CS 391 – Assignment 5
Due: Thursday, May 9^{th}, 2024 - 8:00 AM
Total points: 100

You must solve this assignment in teams of 2 or 3 students. Make sure to submit only once per team and to list the names of all team members at the top of your submission (see details below).

Important: When working in a team, you should first work on each problem individually and then come together to compare your answers and agree on a final one to include in your submission. Any way you do it, **make sure that each member of the team not only understands the solution fully but will also be able to solve a similar problem by themselves on an exam.**

Preparatory steps to complete IMMEDIATELY

1. Re-read section 3.4.1 in our textbook.
2. Review the slides and your handwritten notes pertaining to socket programming with datagram sockets in Java, including the `setSoTimeout` method in the `java.net.Socket` class.
3. Read up on multi-threading in Java but only enough to be comfortable using the following `Thread` methods: `start`, `sleep`, and `yield`.
4. Download the zip file `a5.zip` from the Canvas course web site.
5. Decompress the zip file to get a directory called `a5` containing five java files and a sub-directory with four images:
   - `A5.java` This utility class is fully implemented and may not be modified. It contains predefined constants/variables as well as one helper method to be used by the other classes.
   - `MyDatagramSocket.java` This class is fully implemented and may not be modified, except for testing purposes, when you are trying to assess the robustness of your reliable data transfer protocol that will run on top of this class.
   - `Client.java` and `Server.java` The classes in which you will implement a simple image file transfer protocol. This protocol will be implemented on top of the `RDT` class described below.
   - `RDT.java` This is the class you will complete to implement a reliable data transfer protocol on top of the unreliable data transfer protocol implemented by the `MyDatagramSocket` class.
   - `images/` This directory contains 4 image files you may use to test your client-server application.
6. Write down your full names at the top of the `RDT.java` file to replace the placeholder in the topmost comment block.

Submission procedure

For this assignment, you will submit a zip file containing exactly three java files, namely `Client.java`, `Server.java`, and `RDT.java`, to the `A5` drop box on Canvas by 7:59 AM on the due date. There is no hard copy due for this assignment.

Points will be deducted if your program crashes, does not meet all of the requirements, and/or if you do not meet the naming/formatting/submisssion requirements listed in this handout. The reason your submitted files and directories must be named EXACTLY as specified in this handout (including letter case and file extension) is that I often use shell scripts to automatically grab and grade your assignment so that I can get them back to you in a timely fashion.

Problem statement

The main goal of this assignment is for you to implement a reliable data transfer (rdt) protocol in the application layer on top of an unreliable data transfer (udt) protocol at the transport layer (e.g., UDP). Note that, while the `DatagramSocket` class implements UDP, it rarely drops or ends up having packets corrupted. To ease your testing, you are given a `MyDatagramSocket` class that sub-classes the built-in `DatagramSocket` class and either drops a packet or else randomly flips one of its bits with an adjustable probability. You will build your RDT class on top of this `MyDatagramSocket` class.

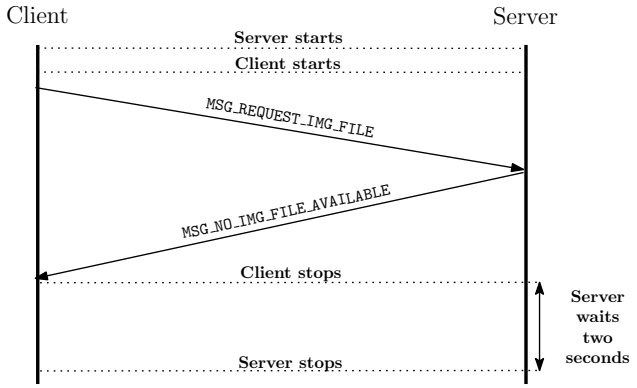
More specifically, your RDT class will implement the `rdt3.0 sender` FSM shown in Figure 3.15 as well as the `rdt2.2 receiver` FSM shown in Figure 3.14 of our textbook, with minor modifications described below. Your RDT implementation must work independently of the applications that use it. In this assignment, you will also implement a simple client-server application that will use your RDT to transfer images. Nevertheless, make sure that your RDT implementation does NOT contain any references to the client or server.

