Reliable Data Transfer Protocol

This application handles the sending and receiving of data at the transport layer. The application executes in a simulated network environment which mimics that of a real OS.

Data is sent over a network via packets. In this application, the contents of each packet include:

- A Sequence number
- An Acknowledgement number
- A Checksum
- Packet data

Packets are sent from entity A and received by entity B. B will have to send a response packet to A, to acknowledge receipt of data. The program must be able to recover from packet corruption and packet loss, which are common occurrence with data transfer.

Implementation

aOutput

The '*aOutput*' method takes a message as a parameter which is passed from the upper application layer i.e sender A. The data within this message must be delivered in-order and correctly to the receiving application layer i.e receiver B. There are a number of precautions that must be taken in to ensure successful transfer.

Firstly as this is a unidirectional transfer of data (side A to side B) a stop and wait functionality will be implemented, allowing only one packet to be sent at a time. If there is a packet currently being sent to side B, return 0, refusing further packets from being sent. Otherwise return 1, indicating that the message has been accepted and sent.

A checksum must then computed on the content of the message sent by side A. The message data will be passed as a String parameter to the checksum method. The method will loop through the string adding the value of each character to the checksum. However, any part of package is susceptible to corruption, not just the message data. To cater for this, the values of the sequence number and acknowledgement will be added to the checksum. The result of the checksum will be included prior to sending the packet to side B.

The package is then assembled and includes the sequence number to keep track of the packets, checksum as well as the message data. A copy of the packet is then saved as an instance variable incase it needs to be resent.

A timer must be implemented in order to cope with packets that did not successfully travel from A to B. The predefined method '*startTimer*' is passed two values. The first specifies which entity from which it starts i.e side A and the second specifies the amount of time that will pass before the timer is interrupted.

Finally the packet is passed to the lower network layer and sent to receiver B. '*aOutput*' is then instructed to use the stop and wait functionality, refusing to send the next packet until an acknowledgment has been returned.

bInput

The '*bInput*' method receives and manages the packet sent from A. It must first of all confirm that packet has not been corrupted.

To ensure that no corruption has occurred, the packet information must first be extracted i.e sequence number, acknowledgement, checksum and packet payload. A checksum of the packet should be recomputed (value of the payload checksum + sequence number + acknowledgement).

```
// the packet should be re-computed using the checksum method
int checksum = checksum(msg)+receivedseq+receivedack;
```

The recomputed checksum must then be compared to the checksum value sent in the packet. If they match the packet has not been corrupted.
If the packet is found to be corrupt a negative response i.e nack will be returned to A.

Assuming that the packet is clear of corruption, the following check must identify that it is in fact the next expected packet. In order to confirm this, the expected sequence number i.e '*expectedPacketSeq*' is compared to sequence number sent in the packet.

If so firstly pass the payload up to the application layer and then create a new packet with a positive response i.e '*ack*' to be sent to A.

Any returning packet could potentially arrive corrupted, even though it does not have a payload. This can be dealt with in the same way as the packet sent from A, by sending the sum of the values (sequence number and acknowledgement) as the packet checksum.
Pass packet down to the transport layer and increment the sequence number i.e '*expectedPacketSeq*' to expect the next packet.

If the sequence check fails and it is not the expected packet, it means that it has already previously been received, but the acknowledgement was lost or corrupted on return to A. This scenario is know as packet duplication.

In the event that this response acknowledgement is lost, the timer interrupt will kick in assuming that the initial packet never reached B and will have it resent.

We know the sequence number has already been incremented, only expecting the next packet, not the packet it has just previously received. The solution to this problem is quite simple, send back another positive acknowledgement. This packet's sequence number should use the same sequence number as the one received.

```
else{
    // packet is still susceptible to corruption even without a payload
    int chkAck = receivedSeq+ack;
    // return positive response i.e ack
    // resend the packet using the recieved sequence number
    Packet p = new Packet(receivedSeq, ack, chkAck, null);
    // send the ack packet down to the transport layer
    toLayer3(B, p);
}
```

aInput

The '*aInput*' method receives and manages the packet sent from B. Similarly to '*bInput*' it firstly confirms that the packet is not corrupt as well as ensuring that the packet contains a positive acknowledgement i.e '*ack*'.

If both of these scenarios are true, the timer that was started when the initial packet was sent(or restarted) will be stopped. The stop and wait functionality will be set to false, allowing another packet to be sent. Finally the sequence number will be incremented i.e '*packetSeq*', so the next packet will have the next sequence number.

If either scenario is incorrect, the timer will be stopped, reset and the packet re-sent.

aInit & bInit

These two methods are loaded before any of the side A or side B routines are called. They are used to do any pre required initialisation.

aTimerInterupt

As previously discussed this method will kick in after a set time if no acknowledgement has been received from B. It stops, restarts the timer and re-sends the packet.

This output demonstrates:

- 5 messages sent from sender A to receiver B
- packet loss probability of 20%
- packet corruption probability of 30%
- trace level of 2 (detail of output)

```
EVENT time: 3169.920875551842  type: 1  entity: A
Packet 2 being sent, Stop and Wait
Timer started
Send packet to B
toLayer3: packet being lost
FROMLAYER5: data accepted by layer 4 (student code)

EVENT time: 3219.920875551842  type: 0  entity: A
PACKET LOST
Restart timer
Resend packet
toLayer3: packet being lost

EVENT time: 3269.920875551842  type: 0  entity: A
PACKET LOST
Restart timer
Resend packet
toLayer3: packet being lost

EVENT time: 3319.920875551842  type: 0  entity: A
PACKET LOST
Restart timer
Resend packet

EVENT time: 3321.387294024102  type: 2  entity: B
Sucessful packet return ack
toLayer3: packet being corrupted
toLayer3: sequence number being corrupted

EVENT time: 3329.032350444608  type: 2  entity: A
Packet corrupted, restart timer, resend packet
toLayer3: packet being corrupted
toLayer3: acknowledgement number being corrupted

EVENT time: 3337.147149751975  type: 2  entity: B
Packet is corrupt, return nack

EVENT time: 3343.394547934103  type: 2  entity: A
Packet corrupted, restart timer, resend packet
toLayer3: packet being lost

EVENT time: 3393.394547934103  type: 0  entity: A
PACKET LOST
Restart timer
Resend packet

EVENT time: 3402.6956457287897  type: 2  entity: B
Packet is a duplicate

EVENT time: 3411.6937936704308  type: 2  entity: A
Packet 2 recieved sucessfully
Timer Stoped
Waiting false
```