RDT: Implementing a Reliable Transport Protocol

**Overview**

In this laboratory programming assignment, you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol. The protocol version you will be implementing is the Alternating-Bit-Protocol version. This lab should be **fun** since your implementation will differ very little from what would be required in a real-world situation.

Your code will 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 are also simulated, and timer interrupts will cause your timer handling routine to be activated.

**The routines you will write**

The procedures 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. Of course, the B side will have to send packets to A to acknowledge (positively or negatively) receipt of data. Your routines are to be implemented in the form of the procedures described below. These procedures will be called by (and will call) procedures that Dr. Kurose has written which emulate a network environment. The overall structure of the environment is shown below (structure of the emulated environment):

A diagram of a machine

Description automatically generated

The unit of data passed between the upper layers and your protocols is a *message,* which is declared as class MSG:

This declaration, and all other data structure and simulator routines, as well as stub routines (i.e., those you are to complete) are in the file rdtsim\_CSCI466.py described later. 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 class Pkt. Your routines will fill in the payload field from the message data passed down from layer5. The other packet fields will be used by your protocols to insure reliable delivery, as we've seen in class.

The routines you will write are detailed in the source code rdtsim\_CSCI466.py. The routines are part of the classes EntityA and class EntityB. 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.

**class EntityA:**

# The following method will be called once (only) before any other

# EntityA methods are called. You can use it to do any initialization.

#

# seqnum\_limit is "the number of distinct seqnum values that your protocol

# may use." The seqnums and acknums in all layer3 Pkts must be between

# zero and seqnum\_limit-1, inclusive. E.g., if seqnum\_limit is 16, then

# all seqnums must be in the range 0-15.

**def \_\_init\_\_(self, seqnum\_limit):**

**pass**

# Called from layer 5, passed the data to be sent to other side.

# The argument `message` is a Msg containing the data to be sent.

**def output(self, message):**

**pass**

# Called from layer 3, when a packet arrives for layer 4 at EntityA.

# The argument `packet` is a Pkt containing the newly arrived packet.

**def input(self, packet):**

**pass**

# Called when A's timer goes off.

**def timer\_interrupt(self):**

**pass**

**class EntityB:**

# The following method will be called once (only) before any other

# EntityB methods are called. You can use it to do any initialization.

#

# See comment above `EntityA.\_\_init\_\_` for the meaning of seqnum\_limit.

**def \_\_init\_\_(self, seqnum\_limit):**

**pass**

# Called from layer 3, when a packet arrives for layer 4 at EntityB.

# The argument `packet` is a Pkt containing the newly arrived packet.

**def input(self, packet):**

**pass**

# Called when B's timer goes off.

**def timer\_interrupt(self):**

**pass**

**NOTE:**

* You may provide helper functions, but the above functions MUST be defined.
* **Pass** is a python statement that does nothing. Similar to an empty c++ statement ; .

**Software Interfaces**

The procedures described above are the ones that you will write. The following routines have already been written and can be called by your code. These are functions that should be called from your EntityA and EntityB methods.

The first argument to each of these student-callable functions is the object that is invoking the function. Within an EntityA or EntityB method, that object is available as `self`. For example, to start a timer in one of your entity methods, you would do something like:

*start\_timer(self, 10.0)* # Start a timer that will go off in 10 time units.

Or to send a packet to layer3, you would do something like:

*to\_layer3(self, Pkt(...))* # Construct a Pkt and send it to layer3.

That is for each of the following functions, the ‘calling\_entity’ argument is *self*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**def start\_timer(calling\_entity, increment):** ## call the simulator to start the timer.

**the\_sim.start\_timer(calling\_entity, increment)**

**def stop\_timer(calling\_entity):** ## call the simulator to stop the timer

**the\_sim.stop\_timer(calling\_entity)**

**def to\_layer3(calling\_entity, packet):** ## call the packet to layer 3

**the\_sim.to\_layer3(calling\_entity, packet)**

**def to\_layer5(calling\_entity, message):** ## call the packet to layer 5

**the\_sim.to\_layer5(calling\_entity, message)**

**def get\_time(calling\_entity):** ## Get the simulated time

**return the\_sim.get\_time(calling\_entity)**

**The simulated network environment**

A call to procedure tolayer3() sends packets into the medium (i.e., into the network layer). The object method *input* is called when a packet is to be delivered from the medium to your protocol layer.

