Networking | **Programming Project 3**
CSC3935 Spring 2024 (Kissel) | **Due Date: Friday April 5, 2024**

! This is a three-person group project.

We have spent the last two units discussing reliable transmission protocols. In this project you will implement the RDT 3.0 protocol, as discussed in the recommended textbook. This is a Stop-and-Wait ARQ protocol. You will not need a copy of this textbook as the state machines will be provided in this document. In order to test out your RDT implementation you will build a small peer-to-peer file sharing service, Mackshare. Due to the size and complexity of this project you will be working in a group of *three* for approximately three weeks. This project is challenging and rewarding so you should plan to start *immediately*. To help you manage your time, I have broken the project up into milestones with recommended (not required) completion dates. I have also provided some starter code for you, this code should not be modified.

Phase I: Implement RDT 3.0

Recommended Completion Date: Wednesday March 20, 2024

In this phase you will implement the reliable data transmission protocol on top of UDP. The protocol implementation consists of four classes.

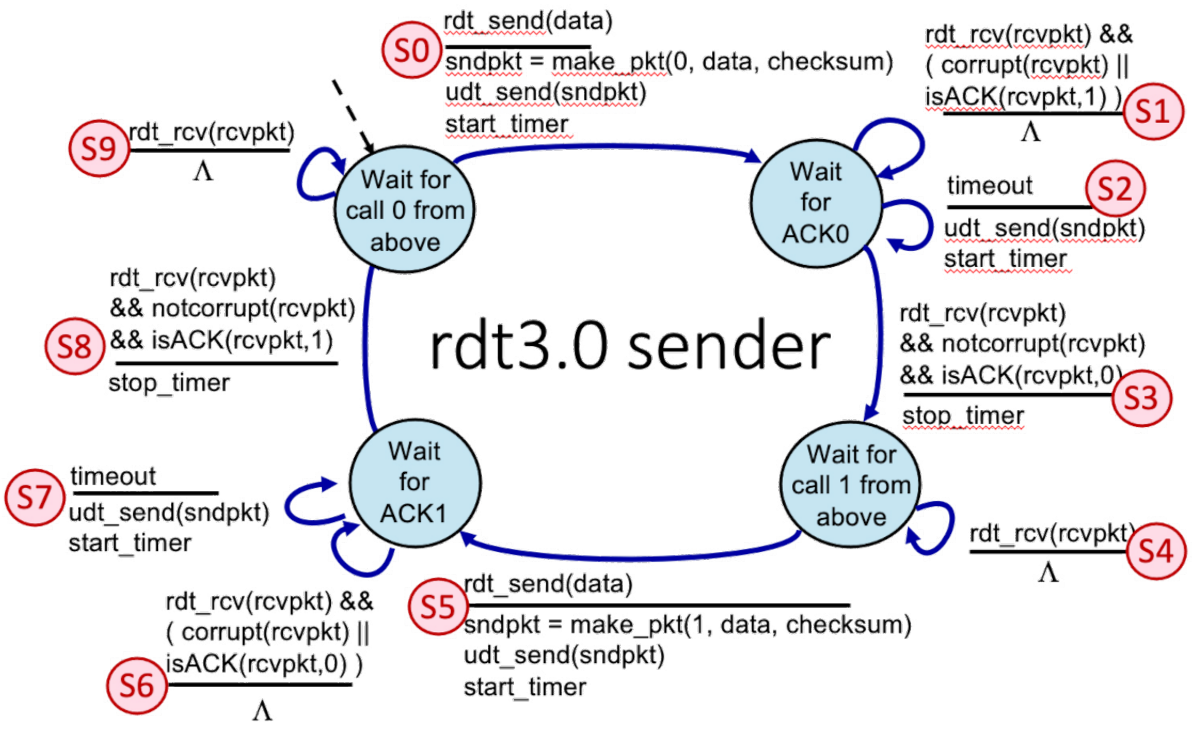
1. `Message` this represent the applications view of a `ReliableSocket` message, it is *not* what is sent over the wire. It consists of the following fields:
   - An `InetAddress` representing the destination address.
   - A `int` representing the destination port.
   - A `byte[]` representing the data to be sent.
2. `ReliablePacket` this is the binary representation of the RDT message it consists of a nine-byte header and a maximum of 500 bytes for the payload. The format of the header is:
   (a) A two-byte sequence number which for our protocol is either 0 or 1.
   (b) A two-byte payload length field which is a count of the number of bytes in the payload.
   (c) A one-byte flag field which is `0x01` if the message is an acknowledgment and `0x00` otherwise.

