(**V.1.0**, 24 October 2021)

Practical Exercises in Computer Networks and Distributed Systems

Java Threads for speeding up Clients and Servers (WIP)

© 2013-2021, José María Foces Morán and José María Foces Vivancos

In this lab we extend the file service C/S application that we completed on the preceding lab exercise by providing more protocol functions and migrating the server from the Single Threaded model (ST) to the more powerful Multi Threaded model (MT).

Brief introduction to Java Threads

Java Threads allow Java programs to be run concurrently, which has many uses in distributed programming and also offers substantial improvements in program speed and resource utilization. In general, a thread of execution is a set of resources that can be applied to a program turning it into a running program. In Java, a class can be run in a Thread other than that of the invoking object, thereby permitting both threads to continue running concurrently. In Java, one can create a thread and hand it an object to run in two ways:

- The object to be run extends Thread()
- The object to be run implements java.lang.Runnable

We will normally opt for the second technique since it will allow our class to implement other interfaces if necessary –the first option no other classes can be extended since Java does not support built-in multiple inheritance.

In the first version of the file service, its server was implemented using the ST model, therefore, once a request was received from a client, the server was stopped then a request server was started to honor the request just received, therefore, the server was practically stopped, completely unable to serve any other incoming client requests. The server throughput is negatively affected by those idle times, and also its availability and the utilization level of the system will be rather low, thus, in order to improve the performance and availability we will implement the server with the MT model.

The Java Virtual Machines of today can also make use of the multi-core microprocessors, thus, several threads of a single process can be run in parallel with no special provision beyond the MT model. Still, with older single-core microprocessors, the concurrent, MT model is convenient since it will allow the programs to achieve higher utilization and higher throughput.

Writing the server to the Java MT (Multi-threaded) and Single-Object (SO) model

In fig. 1 we illustrate how the MT server delegates a new request to a single instance of RequestResponseHandler –note that, now, both the Server and the RequestResponseHandler run concurrently. According to our preceding explanation, the request's delegate (An instance of RequestResponseHandler) will be executed in a new thread, other than the server's, therefore RequestResponseHandler must implement Runnable. You may want to skim the following code so that you understand the basics of Java threading, the code comes from our current C/S exercise:

Collaboration between the Server and the RequestResponseHandler instances

The server starts in its serverMainLoop() method by instantiating a, single, RequestResponseHandler object and handing it a reference to the Welcome Socket that was created in the server's constructor (A java.net.ServerSocket object). The Welcome Socket becomes a field of the *single* RequestResponseHandler object. The server then enters an infinite loop in which it will create a new Thread on which it starts execution of the RequestResponseHandler, then, it executes the wait() method of the RequestResponseHandler (Variable rs). This method waits for another thread to call rs.notify(); when that happens, wait suspends waiting and continues to the next loop iteration. All in all, the infinite loop, first creates a new thread and executes the RequestResponseHandler, waits for the latter to send its notify() message and iterates again. It runs a new thread on the Handler and waits, forever.

Every time the server runs a thread on the Handler (RequestResponseHandler), as specified in Java threads, it will start execution on its run() method, where it will fetch the global variable ws (The Welcome Socket) and call accept:

```
synchronized(this){
  Socket ds = ws.accept();
  this.notify();
}
```

Assume a new connection request arrives from a client (See Figure 3), then, accept() will return a new delegate socket (Variable ds). Note, variable ds is a local variable, which means it is exclusively owned by each thread running this code. From this time on, a successive method call chain will take place which will satisfy the client request semantics; in this call chain, local variable ds will be handed in order to permit the receiving methods to receive and send messages onto it. When the call chain finishes execution, the thread object assigned to it is destroyed. Also, right after local variable ds has successfully received the return value from ws.accept(), it is allowed to send the notify() message, which will wakeup the server to continue to the next iteration. Note, the notify() invocation must take place on the RequestResponseHandler object, which from its inside is referenced with the Java this reference (See the code snippet right above this paragraph). Recall, the server invoked the counterpart wait() method from exactly the same object, in this case, referenced from variable rs in the server.

```
synchronized(rs){
  rs.wait();
}
```