The medium is capable of corrupting and losing packets. It will not reorder packets. When you compile your procedures and my procedures together and run the resulting program, you will be asked to specify values regarding the simulated network environment:

SIMULATION CONFIGURATION

--------------------------------------

(-n) # layer5 msgs to be provided: 10

(-d) avg layer5 msg interarrival time: 100.0

(-z) transport protocol seqnum limit: 16

(-l) layer3 packet loss prob: 0.0

(-c) layer3 packet corruption prob: 0.0

(-s) simulation random seed: 1691767103276928000

--------------------------------------

* **(-n) Number of messages to simulate.** The simulator (and your routines) will stop as soon as this number of messages have 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 simulator 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.
* **(-l) Loss.** You are asked to specify a packet loss probability. A value of 0.1 would mean that one in ten packets (on average) are lost.
* **(-c) Corruption.** You are asked to specify a packet loss probability. A value of 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. Your checksum should thus include the data, sequence, and ack fields.
* **(-v) 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 my own simulator-debugging purposes. A tracing value of 2 may be helpful to you in debugging your code. You should keep in mind that *real* implementors do not have underlying networks that provide such nice information about what is going to happen to their packets!
* **(-d) Average time between messages from sender's layer5.** You can set this value to any non-zero, positive value. Note that the smaller the value you choose, the faster packets will be be arriving to your sender.

**The Alternating-Bit-Protocol Version of this lab.**

You are to implement a stop-and-wait (i.e., the alternating bit protocol, which we referred to as rdt3.0 in the text) unidirectional transfer of data from the A-side to the B-side. **Your protocol should use both ACK and NACK messages.**

You should choose a very large value for the average time between messages from sender's layer5, so that your sender is never called while it still has an outstanding, unacknowledged message it is trying to send to the receiver. I'd suggest you choose a value of 1000. You should also perform a check in your sender to make sure that when class EntityA’s output method is called, there is no message currently in transit. If there is, you can simply ignore (drop) the data being passed to class EntityA’s output method .

**This lab can be completed on any machine supporting C. It makes no use of UNIX features.** (You can simply  copy the prog2.c file to whatever machine and OS you choose).

We recommend that you should hand in a code listing, a design document, and sample output. For your sample output, your procedures might print out a message whenever an event occurs at your sender or receiver (a message/packet arrival, or a timer interrupt) as well as any action taken in response. You might want to hand in output for a run up to the point (approximately) when 10 messages have been ACK'ed correctly at the receiver, a loss probability of 0.1, and a corruption probability of 0.3, and a trace level of 2. You might want to annotate your printout with a colored pen showing how your protocol correctly recovered from packet loss and corruption.

**Helpful Hints and the like**

* **Checksumming.** You can use whatever approach for checksumming you want. Remember that the sequence number and ack field can also be corrupted. We would suggest a TCP-like checksum, which consists of the sum of the (integer) sequence and ack field values, added to a character-by-character sum of the payload field of the packet (i.e., treat each character as if it were an 8 bit integer and just add them together).
* Note that any shared "state" among your routines needs to be in the form of global variables. Note also that any information that your procedures 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 probably 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 connected only by a communication channel can not 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.
* **START SIMPLE.** Set the probabilities of loss and corruption to zero and test out your routines. Better yet, design and implement your procedures for the case of no loss and no corruption, and get them working first. Then handle the case of one of these probabilities being non-zero, and then finally both being non-zero.
* **Debugging.** We'd recommend that you set the tracing level to 2 and put LOTS of printf's in your code while your debugging your procedures.
* **Random Numbers.**The simulator generates packet loss and errors using a random number generator. Our past experience is that random number generators can vary widely from one machine to another. You may need to modify the random number generation code in the simulator we have suplied you. Our emulation routines have a test to see if the random number generator on your machine will work with our code. If you get an error message:

It is likely that random number generation on your machine is different from what this simulator expects. Please take a look at the routine jimsrand() in the simulator code. Sorry.

then you'll know you'll need to look at how random numbers are generated in the routine jimsrand(); see the comments in that routine.

**Q&A**

When Kurose first taught this lab in his introductory neworking course, students have posed various questions.  If you are interested in looking at the questions we've received (and answers), check out <http://gaia.cs.umass.edu/kurose/transport/programming_assignment_QA.htm>