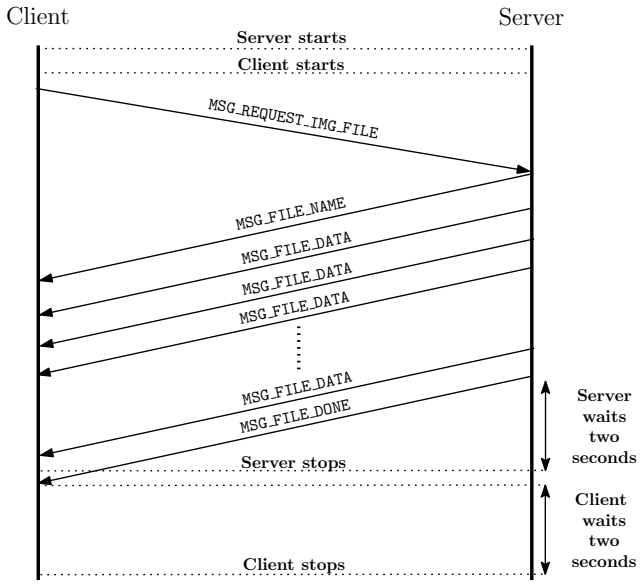
This handout will first describe the protocol used by the client and server in your application. Then, details will be provided on how you must implement the `RDT` class.

Client-server application

Your client will request an image from the server and display it briefly in its own pop-up window (`JFrame`). The protocol used by this application is kept extremely simple so that you can spend most of your time on the RDT implementation. Here is an outline of this protocol:

- Assuming the server is running, when the client comes online, it sends a `MSG_REQUEST_IMG_FILE` message to the server.
- If the server finds no image files to send to the client in its predefined image sub-directory, it sends the client a `MSG_NO_IMG_FILE_AVAILABLE` message and terminates. The client terminates after receiving this message. See the leftmost figure below.
- If the server does have one or more image files to send to the client in its predefined image sub-directory, it picks one randomly and
  1. sends its name in a `MSG_FILE_NAME` message to the client,
  2. then sends the data making up the image file in blocks (or chunks) of 8,192 bytes within `MSG_FILE_DATA` messages, and
  3. finally sends a `MSG_FILE_DONE` message to the client and shuts down. See the rightmost figure below.





Again, your client and server will send these messages using the RDT protocol that you must implement on top of the given `MyDatagramSocket` class. This application-layer protocol assumes that the `RDT.sendData` and `RDT.receiveData` methods it uses are reliable and is not aware of the udt protocol on top of which RDT runs. Before we describe the RDT protocol you must implement, here are the implementation details pertaining to the image transfer, application-layer protocol.

In this application, a message always consists of a one-byte message type header followed by up to 8,192 bytes of data. Application messages will be stored (and sent/received) in byte arrays. The types of application messages are defined in the `A5` class as final values:

1. A `MSG_REQUEST_IMG_FILE` message is sent by the client and contains no data. It starts the session by letting the server know that the client wants an image.
2. A `MSG_FILE_NAME` message is sent by the server and contains the chosen image file's name as its data. Note that this byte array is NOT null-terminated. The client will use this byte array as a string to be displayed in the console.
3. A `MSG_FILE_DATA` sent by the server for each block/chunk of bytes in the image file. Each chunk, and thus the data field in each such message, contains between 1 and 8,192 bytes. Only the last chunk may contain fewer than 8,192 bytes.
4. A `MSG_FILE_DONE` message is sent by the server to indicate that the last chunk of file data was sent. This message contains no data. Both sides must print appropriate messages and then shut down.
5. A `MSG_NO_IMG_FILE_AVAILABLE` message sent by the server contains no data but indicates that the server has no image files in its dedicated sub-directory. Both sides must print appropriate messages and then shut down.

The client and server in this application MUST use the RDT methods, i.e., `sendData`/`receiveData`, for sending/receiving all of their messages. They may NOT contain any reference to `DatagramSocket`, `DatagramPacket`, or related classes. The client and

server must also use new byte arrays of exactly the right size for passing messages to `RDT`. This is particularly important for the very last chunk of data from the image file, which will likely be less than 8,192 bytes of data.

For this part of the assignment, your work will be done in the `Client.java` and `Server.java` files, which are part of the code handout and are partially implemented for you. The UML class diagram shown at the end of this handout specifies the API and private fields of these two classes. Detailed comments inside each class file are included to fully specify each class. Ask questions early if any part of this specification is not clear to your team.

The RDT class

In the `RDT` class, you will implement a slightly modified version of the stop-and-wait rdt protocol that we discussed in class and that is described in full in Section 3.4.1 of our textbook, especially in Figures 3.14 and 3.15.

This class has nested `Runnable` classes, that is, threads, called `Sender` and `Receiver`. These are inner classes so they can access the instance variables in `RDT`. They will implement the finite state machines shown in the textbook figures with the following modifications:

- Unlike in rdt2.2, our receiver will not send an ACK until after the application layer has retrieved the received message. When an uncorrupted packet is correctly received, you should use code like:

  ```
  dataWasReceivedFromBelow = true;
  while ( dataWasReceivedFromBelow )
      Thread.yield();
  ```

  to make the receiver thread wait without blocking the other thread that will be updating the flag. On the other hand, when a corrupted packet is received, the ACK should be sent right away.

