Programming: Reliable Transport - Alternating Bit

This lab is to be completed in teams of two. Collaborative conversations between yours and other teams with regard to syntax, strategies and methods for completing the lab are encouraged; however design and implementation must be the work of the team that is handing in the final product. Make sure the work/understanding is shared between team members.

Overview

In this laboratory programming assignment you will use a guided, building block approach to write the sending and receiving transport-level code for an implementation of the simple reliable data transfer protocol (rtd 3.0). *Remember, the transport level is responsible for passing messages between the application layer and network layer on a host in order to allow two processes to communicate.*

Since we can't modify the Windows XP operating system on our lab computers, your code will have to execute in a simulated hardware/software environment. However, the programming interface provided to your routines, i.e., the code that would call your entities from above and from below is very close to what is done in an actual UNIX environment. (Indeed, the software interfaces described in this programming assignment are much more realistic than the infinite loop senders and receivers that many texts describe). Stopping/starting of timers is also simulated, and timer interrupts will cause your timer handling routine to be activated.

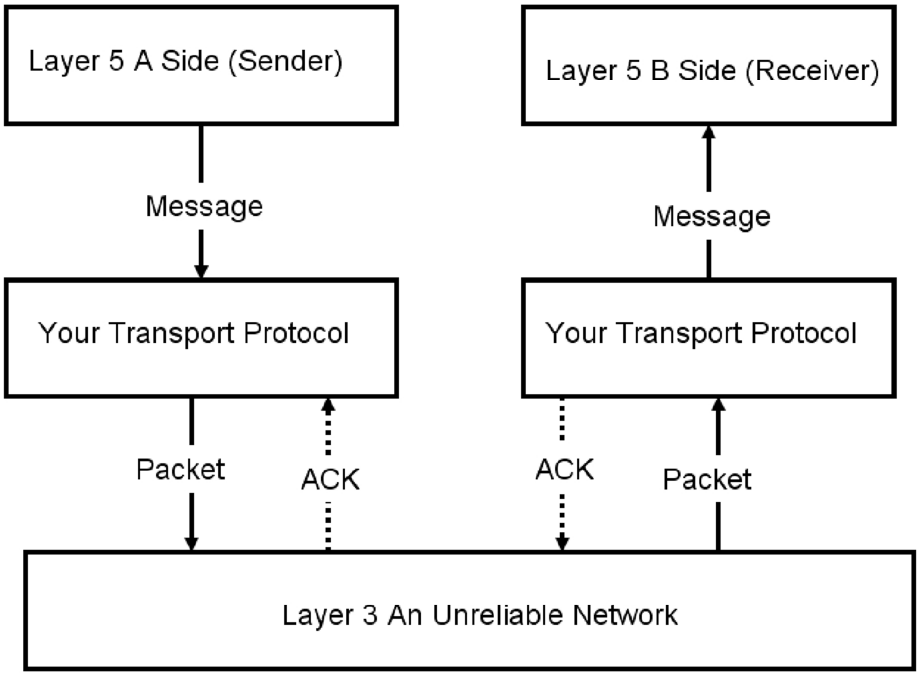
Step 1 Understanding the Simulated Environment

First you will load the code that creates the simulated environment as well as the **single** class that you will modify to complete this lab:

- **Download the ReliableTransportStarter.zip** file from the Campus EFREI.
- Unzip the file to a directory on your X drive where you plan to work.
- In NetBeans 5.5.x (or Eclipse), open a New Project and choose from the "General" category "Java Project with Existing Sources." => Next
- Give your project the name "<Your LName> Lab 7" and a working directory =>Next

- Add folder ReliableTransportStarter as your Source Package. =>Finish
- Open StudentNetworkSimulator.java and add your name and the date to the comments at the beginning of the file. **This is the only file you will need to read and modify.** The other files are provided only for your curiosity and convenience.

The supplied Java code simulates both an application layer (layer 5) and a network layer (layer 3) that your transport protocol must interact with. The methods you will write are for the sending entity (A) and the receiving entity (B). *Only unidirectional transfer of data (from A to B) is required. However, the B side will ultimately have to send packets to A to acknowledge receipt of data.* The general structure of the emulated environment is below:



The unit of data passed between the upper layers and your protocols is a *message*, which is a 20-byte string (implemented as a Java class with getData and setData methods). Your sending entity will thus receive data in 20-byte chunks from layer5; your receiving entity should deliver 20-byte chunks of correctly received data to layer5 at the receiving side.

