-latency, staged responses* to user requests in the form of display of HTML text, followed later by media such as images or audio streams, *takes priority over total throughput* on a bandwidth-constrained network connection. This is a very different but still realistic application of multithreading compared to the state machine (with CyclicBarrier) and dataflow (without CyclicBarrier) solutions to project 2.

Here is what an initial test run of the handout code looks like on Harry.

```
-bash-3.00$ gmake clean test
/bin/rm -f *.o *.class .jar core *.exe *.obj *.pyc
/bin/rm -f *.class *.out *.dif ./tmpfiles/*
/bin/bash -c "javac -g ClientTestDriver.java"
/bin/rm -f ./tmpfiles/*
time /bin/bash -c "java ThreadedServer.ClientTestDriver 8 1024 8 8 >>DEBUG.out 2>ClientTestDriver.out"
```

```
real        5.0
user        5.6
sys         1.7
diff ClientTestDriver.out ClientTestDriver.ref > ClientTestDriver.dif
-bash-3.00$ grep latency DEBUG.out
Average initial latency until 1st response to client: 2494 msecs.
-bash-3.00$
```

After a successful test run, ClientTestDriver.out should be empty (no errors logged) and DEBUG.out contains latency and assorted debugging information from the URL / URLConnection problem. The command line usage is as follows. The handout code ignores the NUM_SERVERTHREADS command line argument. Look for STUDENT comments in the handout code. When you have it working, use **gmake turnitin** before the end of the due date.

```
java ClientTestDriver NUM_CLIENTTHREADS BUFFERSIZE NUM_REQUESTS NUM_SERVERTHREADS
```

~/JavaLang/MP3Server.java

```java
/* MP3Server.java -- Assignment 3 server class.
   Dr. Dale Parson, CSC 580, Spring 2011.
*/

package ThreadedServer ;
import java.util.concurrent.LinkedBlockingQueue ;
import java.util.concurrent.Semaphore ;
import java.util.concurrent.atomic.AtomicInteger ;
import java.io.InputStream ;
import java.io.FileInputStream ;
import java.io.File ;
import java.io.OutputStream ;
import java.io.BufferedInputStream ;
import java.io.BufferedOutputStream ;
import java.io.PipedInputStream ;
import java.io.PipedOutputStream ;
import java.io.IOException ;
import net.jcip.annotations.* ;

/**
    Multithreaded class that emulates a networked server. This
    class runs in the same process as its ClientTestDriver
    because its goal is to exercise capabilities of a multithreaded
    server that uses the java.util.concurrent.ExecutorCompletionService
    to distribute requests among a thread pool, await their completion,
    and print a diagnostic error message to System.err if the task in
    the thread pool encountered an Exception while sending a copy of
    a requested resource to a client process or thread. The completion
    status consists of an Exception object reference == null if no error
    occurs, or is non-null if an error occurs; this server uses
    the CompletionService to allow worker threads to inform the main
    server thread of either successful completion or Exception, and
    the main server thread logs the Exception getMessage() to System.err.
    There is no point in adding networked overhead to this exercise project.
    We want to measure improvements in server-side multithreading.
    The initial implementation uses a single ad hoc thread.
    STUDENTS must refactor it into a multithreaded server using
    ExecutorCompletionService with a fixed-size thread pool.
    @author Dr. Dale Parson
```

```
**/
@ThreadSafe
public class MP3Server implements Runnable {
    private final static int EXITERROR = 1 ;
    // Size of the InputStream and OutputStream byte buffers.
    private final int bufferSize ;
    @GuardedBy("this")
    private volatile Thread mainServerThread = null ;
    private final AtomicInteger shutdownCount = new AtomicInteger(0);
    private final LinkedBlockingQueue<mainThreadRequest> mainThreadQ
        = new LinkedBlockingQueue<mainThreadRequest>();
    // STUDENT get rid of this next field. It is here to allow only
    // 1 AdHocWorker thread to run at a time.