- Unlike in rdt3.0, our sender will also start the timer each time a corrupted ACK is received. This is done by setting the "socket timeout" before the receive call with this code:

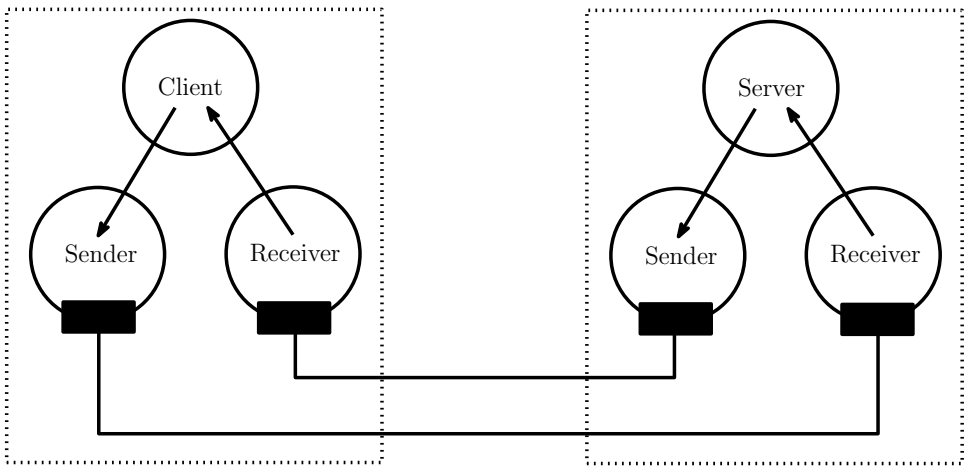
  ```
  senderSocket.setSoTimeout(500);
  ```

  The socket will then throw an exception if and when a time out occurs, in which case the packet must be resent.

When implementing the `RDT` class, you should always keep in mind the following principle:

> Even though both the `Client` and `Server` classes will use this `RDT` class, the `RDT` class may NOT make any reference to the `Client` or `Server` classes. In fact, your class should be usable when building *any* other applications.

Each peer process in the application-layer protocol, that is, the client or the server process, creates an instance of the `RDT` class in order to communicate with its peer process at the other "end". In turn, each RDT instance runs two threads, one for its receiver side and one for its sender side, since both client and server must be able to receive/send messages from/to its peer process. Since the client runs in its own process/thread and so the does the server, a total of six threads are running concurrently in this application, as depicted in the figure below, in which threads are shown as circles and `MyDatagramSocket` instances are represented as black rectangles.



In the previous section, we described the format of the messages sent and received by the client/server in the application layer namely, one header byte followed by zero or more data bytes. These messages constitute the payload for the segments sent between the `RDT` peer instances. We now describe the format of these segments. Each `RDT` segment contains, in this order:

- a one-byte header: only the least significant bit of this header may be used, namely to represent the current value of the sequence number or acknowledgment number, depending on the type of segment being sent.
- its payload, that is, the application message, and

- a one-byte checksum, computed as the bitwise XOR of the bytes in the header and application message that precede it.

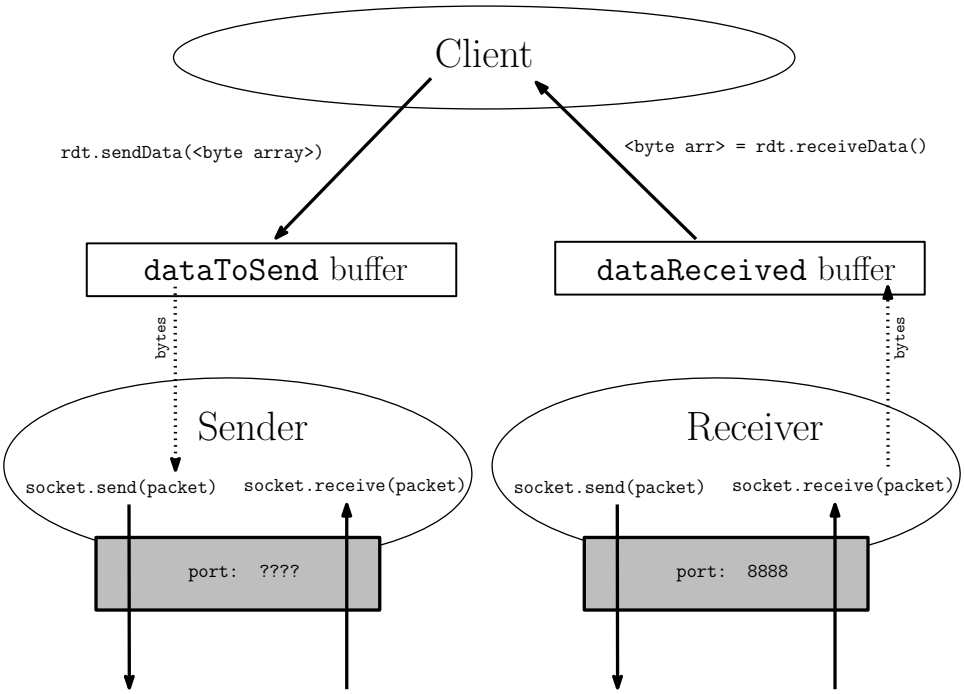
When an `RDT` receiver gets a segment, it computes the checksum over the entire segment, including the three components listed above, and should get a 0 byte value if the segment was reliably transferred from the other host to this one. Any non-zero checksum value guarantees that exactly one bit was flipped. Make sure you understand why this is the case.