(d) A four-byte checksum field which is the output of a CRC-32 checksum (this is available in `java.util.zip.CRC32`). This checksum is computed over the first five bytes of the header and the bytes of the payload.

You may find that using the builder pattern is helpful in constructing this object. Additionally, you may find it helpful to define two constants one to represent the header size of 9 bytes, and one to represent the maximum payload size, 500 bytes.

3. `ReliableSocket` this is the API for the protocol and closely mirrors the `DatagramSocket` API. It has two types of constructors and three methods.

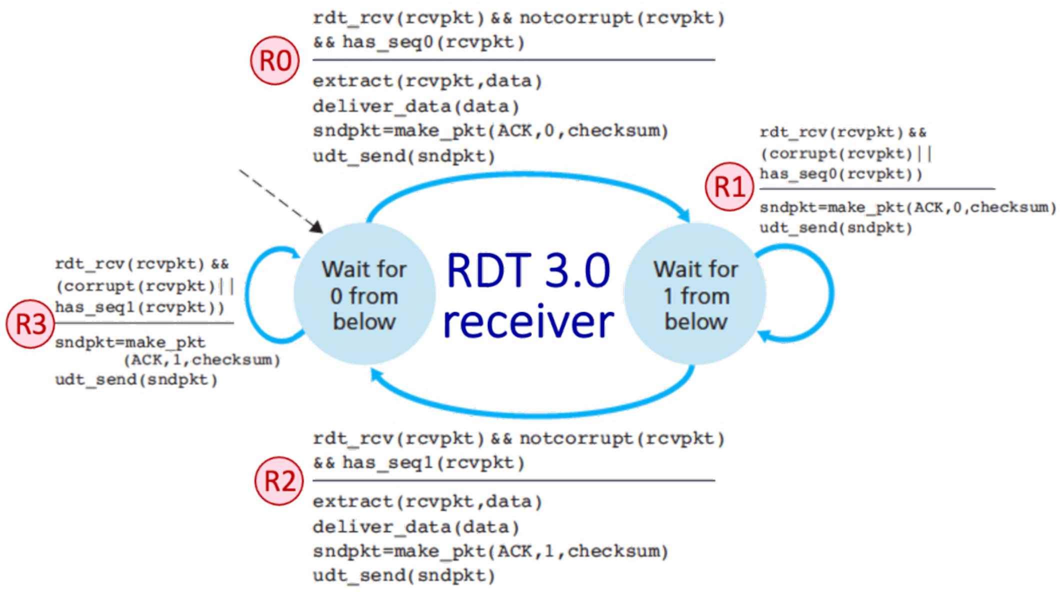
- A constructor that takes the `InetAddress` of another machine as well as its `port`. The constructor connects to this remote machine using the `DatagramSocket`'s `connect` method.
- A constructor that takes a local `port` and constructs a new `DatagramSocket`, using the `DatagramSocket` constructor that takes a port.
- A `send` method that take a `Message` object, constructs a new `ReliablePacket` and sends that packet over the UDP socket. It is important that this method follows the sender finite state machine for RDT 3.0 as given below:



Citation: `https://gaia.cs.umass.edu/kurose_ross/interactive/rdt_30_sender.png`

You will of course want to think about `rdt_send` is your send and `rdt_receive` should be your UDP socket receive. The `notCorrupt` and `isAck` methods should be part of your `ReliablePacket` class. Remember that Λ indicates no action should be taken. Any timer work will be handled using the `PacketTimeoutHandler` object described below. Send is a blocking call which means we won't return from the call to `send` until an ACK has been received.

- A `receive` method that returns a `Message` object to the caller. This is a blocking call that receives a message from the UDP socket and follows the state machine given below.



R0
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
R1
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq0(rcvpkt))
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
Wait for 0 from below
RDT 3.0 receiver
Wait for 1 from below
R3
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq1(rcvpkt))
sndpkt=make_pkt (ACK,1,checksum)
udt_send(sndpkt)
R2
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)

Citation: `https://gaia.cs.umass.edu/kurose_ross/interactive/rdt_30_receiver.png`

Any `deliver_data` call should be treated a `return`.

- A `close` method that stop all timers associated with the `ReliableSocket` and then closes the `DatagramSocket`.

When implementing the `ReliableSocket` it is important that the current state should be maintained on a connection basis. For our purposes a connection identifier should be used. This identifier is a four-tuple that consists of the senders address, the senders port, the destination address, and the destination port. I managed all state information for a connection using a `HashMap` where the key is the connection identifier. Specifically, I had one map for the receiver state, one map for the sender state, and one map to maintain the per connection timer.