    // STUDENT get rid of every class or field with "AdHoc" in its name.
    private final Semaphore AdHocLimit = new Semaphore(1);
    /**
     *  Construct a MP3Server using a ExecutorCompletionService with
     *  a fixed number of threads in its thread pool.
     *  @param threadPoolSize is the number of threads in the pool,
     *  must be > 0. The initial, single-threaded implementation does
     *  nothing with this parameter. Students must change that fact.
     *  @param bufferSize is the size of the InputStream and OutputStream
     *  byte buffers.
    **/
    public MP3Server(int threadPoolSize, int bufferSize) {
        this.bufferSize = bufferSize ;
    }
    /**
     *  Request a data stream whose source is a file, and whose contents
     *  are to be streamed to a client reader. This method is
     *  synchronized in order to restrict connection initialization
     *  to a single client thread at a time.
     *  @param request is the local file path to the data.
     *  @return InputStream for the client to read.
     *  @throws IOException on an invalid request.
     *  @throws InterruptedException if shutdown is invoked on this
     *  object while initializing a request, or if shutdown has already
     *  occurred.
    **/
    public synchronized InputStream makeRequest(String request)
            throws IOException, InterruptedException {
        InputStream istream = new FileInputStream(new File(request));
        BufferedInputStream bufistream ;
        if (istream instanceof BufferedInputStream) {
            bufistream = (BufferedInputStream) istream ;
        } else {
            bufistream = new BufferedInputStream(istream);
        }
        InputStream clientStream = submitWorkerTask(bufistream);
        return clientStream ;
    }
    // Either the caller or submitWorkerTask must be synchronized
    // to ensure safe publication of the inter-thread data pipe.
    private BufferedInputStream submitWorkerTask(
            BufferedInputStream serverResource)
                throws IOException, InterruptedException {
```

```
        if (mainServerThread == null) {
            throw new IOException(
            "MP3Server requires one call to start() before makeRequest.");
        }
        PipedOutputStream serverOutput = new PipedOutputStream();
        PipedInputStream clientStream = new PipedInputStream(serverOutput);
        // Placing the serverResource and serverOutput in a
        // BlockQueue guarantees safe publication to the main thread. p. 52
        mainThreadQ.put(new mainThreadRequest(serverResource,
            new BufferedOutputStream(serverOutput)));
        return new BufferedInputStream(clientStream) ;
    }

    /**
     *  Start the main server thread for this MP3Server active object.
     *  This method must be called once and only once before any calls
     *  to makeRequest.
    **/
    public synchronized void start() {
        if (mainServerThread == null) {
            mainServerThread = new Thread(this);
            // STUDENT COMMENT OUT NEXT LINE WHEN YOU HAVE METHOD
            // shutdown() working correctly. It is currently in here
            // because shutdown() is not implemented, so the main server
            // thread never exits. Writing shutdown() will fix that.
            mainServerThread.setDaemon(true); // COMMENT THIS OUT!
            mainServerThread.start();
        }
    }

    @ThreadSafe
    private class mainThreadRequest {   // container for data
        public final InputStream serverResource ;
        public final OutputStream serverOutput ;
        public mainThreadRequest(InputStream serverResource,
            OutputStream serverOutput) {
            this.serverResource = serverResource ;
            this.serverOutput = serverOutput ;
        }
    }
    /**
     *  Used to start server thread, do not invoke from external client.
    **/
    public void run() {     // Main server thread, not part of pool.
        synchronized(this) {
            if (mainServerThread == null
                    || ! mainServerThread.equals(Thread.currentThread())) {
                System.err.println("FATAL ERROR, "
                    + "Multiple threads in MP3Server");
                System.exit(EXITERROR);
            }
        }
        while (shutdownCount.get() == 0) {
            mainThreadRequest rqst = null;
            boolean gotAdHocLimit = false ;
            try {
                AdHocLimit.acquire();
                gotAdHocLimit = true ;
                rqst = mainThreadQ.take();
            } catch (InterruptedException dying) {
                // Shutdown is under way.
                if (gotAdHocLimit) {
                    AdHocLimit.release();
                }
                continue ;      // Terminate main server thread.
