Assignment 1

Reliable Data Transport

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 will implement two different versions:

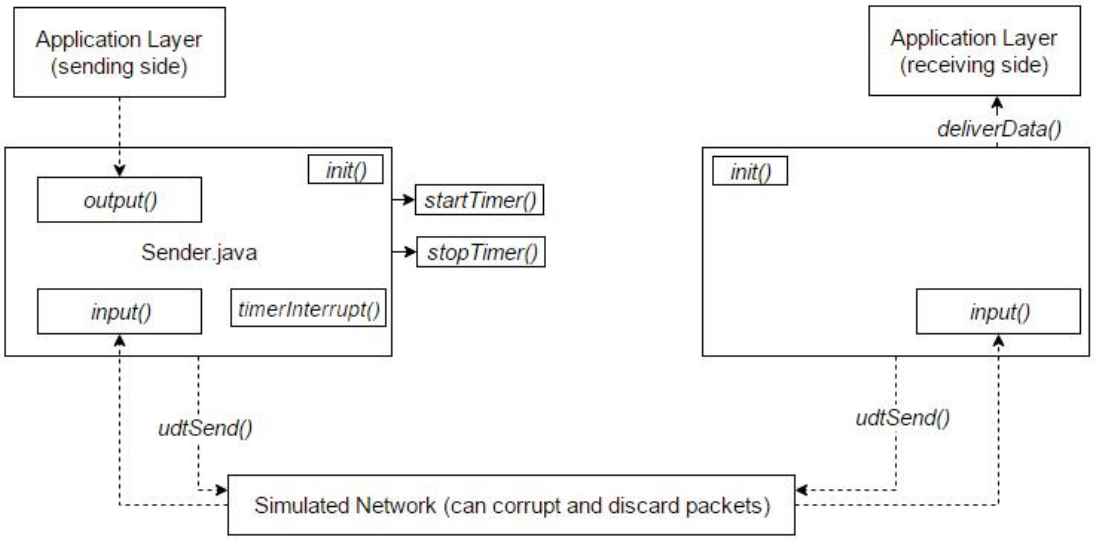
Stop-and-Wait Protocol.

Go-Back-N protocol.

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 the code you will write (i.e., the code that will 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 real operating systems. Stopping/starting of timers is also simulated, and timer interrupts will cause your timer handling methods to be activated.

Note: you do not need a network connection to run this assignment; you can work on any machine you like (Java 7 or 8 and Netbeans 8.0.2 or later are the only requirements).

The methods you will write are for the sending host (Sender.java) and the receiving host (Receiver.java). Both these classes extend the NetworkHost class which is provided to you. **Only unidirectional transfer of data (from sender to receiver) is required.** The receiver side will send packets to the sender only to acknowledge received 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 are provided to you, which simulate a network. The overall structure of the environment is shown in the Figure below.



The unit of data passed between the application layer and your protocol is a message, which is declared as follows:

```
public class Message {
    private String data;        // 20-byte data chunks
```

}

This class is provided to you. Your sending entity will thus receive data in 20-byte (MAXDATASIZE static variable defined in NetworkSimulator.java file) 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 the methods you will implement and the (simulated) network layer is a packet, which is defined as follows:

```
public class Packet {
    private int seqnum;         // sequence number
    private int acknum;         // acknowledgement number
    private int checksum;       // checksum
    private String payload; // payload carried in the packet
}
```

This class is provided to you. The methods that you will implement will fill in the payload field using the message data passed down from the application layer. The other packet fields will be used by your protocols to ensure reliable delivery. **You will have to write methods for two classes as detailed below.**

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 (by the simulator) whenever the application layer at the sending side has a message to send. It is the job of your protocol (Sender.java) to ensure that the data in such a message is delivered correctly, to the receiving side application layer (through the receiver side of your transport protocol).

input(packet): where packet is an instance of the class Packet. This method will be called (by the simulator) 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 (by the simulator) 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 (by the simulator) once, before any of your other sender-side methods are called. You can use it to initialise any required data structures for your transport protocol.

Receiver.java

input(packet): where packet is an instance of the class Packet. This method will be called (by the simulator) 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 (by the simulator) once, before any of your other receiver-side methods are called. You can use it to initialise any required data structures for your transport protocol.

Software Interfaces

The methods described above are the ones that you will write. The following methods are provided to you and can be called by your code:

startTimer(increment): where increment is a double value indicating the amount of time (in time units) that will pass before the timer is fired. 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, on average the RTT (round trip time) is about 20 time units. A good value for increment is twice that much (i.e. 40 time units or more).

stopTimer(): Stops the timer.

udtSend(packet): where packet is an instance of the class Packet. Calling this method will cause the packet to be sent into the simulated network, destined for the other entity. That's where the packet gets "lost" or corrupted.

deliverData(message): where message is a String representing the packet payload. 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. It just prints the received data. Remember that this is a simulation of a network and there is no real receiver application running to process this data.