It's worth to observe that the assignment of the ws.accept() return value and the call to this.notify() are atomic by the use of a synchronized block on the RequestResponseHandler object instance (this). Also, the server creates a counterpart synchronized block on the same object, in this case, referenced by variable rs. All in all, wait() continues when it receives the notify() message if both are from the same object (the RequestResponseHandler object), and, both actions must be included in their respective synchronized blocks from the same rs object.

```java
public void serverMainLoop() {
    /*
     * The object on which we will be delegating the low level protocol
     * handling and the resulting Request/Responses
     *
     * This instance of RequestResponseHandler is the single instance used
     * across all the successive clients: We are using a SINGLE OBJECT
     * MULTITHREADING model
     */
    RequestResponseHandler rs = new RequestResponseHandler(welcomeSocket, fileStorageBaseDir);

    while (true) {

        Thread t = new Thread(rs);

        System.out.println("\t· Next upcoming client thread created: " + t.getName());
        System.out.flush();

        t.start();

        System.out.println("\t· Server is waiting for a new client's connection to complete");
        System.out.flush();

        /*
        * Wait for a notification coming from the single-instance of RequestResponseHandler that
        * is being run by thread t, when that notification arrives, we are sure that a new TCP
        * connection was completed, then the loop iterates again which causes a new thread to be
        * created and started which effectively translates into servicing a newly arrived client
        */
        synchronized (rs) {
            try {
                rs.wait();
                System.out.println("\t· Server has been notified client " + t.getName() + " has es
                System.out.flush();
            } catch (InterruptedException ex) {
                Logger.getLogger(MultiThreadedServer.class.getName()).log(Level.SEVERE, null, ex);
            }
        }

    }
```

Fig. 1. Server's main loop in the MT model

```
* MULTI-THREADED RR handler: once a C/S connection has been established,
 * this object receives a delegate socket that represents a TCP connection with
 * a client program that is interested in executing some of the services offered
 * by the server.
 *
 …
 */
package servers;

import java.io.*;
import java.net.*;
…

public class RequestResponseHandler implements Runnable{
```

Fig. 2. Request delegate object (RequestResponseHandler) implements java.lang.Runnable