            }
            Thread th = new Thread(new AdHocWorker(
                rqst.serverResource, rqst.serverOutput,
                    AdHocLimit, bufferSize));
            th.start();
        }
    }
    // In this implementation the AdHocWorker thread does the work below.
    // In the STUDENT implementation this method must submit a
    // Callable<Exception> task to the ExecutorCompletionService
    // set up by the constructor, where that Callable<Exception> task
    // does all of the following work. When it has completed its work,
    // it passes a null reference back to the main server thread as
    // its result on success, or passes the Exception that occurred
    // as a result of a failed read or write or close operation.
    // The main server thread must log the getMessage() of any
    // Exception it receives from the Callable to System.err.
    // Typical exceptions include IOException on failed Input/Output
    // or InterruptedException which may be raised by invoking shutdown
    // on this object.
    @ThreadSafe
    private static class AdHocWorker implements Runnable {
        private final InputStream serverResource ;
        private final OutputStream serverOutput ;
        private final Semaphore AdHocLimit ;
        private final int bufferSize ;
        public AdHocWorker(InputStream serverResource,
                OutputStream serverOutput, Semaphore AdHocLimit,
                int bufferSize) {
            this.serverResource = serverResource ;
            this.serverOutput = serverOutput ;
            this.AdHocLimit = AdHocLimit ;
            this.bufferSize = bufferSize ;
        }
        public void run() {
            try {
                byte [] buffer = new byte [ bufferSize ];
                int bytesread, totalbytes = 0 ;
                for (bytesread = serverResource.read(buffer) ; bytesread != -1
                        ; bytesread = serverResource.read(buffer)) {
                    serverOutput.write(buffer, 0, bytesread);
                    totalbytes += bytesread ;
                }
                serverResource.close();
                serverOutput.close();
                System.out.println("DEBUG server wrote " + totalbytes
                    + " into Pipe.");
            } catch (IOException ioex) {
                // STUDENT must make this ioex the result of the Callable
                // passed back to the main server thread.
                // This AdHocWorker thread silently terminates.
                return ;
            } finally {
                AdHocLimit.release();
            }
        }
    }
    /**
     *  Shutdown the server. The current implementation does nothing.
     *  The STUDENT implementation must invoke ExecutorService.shutdown
     *  the first time that this method is invoked; it must invoke
     *  ExecutorService.shutdownNow the second time it is called, and must
     *  ignore any additional calls. Also, this method must disable
     *  any subsequent calls to makeRequest; any call to makeRequest after
     *  a call to shutdown must throw an InterruptedException informing
     *  the client that shutdown has occurred. Since a call to
     *  makeRequest may occur concurrently with a call to shutdown,
     *  STUDENTS must use thread-safe mechanisms to ensure that all
     *  concurrent invocations see only consistent data.
     *  Work includes incrementing shutdownCount to 1 or 2 (NO HIGHER),
     *  interrupting mainServerThread if it is non-null,
     *  and shutting down ExecutorService as directed above.
    **/
    public synchronized void shutdown() {
    }
}
```

~/JavaLang/ClientTestDriver.java

```
/* ClientTestDriver.java -- Assignment 3, client test driver for
   Multithreaded implementation of test driver for MP3Server.java.
   Dr. Dale Parson, CSC 580, Spring 2011.
   There are NO STUDENT changes in this file.
*/

package ThreadedServer ;
import java.util.Scanner ;
import java.util.LinkedList ;
import java.util.concurrent.ConcurrentLinkedQueue ;
import java.util.concurrent.atomic.AtomicLong ;
import java.util.concurrent.atomic.AtomicInteger ;
import java.io.InputStream ;
import java.io.FileInputStream ;
import java.io.BufferedInputStream ;
import java.io.OutputStream ;
import java.io.FileOutputStream ;
import java.io.BufferedOutputStream ;
import java.io.File ;
import java.io.IOException ;
import net.jcip.annotations.* ;

/**
    Class to test MP3Server.
