COMPUTER NETWORKS REPORT – ASSIGNMENT 1

Introduction

In the following report I will discuss the two versions of simple reliable data transmission (rdt) protocols implemented, which include Stop-and-Wait and Go-Back-N, with descriptions of the how each protocol operates, I will then use my source code for the sending host and receiving host to describe how I have implemented these protocols along with any design choices made.

Protocols

What is RTD?

For connection-oriented service provided by TCP, it is necessary to have a reliable data transfer (RDT) protocol to ensure delivery of all packets and to enable the receiver to deliver the packets in order to its application layer (Gshute, n.d.).

RTD is split into three versions, 1.0, 2.0 and 3.0. Tools such as error detection, sequence numbering, timers & feedback are essential for any RDT implementation on a unreliable network. RDT 1.0 is for reliable transfer over a reliable channel. This means the underlying channel is perfectly reliable with no lost packets or bit errors. This is very unrealistic as there will always be errors especially as networks can be unreliable (Geeks for Geeks, 2019).

RTD 2.0 protocol is a channel with bit errors, with the underlying channel may flip bits in the packet. Here checksum is used for a form of error detection. Checksuming compares the sequence number with the packet to the expected sequence number, where if they don’t match, there has been a bit error. Uses acknowledgements (ACKs) which are the receiver explicitly tells the sender the packet has OK, and negative acknowledgements (NAKs) if the packet had errors. The sender will retransmit packet on receipt of NAKs. RDT 2.0 has features such as error detection & feedback over RDT 1.0. This is the Stop-and-Wait Protocol, where the sender sends one packet then waits for receiver’s response.

For RDT 2.0, the sender has 2 states, one which waits for the call above, and the other waits for the acknowledgements (ACKs or NAks). The sender will change state once a packet has sent to the acknowledgment waiting state, and will only leave that state once an ACK is received. If NAks are received, the packet will continue to be re-transmitted. The receiver will contain only 1 state, where if the packet is not corrupt, an ACK is sent, else sends a NAK back to the sender. This is a fatal flaw with RDT 2.0, as corruption is only checked on the receiving side, and not on the sender’s side, since corruption can happen anywhere and is not limited to just the receivers side. We need to check is the NAKs or ACKs are corrupted, if so we cannot say anything. This is that the sender may receive inconclusive feedback (or corrupted). To combat this re-transmission is used, but could lead to possible duplication. In order to handle duplicates, sender re-transmits current pack if acknowledgment is corrupt. The sender adds the sequence number to packet, and the receiver can discard packet if it's a duplicate.

This is RTD 2.1. Only two sequence numbers are needed 0 or 1, due to only 1 packet being in-flight at a time, two states will wait for 0 or 1 call from above, and two states wait of ACK or NAK 0/1. Duplicate detection is needed here as corruption of the acknowledgements would be listed as corrupt. In RTD 2.1, corruption detection is now implemented on the sender's side. We acknowledge previous packet received if a duplicate is found. So if a duplicate is found with a 0, the receiver will replace the 0 with a 1 and vice versa. So if the sequence number us incorrect, a duplicate packet has been found.

Overall on the sender's side of RTD 2.1, Sequence numbers are added to the packet, leading to the number of states doubling, as the state must 'remember' whether expected packet should have either a 0 or 1, along with checking acknowledgment isn't corrupt. The receiver must check for supplicate packets, as the states determines whether a 0 or 1 is expected. RTD 2.2 is a NAK free protocol, with the same functionality of RTD 2.1, sending the ACK for last OK packet received, rather than a NAK. The duplicate ACK at sender results in same action carried out if a NAK was received.

Stop-And-Wait

RTD 3.0 is the version that was implemented in the assignment, where it describes the actions to take for a channel with errors and packet loss. This is the most realistic version of RDT stated as channels tend to be unreliable. If packet is lost, send it again, but we don't receiver ACKS, how do we know what has happened? The approach used is timeouts. The sender waits a set amount of time for ACK, and re-transmits packet if no ACK received within time. If packet delayed/lost, re-transmission will be a duplicate, but sequence number handles this. The receiver must specify the sequence number of packet being ACK'd, therefore requires a countdown timer. In action, if a packet or ACK is lost, once the countdown timer finishes, the sender will resend the packet. If the ACK was lost, the receiver will acknowledge the duplicate and send the correct ACK. If the ACK was delayed or even the timeout was premature, the sender will ignore the duplicate ACK as it is expecting an ACK with a different sequence number. The performance of RDT 3.0 is bad. The network protocol limits the use of physical resource by sending one packet at a time, and waiting for an acknowledgement before sending another packet. To improve performance, we need to send more packets while waiting for ACKs (pipelining).