Because of the RDT segment format, when an uncorrupted segment is received, only the application message should be sent up to the client or server. The latter processes should never see the header or checksum bytes. Likewise, when the client or server pass data down to the RDT layer, the header and checksum bytes get added by RDT's sender. But the fact that the application message contains its own one-byte header makes no difference to RDT, which is not aware of it and just treats it as part of its payload.

We now focus on the communication between the client/server threads and the `RDT` threads. We only discuss the client side here, using the figure below, since the server side is similar. Each `RDT` instance maintains two buffers: one contains the bytes the client wants to send to the server, the other contains the bytes the client receives from the server. The labels on the arrows in the figure below indicate the method calls that move the bytes around. See also the UML class diagram at the end of this handout for the full API of the classes involved.

The `dataToSend` buffer is a storage area for the bytes that the client wants to send. This buffer is needed since the Sender thread may not be ready to send the data at the time the Client calls the `rdt.sendData` method. This is because the RDT instance may still be waiting for an ACK for the previous message before it can send the next one. Remember that you are implementing a stop-and-wait protocol.

Similarly, the `dataReceived` buffer is a storage area for the bytes that were last received from the server. This buffer is needed since the `Receiver` thread may have to wait for the application (in this case the Client) to retrieve those bytes by calling the `rdt.receiveData` method.



Note that the `Receiver`'s socket uses a fixed well-known port number defined in the `A5` class, whereas the `Sender`'s socket uses an ephemeral port number. The same is true in the server. Make sure to understand the reason for this implementation feature, as well as the concept of port inversion, which your code must use.

The UML diagram included at the end of this handout contains a detailed specification of all of the classes involved in this project, except for the `A5` class. This diagram, together with the detailed comments included in the code handout and the contents of this document should be enough for you to complete this assignment. However, you are always welcome to ask questions when anything is not clear to you or if you think there is an error or omission in the handout.

The four topmost classes in the UML diagram are only partially implemented in the code handout. Your main task is to complete them all. You must provide all of the missing method bodies and are not allowed to modify the given code. Furthermore, you are not allowed to add any new methods or instance/class variable/constants. Nor are you allowed to add any new classes, nested or otherwise. If you are not sure whether something is allowed, ask before it is too late.

As already mentioned, the `MyDatagramSocket` is fully implemented and you will not be including it in your submission. Nevertheless, you must understand it fully. In fact, you may want to modify (or sub-class) it in order to perform thorough testing of your code

before submission. I recommend you change some its behaviors in order to test all aspects of the implemented protocol. For example, you may change one of its parameter values or temporarily disable one of its functions (i.e., packet drop, bit flip) in order to perform a kind of unit testing of your protocol targeting particular paths in the FSMs.

In fact, since testing this multi-threaded program that also uses random-number generators could take a significant amount of time if you rush through the analysis/design process (don't!), the next section describes additional techniques and requirements that will make it easier for you to test your code and for me to grade it and assign partial credit in case I detect some incorrect behaviors.

Running and testing your code

The Client and Server programs will run in their own process, keeping in mind that the Server must already be running when the Client is launched. Since you will run your tests in localhost mode, with both processes running on the same computer, you will not need to specify any IP address on the command line. Therefore, one way to test your code is to type `java Server` in one terminal window, and then `java Client` in another terminal window.

As described above, each process will run three threads, and each thread is required to output to the console a detailed message at each step of its processing, as shown in the following traces.