   @see MP3Server
   @author Dr. Dale Parson
**/
@ThreadSafe
public class ClientTestDriver {
    private final static int EXITERROR = 1 ;
   private final static AtomicLong sumLatencies = new AtomicLong(0L);
   /**
    *  USAGE: java ClientTestDriver NUM_CLIENTTHREADS BUFFERSIZE
    *      NUM_REQUESTS NUM_SERVERTHREADS
   **/
   private static final String usage =
   "USAGE: java ClientTestDriver NUM_CLIENTTHREADS BUFFERSIZE "
      + "NUM_REQUESTS NUM_SERVERTHREADS";
   public static void main(String [] args) {
      if (args.length != 4) {
         System.err.println(usage);
         System.exit(EXITERROR);
      }
      int clithreads = -1, bufsize = -1, requests = -1, srvthreads = -1 ;
      try {
         clithreads = Integer.parseInt(args[0]);
         bufsize = Integer.parseInt(args[1]);
         requests = Integer.parseInt(args[2]);
         srvthreads = Integer.parseInt(args[3]);
      } catch (NumberFormatException fx) {
         System.err.println("Invalid integer on command line: "
            + args[0] + " " + args[1] + " " + args[2] + " " + args[3]);
         System.exit(EXITERROR);
      }
      if (clithreads < 1 || bufsize < 1 || requests < 1 || srvthreads < 1) {
         System.err.println("Invalid integer on command line: "
            + args[0] + " " + args[1] + " " + args[2] + " " + args[3]);
         System.exit(EXITERROR);
      }
      try {
         MP3Server server = new MP3Server(srvthreads, bufsize);
         server.start();
         InputStream directoryStream = server.makeRequest("mp3files.txt");
         LinkedList<String> fnames = new LinkedList<String>();
         Scanner scanner = new Scanner(directoryStream);
         while (scanner.hasNextLine()) {
            fnames.add(scanner.nextLine());
         }
         scanner.close();
         if (fnames.size() == 0) {
            throw new IOException(
               "ERROR: Dirctory of MP3 resources is empty.");
         }
         // Set up clientRequestQueue before starting any worker threads.
         ConcurrentLinkedQueue<String> clientRequestQueue
            = new ConcurrentLinkedQueue<String>();
         for (int i = 0 ; i < requests ; i++) {
            clientRequestQueue.add(fnames.get(i % fnames.size()));
         }
         Thread [] threads = new Thread [ clithreads ] ;
```

```
            for (int i = 0 ; i < clithreads ; i++) {
                ClientTestHelper helper = new ClientTestHelper(
                    clientRequestQueue, server, bufsize);
                threads[i] = new Thread(helper);
                threads[i].start();
            }
            for (int i = 0 ; i < clithreads ; i++) {
                threads[i].join();
            }
            server.shutdown();
            System.out.println(
                "Average initial latency until 1st response to client: "
                + (sumLatencies.get() / requests) + " msecs.");
        } catch (IOException ioex) {
            System.err.println("CLIENT IO ERROR : "
                + ioex.getMessage());
            System.exit(EXITERROR);
        } catch (InterruptedException intrx) {
            System.err.println("CLIENT INTERRUPTED ERROR : "
                 + intrx.getMessage());
             System.exit(EXITERROR);
         }
    }
    @ThreadSafe
    private static class ClientTestHelper implements Runnable {
        private final ConcurrentLinkedQueue<String> clientRequestQueue ;
        private final MP3Server server ;
        private final byte [] buffer ;
        private final byte [] cmpbuffer ;
        public ClientTestHelper(ConcurrentLinkedQueue<String> requestQ,
                MP3Server theServer, int bufsize) {
            clientRequestQueue = requestQ ;
            server = theServer ;
            buffer = new byte [ bufsize ];
            cmpbuffer = new byte [ bufsize ];
        }
        private static final AtomicInteger tmpfilenum
            = new AtomicInteger(0);
        public void run() {
            InputStream toread = null ;
            File tmpfile = null ;
            OutputStream towrite = null ;
            File tmpdir = new File("tmpfiles");
            for (String todo = clientRequestQueue.poll() ; todo != null
                    ; todo = clientRequestQueue.poll()) {
                try {
                    long before = System.currentTimeMillis();
                    toread = server.makeRequest(todo);
                    // Read one byte for most exact timing until first
                    // available data, before setting up tmpfile.