The simulated network environment

A call to the method udtSend() simulates the sending of packets to the network. Your input() methods (Sender and Receiver) are called whenever a packet is to be delivered from the network to your transport layer.

The network may corrupt and lose packets. It will not reorder packets. When you compile your classes and the provided classes together and run the resulting program, you will be asked to specify values regarding the simulated network environment:

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.
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 (in terms of simulated time units) packets will be be arriving to your sender.

Stop-and-Wait

You will implement 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 application layer data from the sender side to the receiver-side.

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 (effectively dropping) the data being passed to the output() routine.

Go-Back-N

You will implement 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 suggest you to first implement the Stop-and-Wait and then extend your code (in a different Netbeans project) to implement the Go-Back-N version. Some considerations for your Go-Back-N protocol (which do not apply to the Stop-and-Wait protocol) are as follows:
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 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(): 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.

Hints

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.

There is a global variable called time (of type double in time units) 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.

Running the simulator twice with the same seed will give the same exact results, as the seed defines the way random numbers, which in turn define the simulation, are generated.

You can treat the network simulation code as a black box. The description of the methods above should be enough. However, I suggest you to spend a couple of hours understanding how simulation is done. This may make things even clearer.

Marking Criteria

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

I will assess your assignment using the following criteria:

Stop-and-Wait Protocol (45%)

Does the protocol work with corruption on?

Does the protocol work with lost packets?

Go Back N Protocol (40%):

Does the protocol work with corruption on?

Does the protocol work with lost packets?

Description of Protocols (10%):
Is there description for both versions of the protocol?
Is the description well-written with clear references to the source code?
Other Characteristics (5%):
Is the code commented appropriately?
Is it indented correctly?
Is naming of variables and methods sensible?
Source Code
You should start by downloading the .zip file at the end of the page that contains a Netbeans project. The provided source code is as follows:
Assignment.java: contains the main method. All arguments are read from the standard input (keyboard)
Event.java: represents a simulated event of one of the possible types (see below)
EventList.java: stores all events that are currently scheduled
EventType.java: type of the event, FROMAPP, FROMNETWORK, TIMERINTERRUPT
Message.java: represents a message pushed by the application layer
NetworkHost.java: implements functionality for your transport protocol. Sender.java and Receiver.java inherit from this class and can directly use its methods. udtSend() which simulates losses, and data corruption is implemented here.
NetworkSimulator.java:the core functionality of the simulator
Packet.java: represents a packet that is sent in the simulated network.
Receiver.java: the receiving side of your transport protocol. You should implement the class.
Sender.java: the sending side of your transport protocol. You should implement the class.

Remember to rename the Netbeans project to the appropriate names (for each different protocol version), as mentioned below.
Submission Guidelines
You should submit the coursework by the deadline posted on Sussex Direct (and the e-submission link on Study Direct). Standard penalties for late submissions will be applied. You must submit a .zip file containing the following:
Well-formatted and well-documented source code (written in Java) in two separate Netbeans projects that can be compiled and run (including the provided files). The Netbeans projects, named StopAndWait and GoBackN, respectively, will include all required source code to execute the 2 different transport protocols.
A short report (up to 1500 words) describing your protocols, the source code and the design decisions that you made.
Please do not put your names on your submissions, but do include your candidate number. Failure to submit source code, as described in the first bullet, will result to a zero mark as I will not be able to assess your programming effort.
Plagiarism and Collusion
The coursework you submit is supposed to have been produced by you and you alone. This means that you should not: work together with anyone else on this assignment
give code for the assignment to other students
request help from external sources

do anything that means that the work you submit for assessment is not wholly your own work, but consists in part or whole of other people's work, presented as your own, for which you should in fairness get no credit

If you need help ask your tutor The University considers it misconduct to give or receive help other than from your tutors, or to copy work from uncredited sources, and if any suspicion arises that this has happened, formal action will be taken. Remember that in cases of collusion (students helping each other on assignments) the student giving help is regarded by the University as just as guilty as the person receiving help, and is liable to potentially receive the same penalty. Also bear in mind that suspicious similarities in student code are surprisingly easy to spot and sadly the procedures for dealing with it are stressful and unpleasant. Academic misconduct also upsets other students, who do complain to us about unfairness. So please don't collude or plagiarise.