Program 4 – a File Uploader (25 points)

Due Date: Monday, Dec. 9 by 11:59pm

Program Submission: submit all the source code (*.java) to Canvas.

Program Description

You will write both the **Client** (GUI) and the **Server** (console program) for a Microsoft Word document **uploader**. The Client will upload MS Word documents (in bytes stream) to the Server using port number **5520.** The Server only handles one Client at a time. Therefore, you will NOT be using Java Thread. That is, you will NOT have a ServerThread class in the Server program. Your Server class must service the upload after it gets the connection and won't service another client until the current one is over. This avoids having to deal with two clients uploading a file with the same name at the same time. Your programs must not crash under any situation. You must try-catch everything and print out appropriate error messages identifying the specific exceptions.

The Client

❖ The Client sends **bytes of data** to the Server in the following sequence:

(1) Begin with the **file name** followed by an ASCII null '**\0**'; the file name must be JUST the name, NOT the full path name.

(2) Next, the **size** of the selected file, as a string, also terminated with an ASCII null '**\0**'.

(3) Finally, the **contents** of the file.

For example, if the file **ABC.doc** were of size **3125** bytes, and the first byte in the file was ASCII 'X', the sequence of the byte stream sent form the Client to the socket output stream would be:

'A' 'B' 'C' '.' 'D' 'O' 'C' '\0' '3' '1' '2' '5' '\0' 'X' ...

Note that these are ASCII bytes, NOT characters, since in Java, a "character" occupies two bytes whereas a "byte" only occupies one byte.

❖ After the Server has received and saved the file, the Server sends back the single ASCII byte: '**@**' and then disconnects from that client.

❖ The Client disconnects when it gets back the '**@**'. If the client gets back a character other than '**@**', it still disconnects but prints an error message.

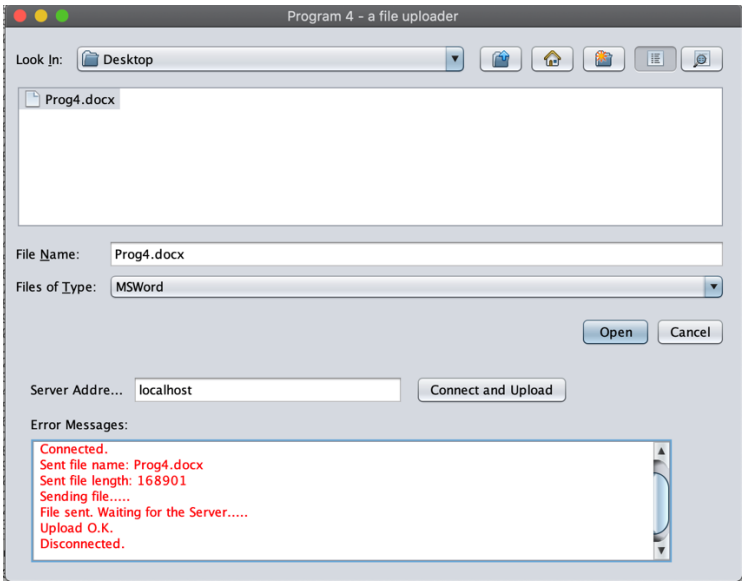
❖ You MUST have a GUI for your client program. For example, if you are using JFrame, you must have the following components:

- A JFileChooser. You can set the fileFilter property to custom code:

```
FileNameExtensionFilter filter = new FileNameExtensionFilter(
                "MSWord", "doc", "docx");
chooser.setFileFilter(filter);
```

- On top of the Client program, import **javax.swing.JFileChooser.*,** or add JFileChooser to the Swing designer.
- Use **getSelectedFile()** method to get the File object, so you know the file name, file size and the file path; **getName(), getPath(),** and **length()** methods of **File** will be very useful.

- A read-only JTextArea and proper JLabel to display all output messages. DO NOT print anything with System.out. Use the append() method of JTextArea to display all output. Be sure to print descriptive messages for ALL steps involving the Client/Server connection and the file transfer.
- Have a "Connect and Upload" JButton; when the user clicks the button, the Client will attempt to connect to the Server and upload the file selected. A sample GUI is shown as follows.



- You can consider having the following methods in the Client program.

```
/**
 * This method takes a String s (either a file name or a file size,) as a
 * parameter, turns String s into a sequence of bytes ( byte[] ) by calling
 * getBytes() method, and sends the sequence of bytes to the Server. A null
 * character '\0' is sent to the Server right after the byte sequence.
 */
private void sendNullTerminatedString(String s) {}

/**
 * This method takes a full-path file name, decomposes the file into smaller
 * chunks (each with 1024 bytes), and sends the chunks one by one to the
 * Server (loop until all bytes are sent.) A null character '\0' is sent to
 * the Server right after the whole file is sent.
 */
private void sendFile(String fullPathFileName){}
```

The Server

❖ The Server must be a stand-alone console program and handle only one client connection at a time. There will only be a Server class - NO ServerThread class.

❖ In the **run()** method of the Server, create a new **ServerSocket** listening on port **5520,** run an infinite loop, waits for a connection, goes through the steps below and save the file in the same folder with your Server program. When done with a client, the Server waits for the next connection.

(1) Get the null-terminated file name
(2) Get the null-terminated file size
(3) Get the file contents
(4) Send the character "@" back to the Client if the transfer is successful

- You MUST have a try-catch inside the infinite loop of the server, so that if an exception is thrown due to a bad exchange with a Client, the Server will recover and wait for the next connection. The server should never crash!
- The Server must print out status messages so all communications can be followed at all times. For example, display "waiting for connection…" when the Server is ready and listening, etc. All messages should be displayed to the standard output (`System.out.println()`.)
- When a connection is made, you can print out when and who, for example:

```
Server running...
Waiting for connection....
Got a connection: Sat Sep 27 15:54:06 CDT 2014
Connected to: /127.0.0.1 Port: 1261
Got file name: prog4.docx
File size: 106527
Got the file.
Waiting for connection....
```

- You will be doing low-level byte I/O - sending and receiving bytes and/or arrays of bytes. This is similar to program 2 in which you transferred files through the HTTP.
- You can consider having the following methods in the Server program:

```
/**
 * This method reads the bytes (terminated by '\0') sent from the Client, one
 * byte at a time and turns the bytes into a String.
 * Set up a loop to repeatedly read bytes until a '\0' is reached.
 */
private String getNullTerminatedString(){}

/**
 * This method takes an output file name and its file size, reads the binary
 * data (in a 1024-byte chunk) sent from the Client, and writes to the output
 * file a chunk at a time.
 * Use the FileOutputStream class to write bytes to a binary file
 * Set up a loop to repeatedly read and write chunks.
 */
private void getFile(String filename, long size){}
```

- I will have my Sever running on constance.cs.rutgers.edu at port 5520 for you to test your Client program before your Server program is done.

Program Grading

Exceptions/Violations	Each Offense	Max Off
Did not implement the client, or the client doesn't run	15	15
Did not implement the server, or the server doesn't run	10	10
Files are not transferred correctly	7	7
Did not use a GUI for the client program	5	5
Client malfunction, or not using 1024-byte chunks, or missing the required GUI components	1	4
Improper exception handling	1	2
Improper message output on Server/Client	1	3