Programming Assignment 2: Reliable Data Transport Protocol

For this assignment you are allowed to work and submit in groups of two.

This assignment counts for 10% of the total module mark. Failure in this assessment may be compensated for by higher marks in other components of the module. The purpose of the assignment is the implementation of a reliable data transport protocol to understand the principles behind such protocols and to better understand TCP. It partly assesses the following learning outcomes:

- (2) Understand how the notion of layering and abstraction apply to the design of computer communication networks.
- (4) Understand the organisation of the Internet, and how this organisation relates to the OSI seven layer model
- (5) Understand the principles of the key protocols that govern the Internet .

The assignment is due Wednesday, 8 November 2017, 17:00.

Late submissions are subject to the University standard system of penalties. (cf. Section 6 in Code of Practice on Assessment)

This assignment assesses the following learning outcomes:

- Understand how the notion of layering and abstraction apply to the design of computer communication networks;
- Understand the principles of the key protocols that govern the Internet.

In this assignment you will implement a reliable data transport protocol. You can use any operating system when programming this assignment.

Submission Instructions

- Make sure that all your submitted files contain your names and student-IDs in the header.
- Submit only from one student account but make sure to include both names in the header.
- Only submit two files called `Sender.java` and `Receiver.java`.
- If you finished your assignment CLICK HERE TO SUBMIT.

Overview

In this programming assignment, you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol. You have the choice between two versions of the assignment:

- The basic assignment will be to implement the Stop-and-Wait Protocol.
- The more challenging assignment will be to implement a Go-Back-N protocol.

For the basic assignment you cannot get a mark >80%.

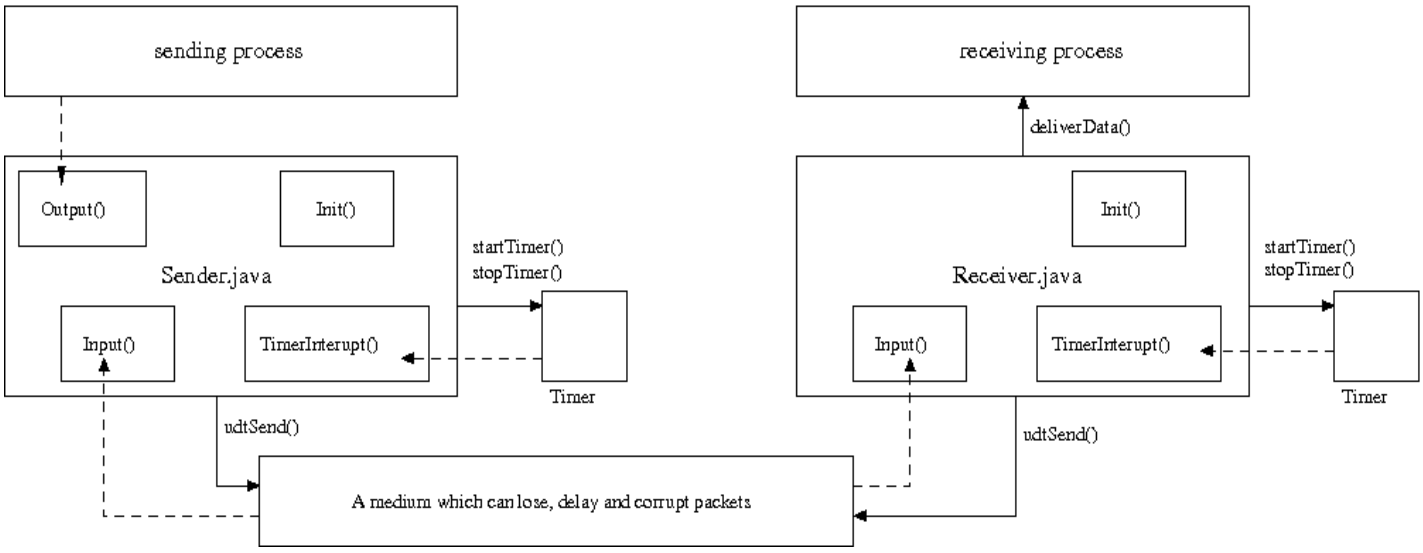
Your implementation will differ very little from what would be required in a real-world situation.

Since we do not have standalone machines (with an OS that you can modify), 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 (i.e., from application layer) and from below (i.e., from network layer)) is very close to what is done in an actual UNIX environment. Stopping/starting of timers are also simulated, and timer interrupts will cause your timer handling routine to be activated.