The unit of data passed between your routines and the network layer is the *packet*, which is declared as:

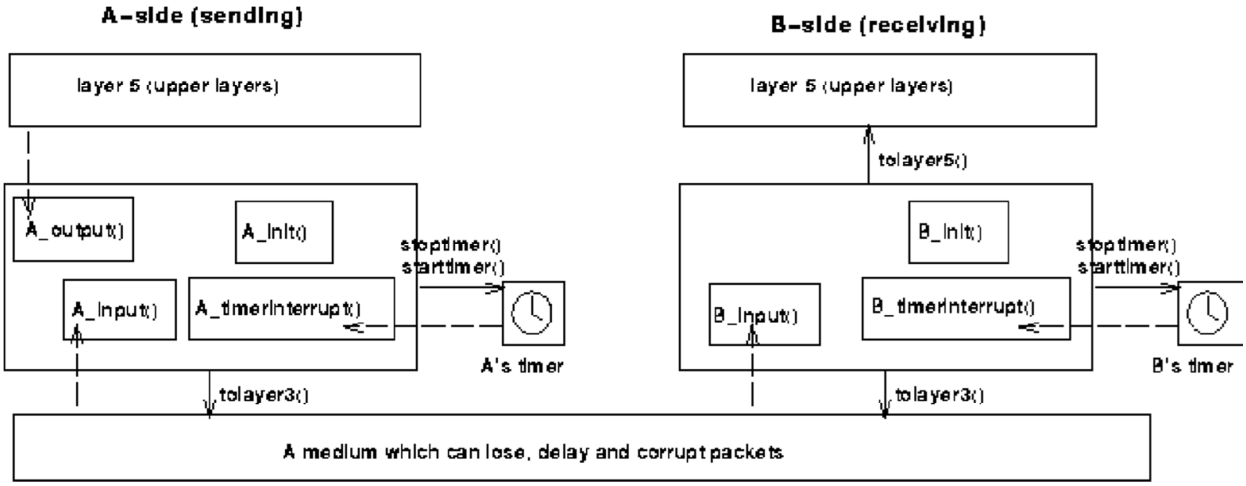
```
public  class  Packet
{
    private int seqnum;
    private int acknum;
    private int checksum;
    private String payload;
}
```

There are associated useful methods to manipulate a Packet object and those are detailed in the comments of the StudentNetworkSimulator.java class. The packet fields will be used by your protocols to insure reliable delivery, as we've seen in class.

The Transport Layer Routines You Will Write

The four methods you will fill in are detailed below. These methods will be called by (and will call) methods that have been provided to interact with the emulated network environment. Your routines will fill in the payload field from the message data passed down from layer 5 and send them through layer 3. As noted above, such methods in real-life would be part of the operating system, and would be called by other methods in the operating system.

1. **aOutput(Message message),** where `message` contains data to be sent to the B-side. This routine will be called whenever the upper layer at the sending side (A) has a message to send. It is the job of your protocol to insure that the data in such a message is delivered in-order, and correctly, to the receiving side upper layer.
2. **aInput(Packet packet) -** This routine will be called whenever a packet sent from the B-side (i.e., as a result of a `toLayer3()` being done by a B-side method) arrives at the A-side. `packet` is the (possibly corrupted) packet sent from the B-side.
3. **aTimerInterrupt() -** This routine will be called when A's timer expires (thus generating a timer interrupt). You'll probably want to use this routine to control the retransmission of packets. See `startTimer()` and `stopTimer()` below for how the timer is started and stopped.
4. **bInput(Packet packet) -** This routine will be called whenever a packet sent from the A-side (as a result of a `toLayer3()` being done by a A-side method) arrives at the B-side. The `packet` object is the (possibly corrupted) packet sent from the A-side.