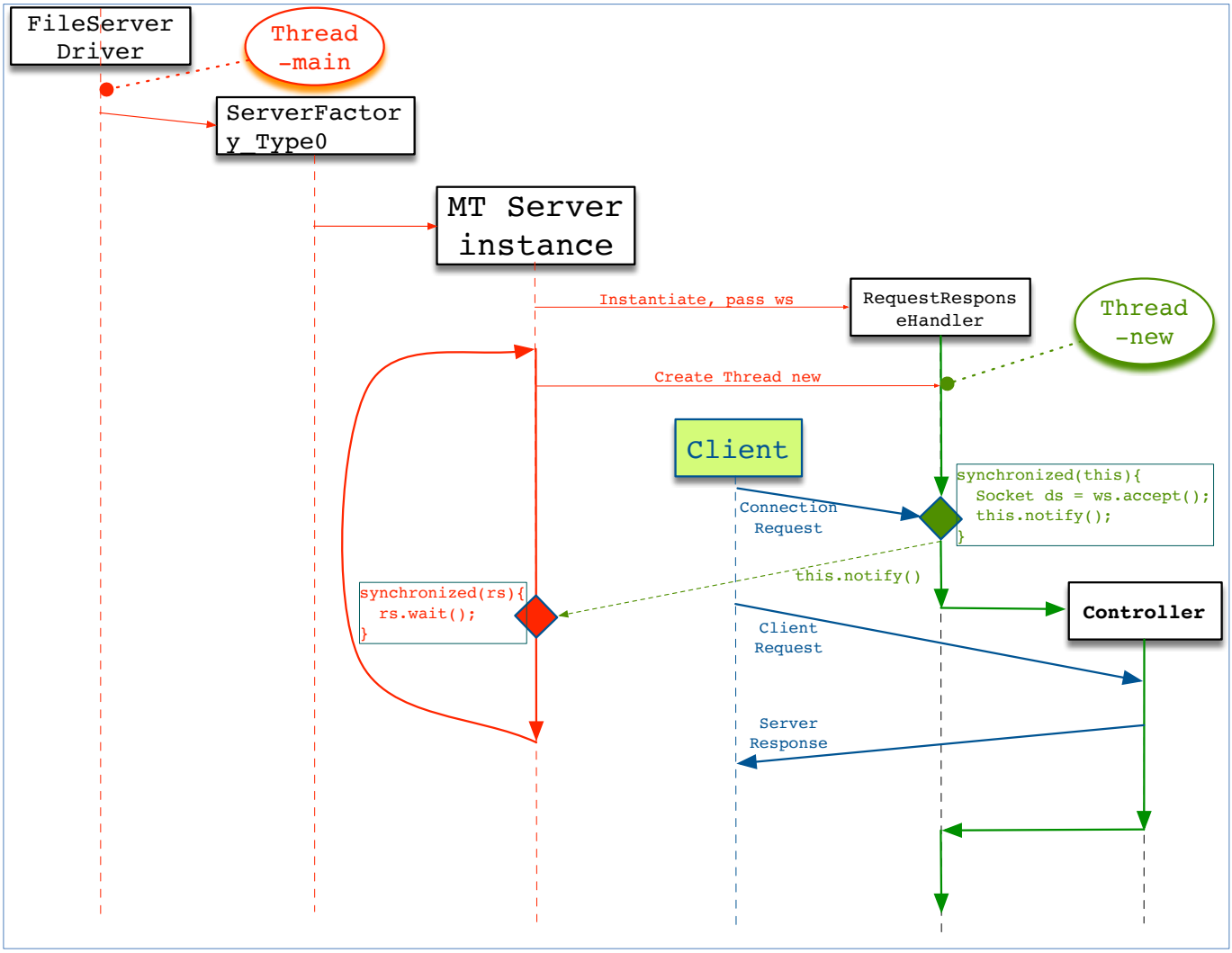

Fig. 3. Server main loop and RequestResponseHandler run concurrently (MT, single-object model)

Notes

- Old versions of JDK might have problems when compiling code containing comments that include certain accented ASCII characters. Having JDK accept UTF-8 accented characters entails using the following command-line parameters:
    - $ javac **-encoding utf8** Program.java
-

Exercises

1. Donwload, compile and test the software as provided, you should be able to upload any file from the client to the server.

2. Identify the most significant code sections:
    a. Server creating a single RequestResponseHandler and creating and running multiple successive threads for servicing the upcoming client connections
    b. Note that the following protocol functions have been implemented:
        i. Download
        ii. Upload
        iii. File List
        iv. File exists
    c. Check the correct functioning of all the preceding functions, you will have to use the clients included. Check each and every function and if some functionality has not been implemented or has been implemented partially, provide your own implementation; finally, test any code added or modified by you

3. Perform cross-checking of your clients and server with other classmate's. Your clients and servers must interoperate with no problems

4. What adverse effect could happen if we moved the Socket ds = ws.accept(); sentence out from the synchronized block?

Appendices

The source code included hereon is being continually **updated**, consequently, we publish the updated versions on the web; the URL is:

http://paloalto.unileon.es/ds/lab/CS-MT-SO.zip

Source code

```
/**
 * ********************************************************************
 * Universidad de León, EIII Grado en Ingeniería Informática Asignatura de
 * Arquitectura de Sistemas Distribuidos (C) 2012 José María Foces Morán,
 * instructor.
 *
 * FileServerDriver.java
 *
 * SINGLE RESPONSIBILITY OF THIS CLASS:
 * Start execution of a file server corresponding to interface Server which
 * only implementation, for the time being is SingleThreadedServer.java
 *
 * Obtains an instance of Server according to arguments received via stdin and
 * calls the new server's tight loop
 *
 ********************************************************************
 */
package servers;

public class FileServerDriver {
```

```
    public static void main(String[] args) {

        int port = Integer.parseInt(args[0]);
        String baseDir = args[1];
        boolean mt = Boolean.parseBoolean(args[2]);

        Server s = null;

        if (args.length == 2) {
            s = ASDFileServerFactory.getInstance(port, baseDir);
        } else if (args.length == 3) {
            s = ASDFileServerFactory.getInstance(port, baseDir, mt);
        } else {
            System.err.print("Usage: fsd <tcp port> <base dir> [mt true/false]");
            System.exit(-1);
        }

        if (s != null) {
            s.serverMainLoop();
        } else {
            System.err.print("Server null, could not be started");
            System.exit(-1);
        }
    }
}
```

```
/*************************************************************
 * Universidad de León, EIII
 * Grado en Ingeniería Informática
 * Asignatura de Arquitectura de Sistemas Distribuidos
 * (C) 2012 José María Foces Morán, instructor.
 *
 * ASDFileServerFactory.java
 *
 * SINGLE RESPONSIBILITY OF THIS CLASS:
 * Correctly produce file server instances
 *
 *************************************************************/

package servers;

import java.io.*;
import java.net.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ASDFileServerFactory {

    static final int BASEPORT = 1024;

    /**
     * Creates a new Server, checks port is not root and that the base
     * storage folder exists and is rw, the new server uses the ST or the
     * MT implementation according to variable MultiThreaded false or true
     * respectively
     *
     */
    public synchronized static Server getInstance(
            int port,
            String fileStorageBaseDir,
            boolean MultiThreaded) {

        Server s = null;

        /*
         * Make sure the base storage folder exists and is rw
         */
        File f = new File(fileStorageBaseDir);

        System.out.println("f.getName() = "+f.getName());
        System.out.println("f.path() = "+f.getPath());
        System.out.println("f.getAbs: " + f.getAbsolutePath() + "  f.getPath "+
f.getPath());
        System.out.println("exists:"+f.exists()+" abs = "+f.isAbsolute()+" dir =
"+f.isDirectory()+" r = "+f.canRead()+"  w = "+f.canWrite());
        System.out.flush();

        if (f.isAbsolute() && f.isDirectory() && f.canRead() && f.canWrite()) {

            /*
             * Make sure a root port is not specified, otherwise,
             * return null and finish
             */
            if (port > BASEPORT) {

                if (MultiThreaded) {
                    s = (Server) new MultiThreadedServer(port, f);
                } else {
                    s = (Server) new SingleThreadedServer(port, f);
```

```
                }

                System.out.println("Server: Welcome socket on port " + port + "
created");
                System.out.flush();

            } else {

                s = null;

                System.err.println("Problem on ServerSocket creation");
                System.err.flush();

            }

        }

        return s;

    }

    /**
     * Two-argument SINGLE THREADED factory method
     */
    public synchronized static Server getInstance(
            int port,
            String fileStorageBaseDir) {

        Server s = null;

        s = getInstance(
                port,
                fileStorageBaseDir,
                false);

        return s;

    }
}
```

```
/**
 * ********************************************************************
 * Universidad de León, EIII Grado en Ingeniería Informática Asignatura de
 * Arquitectura de Sistemas Distribuidos (C) 2014 José María Foces Morán,
 * instructor.
 *
 * RequestResponseHandler.java
 *
 * SINGLE RESPONSIBILITY OF THIS CLASS: Concurrently manage each successive
 * upcoming client connection
 *
 * Wrap the delegate socket obtained from the Welcome Socket into
 * an ObjectOutputStream and an InputObjectStream and hand them to a Controller
 * object
 *
 * By using the delegate socket resulting from each successive TCP connection
completion,
 * it creates the downstream and the upstream and calls a new command
 * dispatcher to take care of the specific client commands received
 * ********************************************************************
 */
package asdfileservice.servers;

import java.io.*;
import java.net.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class RequestResponseHandler implements Runnable {

    ServerSocket welcomeSocket = null;
    File fileStorageBaseDir = null;


    public RequestResponseHandler(ServerSocket welcomeSocket, File
fileStorageBaseDir) {

        this.welcomeSocket = welcomeSocket;
        this.fileStorageBaseDir = fileStorageBaseDir;

    }

    /*
     *
     * The object i/o streams are created wrapping the delegate socket, all is
     * ready for performing the application level protocol dialog with the