Note that you do not need a network connection to run this assignment, so you can do it pretty much on any machine you would like.

The code you will write

The methods you will write are for the sending entity (`Sender.java`) and the receiving entity (`Receiver.java`). Only unidirectional transfer of data (from sender to receiver) is required. Of course, the receiver side will have to send packets to the sender to acknowledge (positively or negatively) receipt of data. Your code is to be implemented in the form of the methods described below. These methods will be called by (and will call) methods that have already been written which emulate a network environment. The overall structure of the environment is shown in Figure 1:



The unit of data passed between the application layers and your protocols is a *message*,

which is declared as:

```
public class Message {
   private String data;
}
```

This declaration, and all other data structure and emulator classes are already constructed. Your sending entity will thus receive data in 20-byte chunks from the application layer; your receiving entity should deliver 20-byte chunks of correctly received data to the application layer 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;
}
```

Your class methods will fill in the payload field from the message data passed down from the application layer. The other packet fields will be used by your protocols to insure reliable delivery, as we've seen in the lectures.

You will have to write methods for two classes as detailed below. As noted above, such procedures in real-life would be part of the operating system, and would be called by other procedures in the operating system.

Sender.java:

`Output(message),`
where `message` is an instance of the class `Message`, containing data to be sent to the receiver.
This method will be called whenever the application layer at the sending side (`Sender.java`) 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 application layer.

`Input(packet),`
where `packet` is an instance of the class `Packet`.
This method will be called whenever a packet sent from the receiver-side (i.e., as a result of a `udtSend()` being called by a Receiver method) arrives at the sender-side. `packet` is the (possibly corrupted) packet sent from the receiver-side.

`TimerInterrupt()`
This method will be called when the timer of the sender expires (thus generating a timer interrupt). You'll probably want to use this method to control the

retransmission of packets. See `startTimer()` and `stopTimer()` below for how the timer is started and stopped.

`Init()`
This method will be called once, before any of your other sender-side methods are called. It should be used to do any required initialization.

Receiver.java:

`Input(packet),`
where `packet` is an instance of the class `Packet.`
This method will be called whenever a packet sent from the sender-side (i.e., as a result of a `udtSend()` being called by a Sender method) arrives at the receiver-side. `packet` is the (possibly corrupted) packet sent from the sender-side.

`Init()`
This method will be called once, before any of your other receiver-side methods are called. It can be used to do any required initialization.

Software Interfaces

The methods described above are the ones that you will write. The following methods have already been written. These methods can be called by your methods:

`startTimer(increment),` 开始计时
where `increment` is a *double* value indicating the amount of time that will pass before the timer interrupts.
To give you an idea of the appropriate increment value to use: a packet sent into the network takes an average of 10 time units to arrive at the other side when there are no other messages in the medium. Thus, in expectation the RTT (round trip time) is about 20 time units. A good value for `increment` is twice that much.

`stopTimer(),` 停止计时
for stopping the timer.

`udtSend(packet),` 向网络层发送数据包
where `packet` is an instance of the class `Packet.`
Calling this method will cause the packet to be sent into the network, destined for the other entity.

`deliverData(message),` 向应用层交付信息（只需要在receiver里使用）
where `message` is an instance of the class `Message.` With unidirectional data transfer, you would only be calling this within `Receiver.java`.
Calling this method will cause data to be passed up to the application layer.

The simulated network environment

A call to the method `udtSend()` sends packets into the medium (i.e., into the network

layer). Your `Input()` methods are 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 classes and the pre-coded classes together and run the resulting program, you will be asked to specify values regarding the simulated network environment:

- **Number of messages to simulate.** The emulator (and your methods) will stop as soon as this number of messages have been passed down from the sender's application layer, 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 0.1 would mean that one in ten packets (on average) are lost.
- **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.
- **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 in-depth emulator-debugging purposes. 设置为2，便于debug
  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!
- **Average time between messages from sender's application layer.** 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.

Different Options for the Assignment

You have the choice of implementing either the Stop-and-Wait or the Go-Back-N version of this assignment.

For both versions you have to write some methods for `Sender.java` and `Receiver.java`. You will find a link to a skeleton of these files and also the other supporting java classes at the end of this document.

Stop-and-Wait version