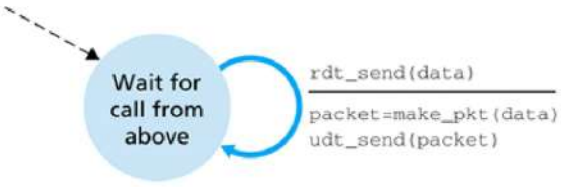


Software Interfaces

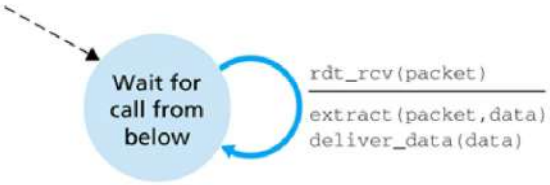
To help you with this project, you are provided with following routines which can be called by your methods:

- **toLayer3(int callingEntity, Packet p),** where `calling_entity` is either 0 (for the A-side send) or 1 (for the B side send), and `p` is a Packet. Calling this routine will cause the packet to be sent into the network, destined for the other entity.
- **toLayer5(String string)** With unidirectional data transfer, you would only be calling this from to the B-side. Calling this routine will cause data to be passed up to layer 5.
- **protected void startTimer(int callingEntity, double increment),** where `entity` is either 0 (for starting the A-side timer) or 1 (for starting the B side timer), and `increment` is a *double* value indicating the amount of time that will pass before the timer interrupts. A's timer should only be started (or stopped) by A-side routines, and similarly for the B-side timer. To give you an idea of the appropriate increment value to use: a packet sent into the network takes an average of 5 time units to arrive at the other side when there are no other messages in the medium.
- **stopTimer(int callingEntity),** where `entity` is either 0 (for stopping the A-side timer) or 1 (for stopping the B side timer).

Your first step is to write the java code that simulates a transport protocol that sits on a *completely reliable network layer*. Recall from the text that the FSM for the sender and receiver in this unlikely scenario is as follows:



a. rdt1.0: sending side



b. rdt1.0: receiving side

Figure 3.9 ♦ `rdt1.0` – A protocol for a completely reliable channel

After examining these FSMs you should very quickly realize that all the transport layer needs to do in this case is take a message from the application layer, packetize it by encapsulating it in

some additional data, and then send it through the network layer. The steps are reversed on the receiver side.

In StudentNetworkSimulator.java class this can all be accomplished by filling in the missing code from the following methods: `aOutput(Message message)` and `bInput(Packet packet).` You will simply need to use the previously mentioned software interfaces to turn a message into a packet, send it through layer 3, then retrieve the message on the receiver side and pass it up to layer 5. This should require no more than five lines of code.

When creating the packet for this first task, use a value of zero in each of the data fields int seq, int ack, int check. It is highly encouraged that you use `System.out.println` to display what data the message is comprised of (i.e. "A sent :"). When you compile and run the program, accept all defaults from the GUI control panel. Your output should look similar to this:

```
A  sent:  aaaaaaaaaaaaaaaaaaaa
B  received:  aaaaaaaaaaaaaaaaaaaa
A  sent:  bbbbbbbbbbbbbbbbbbbb
B  received:  bbbbbbbbbbbbbbbbbbbb
```

If your code works correctly, a file named OutputFile will be created in your Project Folder (not in your Source Folder) containing the contents of the packets sent from A to B.

Step 2 Create Pseudo Code

Now that your simple code works inside the emulator you should have an understanding of how your transport layer interacts with the layers above and below it. As noted in lecture, rdt 1.0 is basically useless and we didn't have a realistic protocol until we developed rdt 3.0. The ultimate goal of this lab is to implement rdt 3.0 within our emulator.

rdt 3.0 Receiver

Both the sender and receiver side of rdt 3.0 have complex FSM's (see the diagrams on the next page), and to get ourselves organized before we program their code we need to make some pseudo code**. Download the file "LabPsuedo.txt"** from Campus EFREI. It contains general pseudo code for most of the steps which need to take place for a successful coding of rdt 3.0… with the exception of some key words that were blanked out. Take the time now fill in the blanks of the pseudo code based on the steps dictated by the FSMs of rdt 3.0. **You will turn this page in for grading.**