For full (or even partial) credit, your code is required to match the format of these messages precisely, as described here. These messages will also help you when debugging your code. For example, each output line must start with an uppercase keyword that specifies which thread is producing this output, namely `SERVER`, `CLIENT`, `SENDER`, or `RECEIVER`, followed by a single space, and then a clear and detailed description of the step that was just performed. In the case of udt "errors", the error messages are already written for you inside the `MyDatagramSocket` class. They all start with three blank spaces followed by three asterisks, so that udt errors stand out in the output.

The required format of all other console messages is implicitly defined by its inclusion in one of the full traces that follow. For example, when the Server is launched and the provided image file `black.png` is selected and the udt transport protocol (by chance or otherwise) generates no packet errors, the two traces shown on the next page must be produced.

Client process

```
> java Server
SENDER started
RECEIVER started
SERVER started
RECEIVER got data from below
RECEIVER got good data: send new ACK
SERVER looping in getRequest
SERVER got request for image file
SENDER got data from above
SERVER sent file name "black.png"
SENDER sent packet
RECEIVER sent ACK: 0
SENDER got GOOD ACK: 0
SENDER got data from above
SERVER sent last file chunk #1 [4176 bytes starting with 0x89]
SERVER done sending the file black.png
SENDER sent packet
SENDER got GOOD ACK: 1
SENDER got data from above
SERVER sent file done message
SERVER done
SENDER sent packet
SENDER got GOOD ACK: 0
SERVER shutting down
```

Server process

```
> java Client
SENDER started
RECEIVER started
CLIENT started
CLIENT sent file request
SENDER got data from above
SENDER sent packet
RECEIVER got data from below
RECEIVER got good data: send new ACK
CLIENT got file name: black.png
SENDER got GOOD ACK: 0
RECEIVER sent ACK: 0
RECEIVER got data from below
RECEIVER got good data: send new ACK
CLIENT got file chunk #1 [4176 bytes starting with 0x89]
RECEIVER sent ACK: 1
RECEIVER got data from below
RECEIVER got good data: send new ACK
RECEIVER sent ACK: 0
CLIENT done
CLIENT shutting down
```

Make sure to spend time understanding the source, meaning, and ordering of these outputs. In this case, the image file is small enough that its bytes all fit in a single data chunk, which is both its first and last chunk. Also, as mentioned above, I picked this first test case because it does not contain any packets being lost or corrupted. Note that in addition to the console output, the image was displayed by the client for two seconds in a pop-up window before the client shut down.

Despite its extreme simplicity, even this small test case may be non-trivial to debug because the output is pretty detailed, and crucially, the interleaving of messages from the Server and Client, as well as the output of their `RDT` threads, is not clearly visible, since the two processes were run in different terminal windows. To mitigate this difficulty, it is possible to run both processes in the background from the same terminal window, by typing:

`java Server & java Client &`

The resulting console output is as follows:

```
> java Server & java Client &
SENDER started
RECEIVER started
SENDER started
RECEIVER started
SERVER started
CLIENT started
CLIENT sent file request
SENDER got data from above
SENDER sent packet
RECEIVER got data from below
RECEIVER got good data: send new ACK
SERVER looping in getRequest
SERVER got request for image file
SENDER got data from above
SERVER sent file name "black.png"
SENDER sent packet
RECEIVER got data from below
RECEIVER got good data: send new ACK
CLIENT got file name: black.png
RECEIVER sent ACK: 0
SENDER got GOOD ACK: 0
RECEIVER sent ACK: 0
SENDER got GOOD ACK: 0
SENDER got data from above
SERVER sent last file chunk #1 [4176 bytes starting with 0x89]
SERVER done sending the file black.png
SENDER sent packet
RECEIVER got data from below
RECEIVER got good data: send new ACK
CLIENT got file chunk #1 [4176 bytes starting with 0x89]
RECEIVER sent ACK: 1
SENDER got GOOD ACK: 1
SENDER got data from above
SERVER sent file done message
SERVER done
SENDER sent packet
RECEIVER got data from below
RECEIVER got good data: send new ACK
RECEIVER sent ACK: 0
SENDER got GOOD ACK: 0
CLIENT done
SERVER shutting down
CLIENT shutting down
```

Unfortunately, in this case, it is not clear which of the lines starting with SENDER or RECEIVER were produced by the server and which ones were produced by the Client.

To remedy this problem, you may use the `A5.print` utility method that takes in a message and sends it to the console, but with two added features. First, it takes another (first) argument that serves as a flag. When this flag is equal to the string `''S''`, then the message is preceded by a fixed amount of white space before it is sent to the console. Therefore, if you call this method with the `''S''` tag each time the server side needs to produce console output and with an empty tag/string each time the client side needs to produce console output, the server's output will be clearly identified by appearing as if in a second column, whereas the client-side output will be printed starting at the left side of the window.