You are to write the methods, `Output()`,`Input()`,`TimerInterrupt()` and `Init()` for `Sender.java`, and `Input()` and `Init()` for `Receiver.java`. Together these will implement a stop-and-wait unidirectional transfer of data from the sender-side to the

receiver-side. Your protocol should be similar to rdt3.0, which we discussed in the lectures. **It is your choice whether you want your protocol to be NACK-free or whether you want to use ACK and NACK messages.** For a NACK-free protocol you can e.g. implement the rtd3.0 sender on slide 3.37 together with the rdt2.2 receiver on 3.35.

You should choose a very large value for the average time between messages from sender's application layer, 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 `Output()` is called, there is no message currently in transit. If there is, you can simply ignore (drop) the data being passed to the `Output()` routine.

Make sure you read the ``helpful hints'' for this assignment following the description of the extra credit assignment.

Go-Back-N version

You are to write the methods, `Output(),Input(),TimerInterrupt()` and `Init()` for `Sender.java`, and `Input()` and `Init()` for `Receiver.java`. Together these will implement a Go-Back-N unidirectional transfer of data from the server-side to the receiver-side, with a window size of 8.

I would **STRONGLY** recommend that you first implement the basic assignment (Stop-and-Wait) and then extend your code to implement the extra-credit assignment (Go-Back-N). Believe me - it will **not** be time wasted! However, some new considerations for your Go-Back-N code (which do not apply to the Stop-and-Wait protocol) are:

`Output(message),`
where `message` is an instance of the class `Message,` containing data to be sent to the receiver-side.
Your `Output()` method in `Sender.java` will now sometimes be called when there are outstanding, unacknowledged messages in the medium - implying that you will have to buffer multiple messages in your sender. Also, you'll now need buffering in your sender because of the nature of Go-Back-N: sometimes your sender will be called but it won't be able to send the new message because the new message falls outside of the window.
Rather than have you worry about buffering an arbitrary number of messages, it will be OK for you to have some finite, maximum number of buffers available at your sender (say for 50 messages) and have your sender simply abort (give up and exit) should all 50 buffers be in use at one point. In the ``real-world,'' of course, one would have to come up with a more elegant solution to the finite buffer problem!

`TimerInterrupt()` 超时，计时停止
This method of `Sender.java` will be called when the timer expires (thus generating a timer interrupt). Remember that you've only got one timer, and may have many outstanding, unacknowledged packets in the medium, so you'll have to think a bit about how to use this single timer. In fact, we discussed how to do this

in the lectures.

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. I 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).
  You should make sure that both, the sender and receiver, use the same checksumming approach. One way to ensure this is to implement a checksumming method and include it in `Sender.java` and `Receiver.java`.
- Note that any shared ``state'' among your methods needs to be in the form of global variables. Note also that 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 methods 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.
- There is a global variable called *time* (of type double) 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 methods. Better yet, design and implement your classes 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.** I'd recommend that you set the tracing level to 2 and put LOTS of `System.out.println()` in your code while your debugging your classes.

Evaluation

You should make sure that your code compiles. Code which does not compile will receive at most 20%.

We will assess your assignment using the following questions:

Stop-and-Wait Protocol

- Is there a clean output, free from messy debugging messages?
- Does the protocol work with corruption on?
- Does the protocol work with lost packets?

Go Back N Protocol:

- Is there a clean output, free from messy debugging messages?
- Does the protocol work with corruption on?
- Does the protocol work with lost packets?

- Does the protocol work with out-of-order packets?

Other Characteristics:

- Is the code commented?
- Is it free of numerical constants sprinkled in the code?
- Is it indented correctly?

Code

Here are the JAVA files that you'll need:

- `Assignment2.java`
- `Event.java`
- `EventList.java`
- `Message.java`
- `NetworkHost.java`
- `NetworkSimulator.java`
- `Packet.java`
- `Receiver.java`
- `Sender.java`
- `Testing.java`(Command line interface used to test your protocol.)

Here is a zip-achive of all java files:

- Assign2code.zip

This zip file also includes `Testing.java`, which is a command line interface for testing your protocol. The following is an example for using this interface for sending 20 mesages over a channel with loss probability 0.1, corruption probability 0.2, delay between messages 1000, trace level 2, and seed 256:

```
java Testing 20 0.1 0.2 1000 2 256
```

Some notes:

- `Sender` and `Receiver` are the only classes that you will have to modify. Both classes extend `NetworkHost`.
- `NetworkSimulator` is the bulk of the simulator.
- `Packet, Message, Event,` and `EventList` are support classes. `Assignment2` is the "driver" for the whole thing.
- `Sender.java` and `Receiver.java` contain inline comments documenting the interfaces of the other classes that you will need. These class files will need to be in the CLASSPATH (which will happen automatically if you edit and compile your java files in the same directory as the other class files).