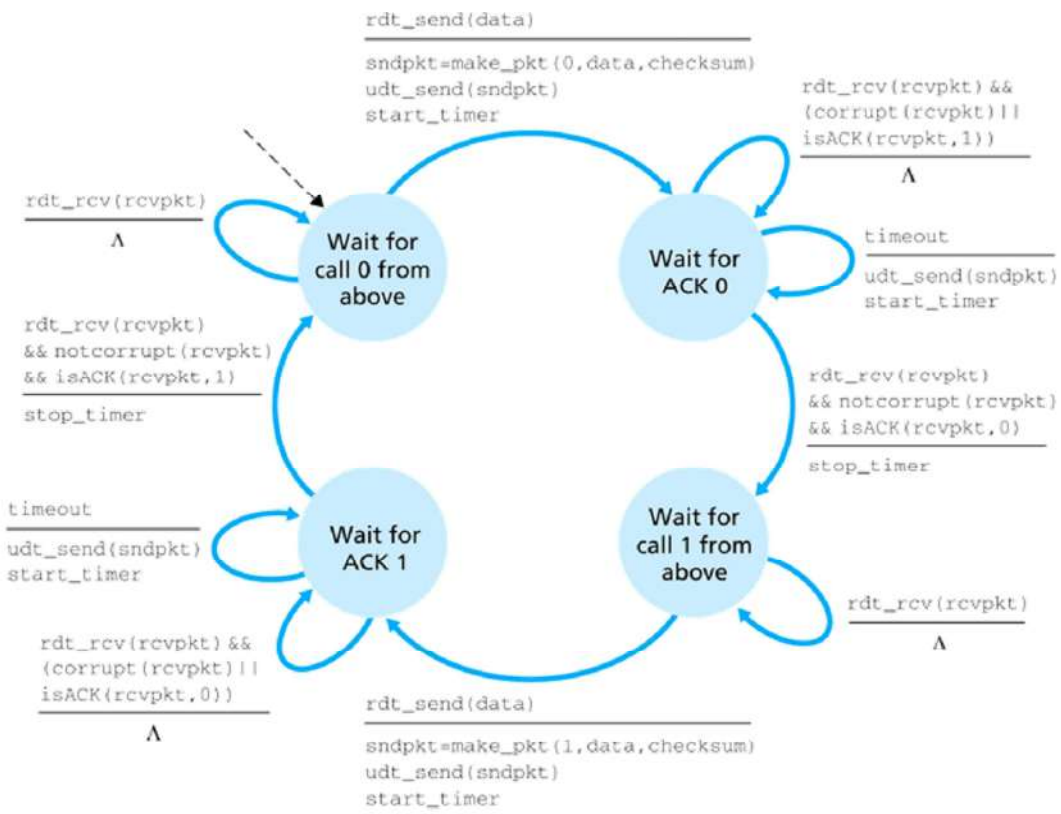
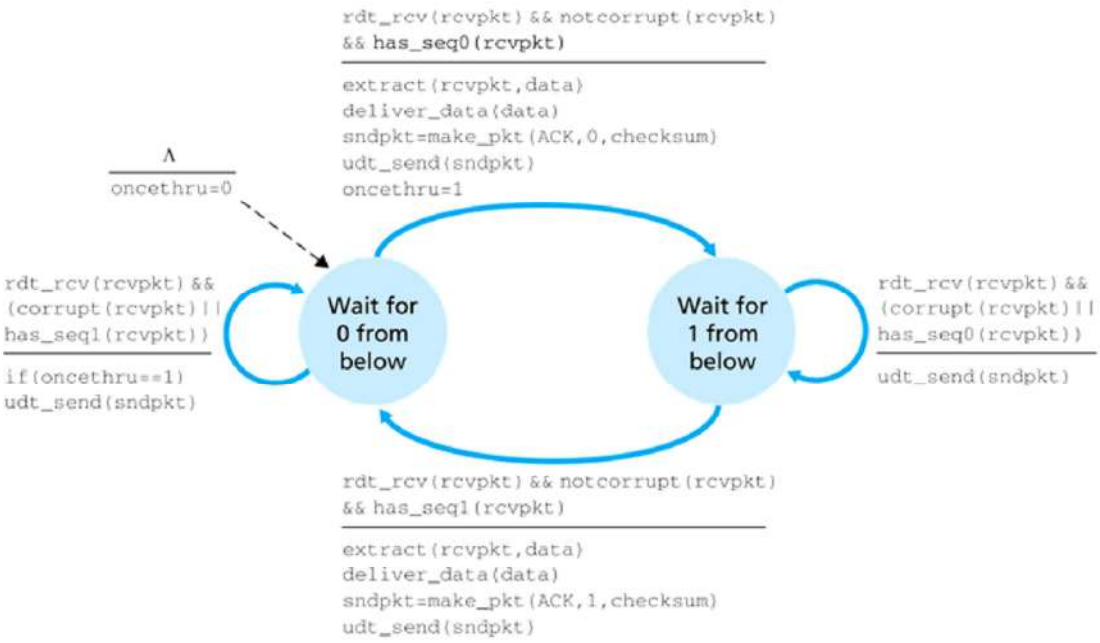