While this approach is trivial to implement when the messages are produced directly in the Client or Server processes, the Sender/Receiver threads are running in the RDT instance and are not aware of the application process that is using them. **We definitely want to keep this feature.** To get around this issue, I added a tag field to each `RDT` instance so that it only gets tagged with an `''S''` when it is running on the server side (at the time it is created in the Server constructor). **Again, you are not allowed to use this `RDT` tag field in any part of your code except in a call to `A5.print` and for debugging purposes.** Other uses would break the separation of concerns principle applied to the boundary between the application and transport layers and would be heavily penalized.

The second feature added by the `A5.print` utility method is that each console message is preceded by a time stamp (seconds and milliseconds) that help keep track of the timing of events. In the same test case discussed above, the trace produced when using this utility method for all console output is as follows.

```
> java Server & java Client &
                          [66.141] RECEIVER started
                          [66.141] SENDER started
[66.142] RECEIVER started
[66.142] SENDER started
                          [66.192] SERVER started
[66.197] CLIENT started
[66.198] CLIENT sent file request
[66.198] SENDER got data from above
[66.303] SENDER sent packet
                          [66.305] RECEIVER got data from below
                          [66.305] SERVER looping in getRequest
                          [66.305] SERVER got request for image file
                          [66.305] RECEIVER got good data: send new ACK
                          [66.306] SENDER got data from above
                          [66.310] SERVER sent file name "black.png"
                          [66.407] SENDER sent packet
[66.408] RECEIVER got data from below
                          [66.411] RECEIVER sent ACK: 0
[66.412] SENDER got GOOD ACK: 0
[66.413] RECEIVER got good data: send new ACK
[66.414] CLIENT got file name: black.png
[66.514] RECEIVER sent ACK: 0
                          [66.515] SENDER got GOOD ACK: 0
                          [66.515] SENDER got data from above
                          [66.519] SERVER sent last file chunk #1 [4176 bytes starting with 0x89]
                          [66.519] SERVER done sending the file black.png
                          [66.616] SENDER sent packet
[66.616] RECEIVER got data from below
[66.616] RECEIVER got good data: send new ACK
[66.620] CLIENT got file chunk #1 [4176 bytes starting with 0x89]
[66.720] RECEIVER sent ACK: 1
                          [66.721] SENDER got GOOD ACK: 1
                          [66.721] SENDER got data from above
                          [66.721] SERVER sent file done message
                          [66.721] SERVER done
                          [66.821] SENDER sent packet
[66.822] RECEIVER got data from below
[66.822] RECEIVER got good data: send new ACK
[66.923] RECEIVER sent ACK: 0
                          [66.923] SENDER got GOOD ACK: 0
[67.983] CLIENT done
                          [68.722] SERVER shutting down
[69.987] CLIENT shutting down
```

Make sure you understand this trace fully, especially the interleaving of console messages coming from the Server versus the Client (based on the indentation), but also the interleaving of messages coming from different threads within the Client (or the Server) and why the time stamps do not always appear to be in chronological order.

The next two pages show additional traces using the same image file with either some bit flips or a dropped packet. Of course, larger image files, like the other two included in the code handout, will constitute more stringent test cases, but will also yield much longer traces.