     * client. The client-server protocol is made up of messages that have been
     * established by the protocol/software designer. We refer to those messages
     * as commands (protocol commands). The CommandDispatcher instance will
     * receive commands from the client by using object ois and send the
     * responses by using object oos
     *
     */
    public void run() {

        Socket delegateSocket = null;
        ObjectInputStream ois = null;
        ObjectOutputStream oos = null;

        try {
```

```
            synchronized(this){
                delegateSocket = welcomeSocket.accept();
                notify();
            }


            ois = new ObjectInputStream(delegateSocket.getInputStream());
            oos = new ObjectOutputStream(delegateSocket.getOutputStream());

        } catch (IOException ex) {

Logger.getLogger(RequestResponseHandler.class.getName()).log(Level.SEVERE, null, ex);
        }

        new Controller().dispatch(ois, oos, fileStorageBaseDir);

    }
}
```

```
/**
 * ****************************************************************************
 * Universidad de León, EIII Grado en Ingeniería Informática Asignatura de
 * Arquitectura de Sistemas Distribuidos (C) 2014 José María Foces Morán,
 * instructor.
 *
 * MultiThreadedServer.java
 *
 * SINGLE RESPONSIBILITY OF THIS CLASS:
 * Using a ServerSocket has a single instance of RequestResponseHandler run on a
 * new Thread and waits for it to invoke notify() when it hase completed a new
 * TCP connection, then it proceeds to create a new thread, pass it the
 * ServerSocket instance, etc. The RequestResponseHandler instance is
 * responsible for processing of the C/S RR protocol
 *
 ******************************************************************************
 */
package asdfileservice.servers;

import java.io.*;
import java.net.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class MultiThreadedServer implements Server {

    private ServerSocket welcomeSocket = null; //The welcome socket
    private File fileStorageBaseDir = null;

    public MultiThreadedServer(int port, File fileStorageBaseDir) {

        this.fileStorageBaseDir = fileStorageBaseDir;

        /*
         * Create a new ServerSocket instance on the specified port
         */
        try {

            welcomeSocket = new ServerSocket(port);

        } catch (IOException ex) {
            Logger.getLogger(ASDFileServerFactory.class.getName()).log(Level.SEVERE, null, ex);
            System.exit(-1);
        }

    }


    public void serverMainLoop() {
        /*
         * The object on which we will be delegating the low level protocol
         * handling and the resulting Request/Responses
         *
         * This instance of RequestResponseHandler is the single instance used
         * across all the successive clients: We are using a SINGLE OBJECT
         * MULTITHREADING model
         */
        RequestResponseHandler rs = new RequestResponseHandler(welcomeSocket, fileStorageBaseDir);

        while (true) {

            Thread t = new Thread(rs);

            System.out.println("\t· Next upcoming client thread created: " + t.getName());
            System.out.flush();

            t.start();

            System.out.println("\t· Server is waiting for a new client's connection to complete");
            System.out.flush();

            /*
            * Wait for a notification coming from the single-instance of RequestResponseHandler that
            * is being run by thread t, when that notification arrives, we are sure that a new TCP
            * connection was completed, then the loop iterates again which causes a new thread to be
            * created and started which effectively translates into servicing a newly arrived client
            */
            synchronized (rs) {
```

```
                try {
                    rs.wait();
                    System.out.println("\t· Server has been notified client " + t.getName() + " has
established a new connection");
                    System.out.flush();
                } catch (InterruptedException ex) {
                    Logger.getLogger(MultiThreadedServer.class.getName()).log(Level.SEVERE, null, ex);
                }
            }

        }

    }

}
```

```
/**
 * ************************************************************************
 * Universidad de León, EIII Grado en Ingeniería Informática Asignatura de
 * Arquitectura de Sistemas Distribuidos (C) 2012 José María Foces Morán,
 * instructor.
 *
 * CommandDispatcher.java
 *
 * SINGLE RESPONSIBILITY OF THIS CLASS:
 * Receive commands (file server protocol commands) and invoke the specific
 * protocol-implementation method (upload, download, list, etc).
 *
 * Receives the upstream and downstream objects and implements a simple file
 * upload and download protocol. Commands (High level protocol functions) are
 * encapsulated into class methods, they should be moved into Command-pattern
 * command classes, thus, they can more flexibly be reused and their functions
 * more conceptually organized
 *
 *
 ************************************************************************
 */
package servers;

import java.io.*;
import java.util.logging.Level;
import java.util.logging.Logger;

import transfers.Transfers;

public class Controller {

    private final static String COMMAND_DOWNLOAD = "Download byte array REQ";
    private final static String COMMAND_DOWNLOAD_ACK = "Download ACK";
    private final static String COMMAND_DOWNLOAD_NACK = "Download NACK";
    private final static String COMMAND_UPLOAD = "Upload byte array REQ";