Figure 3.15 ♦ rdt3.0 sender



rdt3.0 Receiver

Step 3 Coding the Alternating-Bit-Protocol

Once you have organized your thoughts with pseudo code you are almost ready to write the methods `aOutput(), aInput(), aTimerInterrupt(),`and `bInput(),` which together will implement a stop-and-wait (the alternating bit) protocol which we referred to as rdt3.0 in the text. First you need more background on the environment.

The network layer is capable of corrupting and losing packets. It will not reorder packets. When you compile your methods and the provided code together and run the resulting program, you are able to specify the following values regarding the simulated network environment:

- **Number of messages to simulate.** The emulator (and your routines) will stop as soon as this number of messages has been passed down from layer 5, regardless of whether or not all of the messages have been correctly delivered. Thus, you need **not** worry about undelivered or unACK'ed messages still in your sender when the emulator stops. Note that if you set this value to 1, your program will terminate immediately, before the message is delivered to the other side. Thus, this value should always be greater than 1.
- **Loss.** You are asked to specify a packet loss probability. A value of 10% (0.1) would mean that one in ten packets (on average) are lost.
- **Corruption.** You are asked to specify a packet corruption probability. A value of 20% (0.2) would mean that one in five packets (on average) are corrupted. Note that the contents of payload, sequence, ack, or checksum fields can be corrupted.
- **Tracing.** Setting a tracing value of 1 or 2 will print out useful information about what is going on inside the emulation (e.g., what's happening to packets and timers). A tracing value of 0 will turn this off. A tracing value greater than 2 will display all sorts of odd messages that are for the emulator’s debugging purposes. A tracing value of 2 may be helpful to you in debugging your code.
- **Average time between messages from sender's layer 5.** You should choose a very large value for the average time between messages from sender's layer5 (~1000) so that your sender is never called while it still has an outstanding, unacknowledged message it is trying to send to the receiver.

Helpful Hints and the like

- A simplified checksum method has been provided for you. Furnish it the requested fields at either A or B and it will return the same int value.
- START SIMPLE. Set the probabilities of loss and corruption to zero and test out your routines. Then handle the case of one of these probabilities being non-zero, and then finally both being non-zero.
- Any shared "state" among your routines needs to be in the form of global variables. Any information that your methods need to save from one invocation to the next must also be a global (or static) variable. For example, your routines will need to keep a copy of a packet for possible retransmission. It would be a good idea for such a data structure to be a global variable in your code. Note, however, that if one of your global variables is used by your sender side, that variable should **NOT** be accessed by the receiving side entity, since in real life communicating entities cannot share global variables.

- There is a float global variable called *time* that you can access from within your code to help you out with your diagnostics msgs.
- **Debugging.** We'd recommend that you set the tracing level to 2 and put LOTS of printlns in your code while you are debugging your methods.

Submission

Turn in your filled in your hard copy of the pseudocode, an electronic copy of your modified StudentNetworkSimulator.java code and a hard copy printout of your code output running with trace level 2, 10 messages, a loss probability of 0.1 and a corruption probability of 0.3. Make sure you have included sufficient printlns in your code so that the output is clear. For example, your methods might print out a message whenever an event occurs at your sender or receiver.