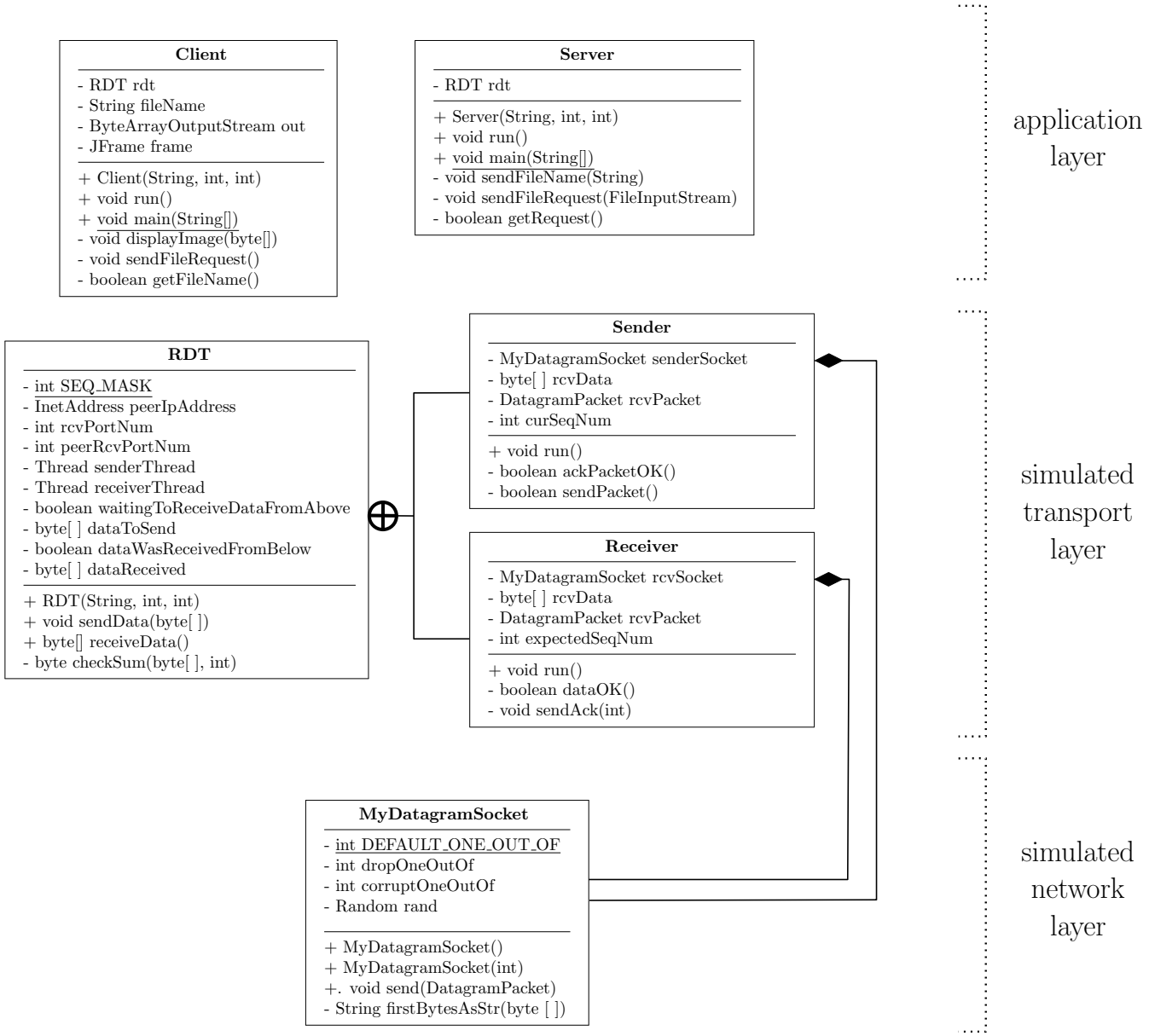
Have fun and good luck!

As another example, in the following trace, we observe two random bit flips.

```
                        [33.826] SENDER started
                        [33.826] RECEIVER started
[33.830] SENDER started
[33.830] RECEIVER started
                        [33.897] SERVER started
[33.900] CLIENT started
[33.900] CLIENT sent file request
[33.900] SENDER got data from above
[34.001] SENDER sent packet
                        [34.003] RECEIVER got data from below
                        [34.003] RECEIVER got good data: send new ACK
                        [34.003] SERVER looping in getRequest
                        [34.003] SERVER got request for image file
                        [34.004] SENDER got data from above
                        [34.008] SERVER sent file name "black.png"
                        [34.105] SENDER sent packet
[34.106] RECEIVER got data from below
[34.108] CLIENT got file name: black.png
                        [34.109] RECEIVER sent ACK: 0
[34.109] SENDER got GOOD ACK: 0
[34.116] RECEIVER got good data: send new ACK
[34.217] RECEIVER sent ACK: 0
                        [34.218] SENDER got GOOD ACK: 0
                        [34.219] SENDER got data from above
                        [34.222] SERVER sent last file chunk #1 [4176 bytes starting with 0x89]
                        [34.223] SERVER done sending the file black.png
                        [34.322] SENDER sent packet
[34.322] RECEIVER got data from below
[34.323] RECEIVER got good data: send new ACK
[34.326] CLIENT got file chunk #1 [4176 bytes starting with 0x89]
[34.423] RECEIVER sent ACK: 1
                        [34.423] SENDER got GOOD ACK: 1
                        [34.423] SENDER got data from above
                        [34.423] SERVER sent file done message
                        [34.424] SERVER done
   *** CORRUPTED packet with first bytes = 0x01 0x04 0x04
                        [34.527] SENDER sent packet
[34.527] RECEIVER got data from below
[34.528] RECEIVER checksum error on packet 0
[34.528] RECEIVER go bad data: resend previous ACK
[34.630] RECEIVER sent ACK: 1
                        [34.630] SENDER bad ACK seq num; expected: 0
                        [35.131] SENDER timed out waiting for ACK, resending packet
[35.234] RECEIVER got data from below
                        [35.234] SENDER resent packet
[35.234] RECEIVER got good data: send new ACK
   *** CORRUPTED packet with first bytes = 0x00 0x40
[35.339] RECEIVER sent ACK: 0
                        [35.339] SENDER bad ACK checksum: 0
[35.804] CLIENT done
                        [35.840] SENDER timed out waiting for ACK, resending packet
                        [35.942] SENDER resent packet
[35.943] RECEIVER got data from below
[35.943] RECEIVER wrong sequence number, expected: 1
[35.943] RECEIVER go bad data: resend previous ACK
[36.050] RECEIVER sent ACK: 0
                        [36.050] SENDER got GOOD ACK: 0
                        [36.427] SERVER shutting down
[37.808] CLIENT shutting down
```