For this class I have provided you two methods `udtSend` and `udtReceive` that handle the unreliable connection to the UDP channel. You will also notice that I have instrumented some code that allows a lossy channel to be simulated for testing. This is important as UDP on your machine through the loopback interface is reliable. Setting the failure probability to zero will turn off the introduction of failures, setting the failure probability to one will essentially make the protocol useless as no send will ever occur.

   Any socket UDP socket you create should have its timeout set to 1000 milliseconds using the `setSoTimeout` method of the `DatagramSocket` class.

4. `PacketTimeoutHandler` is a class that implements the `Runnable` interface. The run method simply retransmits a previously transmitted `DatagramPacket`. This object has been provided for you in the stub. You will not have to modify it. To use this object you will have to read up on the `Timer` class. The `Timer` class essentially schedules a thread to run sometime in the future (see the `schedule`) method. To stop (also called cancel) the timer you call that `cancel` method of the `Timer` class. Once a `Timer` object has been canceled it *can not* be used to schedule future timeouts, so you should make a new `Timer` object for the connection after every call to `cancel`. All timeouts should be set to 1000 milliseconds.

Phase II: Peer-to-Peer File Sharing Implementation

Recommended Completion Date: Wednesday April 3, 2024

As we discussed in unit 3, peer-to-peer file sharing is a design technique where multiple computers form a loose peer group (BitTorrent calls these a Torrent) and this group can share files with one another. There is no centralized file server. In the most basic form every peer has access to a "seed" file that describes what files are available in the system and what peers have those files. When a peer wishes to download a file from the peer group, they don't download the entire file from one peer but instead download file chunks from several different peers that have the file (these hosts are found in the seed file). Once all of the chunks are received the file is reconstructed and stored on disk.

For our implementation we will assume that each peer already has the seed file and a directory containing all files they can share. The actual request of a file is handled in the `Driver.java` file of the stub. While you are encouraged to read the code found in `Driver.java` you will not be required to make any additions or modifications. An example seed file is provided as `seeds.json` in the stub. The file name field represents the name of the file (as found in its file directory), the hosts are `Host` objects that represent machines that have a copy of the file, and chunks is a field that represents how many *50 byte* chunks the file consists of. The protocol itself is a plaintext protocol that uses JSON marshaling of objects.

Your task for this phase is to implement the file share protocol (in the `fileshare` package). This package consists of four objects:

1. A `ChunkRequest` object that represents a request for a chunk of a file. This object should implement `JSONSerializable`. The JSON representation has the following fields:

- A field with key `"file-name"` whose value is a string that represents the file name to get the chunk of.
- A field with key `"chunk-id"` whose value is an integer that represents the chunk to retrieve from the file.

The class should have any accessors and constructors you deem necessary.

2. A `ChunkResponse` object that represents a response to a chunk request. This object should implement `JSONSerializable`. The JSON representation has the following fields:

   - A field with key `"file-name"` whose value is a string that represents the file name to get the chunk of.
   - A field with key `"chunk-id"` whose value is an integer that represents the chunk to retrieve from the file
   - A field with key `"chunk-data"` whose value is a string that represents the `Base64` encoded form of the 50-byte chunk of the file.
     - You will want to read up on `java.util.Base64` to understand how encode and decode `byte` arrays.

   The class should have any accessors and constructors you deem necessary.

3. A `FileShareThread` which implements the `Runnable` interface. The `run` method should be an infinite loop that reads a chunk request from the `ReliableSocket` and responds with a chunk response. The constructor should take as input a `Configuration` object as passed from the `Driver.java` and construct the appropriate reliable socket. It should be noted that the `Configuration` object also contains the location of the directory containing the files to share.

   To get a chunk of a file I would utilize the `FileInputStream` objects `skip` and `read` method to get the chunk you want. The `skip` method skips the first n bytes of the file when given n as input, leaving the file cursor at byte n. There is a form of `read` that takes a an array of `byte`s and fills that array with bytes from the file starting at the current cursor position.

4. A `DownloadThread` which implements the `Runnable` interface. The constructor should take the `Host` object representing the host to download from, and `ArrayList` of `Integers` that represents the chunks that should be downloaded from the host, and the name of the file to download. The `run` method should loop over the `ArrayList` of chunks to download, and download all of the chunks. The received chunks should be placed in a `HashMap` with a key that holds the chunk id (an `Integer`) and a value that is an array of bytes (i.e. `byte[]`). Once all of the chunks have been downloaded the `run` method should end. Please do *not* have the run method loop forever. I have provided you with a

You have been provided a stub of some of the Java files for the filehsare package so that they integrate correctly with the main driver class.