                    int firstByte = toread.read();
                    int totalbytes = 0 ;
                    long after = System.currentTimeMillis();
                    sumLatencies.addAndGet(after - before);
                    // Fold a unique integer into the tmpfile name and make
                    // sure that no 2 threads call createTempFile concurrently.
                synchronized (tmpfilenum) {
                    tmpfile = File.createTempFile(("media"
                        + tmpfilenum.getAndIncrement() + "_"),
                            ".tmp",tmpdir);
                }
                towrite = new BufferedOutputStream(
                    new FileOutputStream(tmpfile));
                if (firstByte != -1) {
                    towrite.write(firstByte);
                    totalbytes = 1 ;
                    int bytesread ;
                    for (bytesread = toread.read(buffer) ;
                            bytesread != -1
                            ; bytesread = toread.read(buffer)) {
                        towrite.write(buffer, 0, bytesread);
                        totalbytes += bytesread ;
                    }
                }
                toread.close();
                toread = null ;
                towrite.close();
                towrite = null ;
                System.out.println("DEBUG client wrote " + totalbytes
                    + " into tmp file.");
                String diffstring = cmpBytes(todo, tmpfile, totalbytes);
                if (diffstring == null) {   // cmp is OK
                    tmpfile.delete();
                } else {
                    System.err.println("ERROR: Client compare diff: "
                        + diffstring);
                    System.exit(EXITERROR);
                }
            } catch (IOException ioex) {
                System.err.println("WARNING: Client I/O Exception: "
                    + ioex.getMessage());
            } catch (InterruptedException intrx) {
                System.err.println("WARNING: Client Thread Interrupted: "
                    + intrx.getMessage());
            } finally {
                try {
                    if (toread != null) {
                        toread.close() ;
                        toread = null ;
                    }
                    if (towrite != null) {
                        towrite.close();
                        towrite = null ;
                    }
                    // Leave tmpfile intact if it choked above.
                    tmpfile = null ;
                } catch (IOException shouldNotHappen) {
                }
            }
        }
    }
    private String cmpBytes(String todo, File tmpfile, int totalbytes) {
        try {
            InputStream reference = new FileInputStream(
                new File(todo));
            if (! (reference instanceof BufferedInputStream)) {
                reference = new BufferedInputStream(reference);
            }
            BufferedInputStream tmpdata = new BufferedInputStream(
                new FileInputStream(tmpfile));
            int refbytes, tmpbytes, location = 0 ;
            for (refbytes = reference.read(buffer),
                        tmpbytes = tmpdata.read(cmpbuffer) ;
                    refbytes != -1 && refbytes == tmpbytes ;
                    refbytes = reference.read(buffer),
                        tmpbytes = tmpdata.read(cmpbuffer)) {
                for (int i = 0 ; i < refbytes ; i++) {
                    if (buffer[i] != cmpbuffer[i]) {
                        return new String("CMP ERROR BETWEEN "
                            + todo + " and " + tmpfile.getPath()
                            + " at byte " + (location + i)
                            + " on " + totalbytes
                            + " read via input Pipe.");
                    }
                }
                location += refbytes ;
            }
            if (refbytes != tmpbytes) {
                return new String("CMP ERROR BETWEEN LENGTH OF "
                    + todo + " and " + tmpfile.getPath()
                    + " at end of files.");
            }
            reference.close();
            tmpdata.close();
        } catch (Exception diffx) {
            return new String("Exception during COMPARE: "
                + diffx.getMessage());
        }
        return null ;
    }
  }
}
```