As a final example, in the following trace, we observe one randomly dropped packet.

```
                        [40.550] RECEIVER started
                        [40.550] SENDER started
[40.554] RECEIVER started
[40.554] SENDER started
                        [40.621] SERVER started
[40.630] CLIENT started
[40.630] CLIENT sent file request
[40.630] SENDER got data from above
[40.734] SENDER sent packet
                        [40.736] RECEIVER got data from below
                        [40.736] SERVER looping in getRequest
                        [40.736] RECEIVER got good data: send new ACK
                        [40.736] SERVER got request for image file
                        [40.737] SENDER got data from above
                        [40.741] SERVER sent file name "black.png"
                        [40.841] SENDER sent packet
                        [40.842] RECEIVER sent ACK: 0
[40.843] RECEIVER got data from below
[40.843] SENDER got GOOD ACK: 0
[40.844] RECEIVER got good data: send new ACK
[40.845] CLIENT got file name: black.png
[40.945] RECEIVER sent ACK: 0
                        [40.946] SENDER got GOOD ACK: 0
                        [40.946] SENDER got data from above
   *** DROPPED packet with first bytes 0x01 0x03 0x89
                        [40.947] SENDER sent packet
                        [40.949] SERVER sent last file chunk #1 [4176 bytes starting with 0x89]
                        [40.950] SERVER done sending the file black.png
                        [41.448] SENDER timed out waiting for ACK, resending packet
                        [41.551] SENDER resent packet
[41.551] RECEIVER got data from below
[41.551] RECEIVER got good data: send new ACK
[41.555] CLIENT got file chunk #1 [4176 bytes starting with 0x89]
[41.652] RECEIVER sent ACK: 1
                        [41.652] SENDER got GOOD ACK: 1
                        [41.652] SENDER got data from above
                        [41.652] SERVER sent file done message
                        [41.652] SERVER done
                        [41.757] SENDER sent packet
[41.757] RECEIVER got data from below
[41.758] RECEIVER got good data: send new ACK
[41.862] RECEIVER sent ACK: 0
                        [41.862] SENDER got GOOD ACK: 0
[42.343] CLIENT done
                        [43.655] SERVER shutting down
[44.348] CLIENT shutting down
```


Client
- RDT rdt
- String fileName
- ByteArrayOutputStream out
- JFrame frame
+ Client(String, int, int)
+ void run()
+ void main(String[])
- void displayImage(byte[])
- void sendFileRequest()
- boolean getFileName()
Server
- RDT rdt
+ Server(String, int, int)
+ void run()
+ void main(String[])
- void sendFileName(String)
- void sendFileRequest(FileInputStream)
- boolean getRequest()
application layer
RDT
- int SEQ_MASK
- InetAddress peerIpAddress
- int rcvPortNum
- int peerRcvPortNum
- Thread senderThread
- Thread receiverThread
- boolean waitingToReceiveDataFromAbove
- byte[ ] dataToSend
- boolean dataWasReceivedFromBelow
- byte[ ] dataReceived
+ RDT(String, int, int)
+ void sendData(byte[ ])
+ byte[] receiveData()
- byte checkSum(byte[ ], int)
Sender
- MyDatagramSocket senderSocket
- byte[ ] rcvData
- DatagramPacket rcvPacket
- int curSeqNum
+ void run()
- boolean ackPacketOK()
- boolean sendPacket()
Receiver
- MyDatagramSocket rcvSocket
- byte[ ] rcvData
- DatagramPacket rcvPacket
- int expectedSeqNum
+ void run()
- boolean dataOK()
- void sendAck(int)
simulated transport layer
MyDatagramSocket
- int DEFAULT_ONE_OUT_OF
- int dropOneOutOf
- int corruptOneOutOf
- Random rand
+ MyDatagramSocket()
+ MyDatagramSocket(int)
+. void send(DatagramPacket)
- String firstBytesAsStr(byte [ ])
simulated network layer