Helpful Hints

I offer the following helpful hints as you embark on the project:

- Networking code is notorious for getting complicated fast, be sure to abstract and modularize your code for ease of debugging.
- Employ good design principles.
- Use Wireshark to aid you in debugging your code. Since you are designing your own binary protocol that runs on top of UDP you should set your filters to UDP in my testing some of my packets were flagged as TAPA packets (a decoding error). This is because Wireshark thinks they look different. You can just turn off TAPA decoding to look at these packets as all UDP. Also to keep your self sane, you should only run Wireshark on the localhost/loopback interface. This is a much quieter interface to debug with.
- Protocols are beautiful examples of state machine based programming. A *state machine* is a directed graph where each node is called a state and each edge is represented with an event that if that event is performed, allows the machine to transition to a new state. If you have had CSC 3320 Operating Systems, CSC 3555 Computability & Complexity, or CSC 3910 Graphics you have seen state machines. To implement a state machine, just add a variable, `state`, that keeps track of the current state. You should use a `switch` statement to perform the action of the the current state and then go to the next state by updating the `state` variable. Please use the state machines as discussed in the RDT protocol.
- Only one `DatagramSocket` can be bound to a port at a time. Remember these port numbers should be on the high end of the numbers (anything 1024 or below is reserved, and probably used by your machine) It is possible that a port can be hung during development. You can correct this by resetting your network card (or just restarting your computer).
- All JSON objects should have a corresponding Java class that implements the `JSONSerializable` interface.
- The Java docs are your friend when working with new classes such as `DatagramPacket`, `DatagramSocket`, and `ByteBuffer`, `Timer`, and `Base64`.
- You should *not* have to disable your firewall to work on this project. Again, **do not disable your firewall**.

- I've mentioned it earlier but, **start early** and work as a team.
- If you follow the schedule you will notice time has be set aside for testing purposes. As a general rule I would make sure that you test the RDT protocol to ensure it works correctly before moving on to the fileshare protocol. This way you are sure the the only problems you will likely see are in the fileshare protocol.
- In order to get the JSON support you need to download or clone `merrimackutil` from `https://github.com/kisselz/merrimackutil`

Testing

I have provided you a limited amount of testing files in the `example-configs` directory of the stub. Please feel free to use this for testing or extend the files on your own. By default, the example test files turn off the lossy channel simulation, so if you wish to experiment with lossy channels you will need to modify the configuration files.

There is one file that is available to download by the application using `config3.json` called `cs.jpg` a "Map of Computer Science". You can add your own files but you will have to determine how many 50 byte chunks they consist of. To do this take the file size in bytes, divide it by 50 and take the ceiling.

I would test the RDT protocol separate using some small test programs that way you are sure your RDT protocol is working correctly before you worry about checking the file sharing protocol.

Examples

The P2P application ahs the following options.

```
$ java -jar dist/p2p.jar -h
usage:
  p2p --config <config>
  p2p --help
options:
  -c, --config Config file to use.
  -h, --help Display the help.
```

The application when started with present you with a command line interface (CLI). This interface will allow you to request files by typing the name of the file to download. The files will be downloaded to the current working directory. If you wish to quit the application you do so by write `.quit`. Please note the leading dot.

Source Control

In all projects in this class you will use source control. The source control system the department has chosen is `git`. In particular we will use GitHub. For this project please create a *private* GitHub repository under and account linked to your Merrimack e-mail.

As part of your grade on this project I will be requiring good use of the source control. This means your project should have several commits over the course of time.

STOP Don't develop your entire program outside of source control and then add it to source control before submission!

Every commit should have a well worded commit message that explains what the update to the repository is all about. Please be sure to not put automatically generated files in the repository. This includes `class` files, `jar` files, and `zip` files.

Since this a group project each members contributions will be judged based on the commit history. For example, a group member that has one tiny commit will receive a lower grade than other members. To make your submission clear, please have a `README` in your repository that lists all of the members of the group.

Submitting

To submit this program, please add me as a contributor to your private GitHub repository, my username is (`kisselz@merrimack.edu`). Since this is a group project, there should only be one repository which each group member contributes to.

Rubric

Your grade on this project will be determined as follows:

- Good use of source control (10 points).
- Code is well architected (10 points).
- Protocols are interoperable (10 points).
- RDT 3.0 Send correct (20 points).
- RDT 3.0 Receive correct (20 points).
- Chunk reuqest message corect (5 points).
- Chunk response message correct (5 points).
- Donwload thread correct (10 points).
- File share thread correct (10 points).