    private final static String COMMAND_UPLOAD_ACK = "Upload ACK";
    private final static String COMMAND_UPLOAD_NACK = "Upload NACK";
    private final static String COMMAND_LISTFILES = "Fetch File List REQ";
    private final static String COMMAND_LISTFILES_ACK = "Fetch File List ACK";
    private final static String COMMAND_LISTFILES_NACK = "Fetch File List NACK";
    private final static String COMMAND_FILEXISTS = "File Exists REQ";
    private final static String COMMAND_FILEXISTS_ACK = "File Exists ACK";
    private final static String COMMAND_FILEXISTS_NACK = "File Exists NACK";
    private final static String COMMAND_SUCCESSFUL = "Command successful";
    private final static String COMMAND_NOT_SUCCESSFUL = "Command not successful";


    /**
     * Receives the first string in a C/S interaction and interprets it as a
     * protocol command and calls a command handler method which will proceed
     * with the sequence of RR that will ultimately complete the command
     */
    void dispatch(ObjectInputStream input,
            ObjectOutputStream output,
            File fileStorageBaseDir) {

        String command = Transfers.protoReceiveString(input);

        /*
         * Determine which command has been received
         */
        if (command.equalsIgnoreCase(COMMAND_UPLOAD)) {
```

```
            doUpload(input, output, fileStorageBaseDir);

        } else if (command.equalsIgnoreCase(COMMAND_DOWNLOAD)) {

            doDonwload(input, output, fileStorageBaseDir);

        } else if (command.equalsIgnoreCase(COMMAND_LISTFILES)) {

            doListFiles(input, output, fileStorageBaseDir);

        } else if (command.equalsIgnoreCase(COMMAND_FILEXISTS)) {

            doFileExists(input, output, fileStorageBaseDir);

        } else {

            Transfers.handleProtocolError(input, output);

        }

    }


    /**
     * UPLOAD COMMAND HANDLING METHOD
     *
     * Performs the server part of a file upload
     */
    private void doUpload(ObjectInputStream input,
            ObjectOutputStream output,
            File fileStorageBaseDir) {

        byte b[] = null;

        String fileName = Transfers.protoReceiveString(input);

        File file = new File(fileName);

        if (!file.isAbsolute()) {

            long fileSize = Transfers.protoReceiveLong(input);

            Transfers.protoSendString(output, COMMAND_UPLOAD_ACK);

            if (fileSize > 0) {

                b = new byte[(int) fileSize];
                Transfers.protoReceiveBytes(input, b);

            }

            String localFileName = fileStorageBaseDir.toString() +
                    File.separator + fileName;

            if (Transfers.handleLocalFileWrite(localFileName, b) == true) {

                Transfers.protoSendString(output, COMMAND_SUCCESSFUL);

            } else {

                Transfers.protoSendString(output, COMMAND_NOT_SUCCESSFUL);

            }
```

```
        }else{
            Transfers.protoSendString(output, COMMAND_UPLOAD_NACK);
        }

    }

    /**
     * DOWNLOAD COMMAND HANDLING METHOD
     *
     * Performs the server part of a file donwload
     */
    private void doDonwload(ObjectInputStream input,
            ObjectOutputStream output,
            File fileStorageBaseDir) {

        String fName = Transfers.protoReceiveString(input);

        File file = new File(fileStorageBaseDir.toString() + File.separator + fName);

        System.err.println("SERVER: Download file name = " + file.getPath());

        if (file.exists() && file.length() > 0) {

            byte b[];

            Transfers.protoSendString(output, COMMAND_DOWNLOAD_ACK);
            Transfers.protoSendLong(output, file.length());

            b = Transfers.handleLocalFileRead(file.getPath());

            if (Transfers.protoSendBytes(output, b)) {
                Transfers.protoSendString(output, COMMAND_SUCCESSFUL);
            } else {
                Transfers.protoSendString(output, COMMAND_NOT_SUCCESSFUL);
            }

        } else {
            Transfers.protoSendString(output, COMMAND_DOWNLOAD_NACK);
        }


    }

    /**
     * FILE EXISTS COMMAND HANDLING METHOD
     *
     * Performs the server part of a file exists command
     */
    private void doFileExists(ObjectInputStream input,
            ObjectOutputStream output,
            File fileStorageBaseDir) {

        String fileName = Transfers.protoReceiveString(input);

        File testFile = new File(fileName);

        if (testFile.exists()) {
            Transfers.protoSendString(output, COMMAND_FILEXISTS_ACK);
        } else {
            Transfers.protoSendString(output, COMMAND_FILEXISTS_NACK);
        }
```

```
    }

    /**
     * LIST FILES COMMAND HANDLING METHOD
     *
     * Performs the server part of a file listing, it lists the names
     * of the files present in the storage base folder
     */
    private void doListFiles(ObjectInputStream input,
            ObjectOutputStream output,
            File fileStorageBaseDir) {

        File files[] = fileStorageBaseDir.listFiles();

        long nfiles = (long) files.length;

        if (nfiles > 0) {

            Transfers.protoSendString(output, COMMAND_LISTFILES_ACK);
            Transfers.protoSendLong(output, nfiles);

            for (int i = 0; i < files.length; i++) {
                Transfers.protoSendString(output, files[i].getName());
            }

            Transfers.protoSendString(output, COMMAND_SUCCESSFUL);

        } else {
            Transfers.protoSendString(output, COMMAND_LISTFILES_NACK);
            Transfers.protoSendString(output, COMMAND_NOT_SUCCESSFUL);
        }
    }
}
```