Go-Back-N

By having a greater window size for sender, the utilisation would increase. Go-Back-N Protocol effective increases the utilization by sending more than one packet before receiving the first acknowledgement. Pipelining is when the sender allows multiple packets to be in-flight, increasing the number of yet-to-be-acknowledged packets. To allow this the range of sequence numbers must be increased, but to a reasonable number. Too many packets being sent could lead to far too many timeouts / losses to recover from, which requires lots of re-transmission. In addition, this would lead to congestion on the network, reducing performance. Furthermore swamping the receiver with packets is not good, as the window size is still only 1, so can only handle 1 packet at a time. For Go-Back-N, the sender can have up to N unack'd packets in pipeline, where N is the window size. Receiver only sends cumulative ACK, so wont ACK packets if there's a gap. Sender will have a timer for oldest unACk'd packet (base), so when the timer expires, **all unack'd packets** are retransmitted.

The sender will use a k-bit sequence number in the packet's header, and a window of up to N, consecutive unack'd packets allowed, This will include sent packets not yet ACK'd and usable packets that haven't been sent yet. The method ACK(n) will ACK all packets up to, including sequence number n, known as the cumulative ACK.

The receiver will always send ACK for a correctly received packet with highest in-order sequence number. This may generate duplicate ACKs, but by remembering the sequence number, duplicate ACKs will be ignored. The receiver will discard any frame that does not have the exact sequence number it expects (either a duplicate frame it already acknowledged, or an out-of-order frame it expects to receive later) and will resend an ACK for the last correct in-order frame.

My Implementation & Design Choices

Variables

For the receiver hosts in both versions, two variables where used, the constant ACK set to 1, and the expected Sequence number (expectedNo), which was set to 0, in the initialise method (init).

For the sender hosts in both versions, the two constants ACK set to 1, and RTT set to 20 where used. In the Go-Back-N implementation a further constant of Window_Size was set to 8, given in the brief. This constant represents the max number of unAck'd packets in the pipeline. I decided to set to make this a constant and not to set it in the init method as it can be updated more easily. The variable used for both protocols are the current sequence number (currSeqNo) which is used to help duplicate detection., set to 0 in the init method. The Stop-And-Wait implementation also has a currentPacket variable which will store the current packet, saved if the need for re-transmission is needed. The Go-Back-N implementation has an array list of packets to represent the pipeline used to buffer and allow for re-transmission along with a variable to store the sequence number of the base packet which is essential in the case of re-transmission.

Private Methods

The private methods used where checksum, corruptCheck & duplicateCheck (used for error detection). duplicateCheck was not used in Go-Back-N, as by remembering the sequence number, the duplicate packets are ignored.

Checksuming, method name checkSum takes the payload of packet is input and computes the SUM of the ascii codes of individual chars and returns the sum of message chars ascii values. A for loop was used here as the length of the payload can be calculated using .length().

Corruption checking, method name corruptCheck, returns true if and only if packet was corrupted during transmission, with the Input being the packet being checked. This is done by comparing the checksum known (using method getChecksum()) of the packet with a manual calculation of the checksum being the overall addition of the checksum of the payload, the sequence number and the ACK number.

Duplicate checking, method name duplicateCheck, will return true if and only if the packet received has been received before i.e. is a duplicate of an already received packet with an input of the packet being checked. This is done by comparing the sequence number known (getSeqnum()) to a calculation ((currSeqNo + 1) % 2). Modulo 2 is used as there can only be 2 sequence numbers, 1 or 0, so this will ensure the output will be a 1 or 0. This isn't used for the Go-Back-N protocol as the sequence numbers have been increased, so can be more than a 1 or 0. To combat duplication checking in Go-back-N the sequence number is remembered.

Zero check method is a design choice made and only used in the Go-Back-N protocols receiver. It is called only in the input method used to avoid indexing errors, where the udtSend method executed with the expectedNo (expected sequence number) minus 1 as a parameter. As there is a chance the expected sequence number is 0, the packet before would be calculated as -1, causing an index error. This method just increases the expected number from 0 to 1, so the packet is essentially resent as no previous packets have been received OK.

Public Methods

Initialise (Init) method is used in both sender and receiver hosts for both implementations. For both the receiver hosts, the init method just initialises the expected sequence number to 0, as when the transmission begins the expected number will always be 0. For Stop-And-Wait, the init method in the sender current sequence number to 0, and the current packet to null. The Go-Back-N sender similarly sets the base sequence number and current sequence to 0, and initialises pipeline. This method will be called once, before any of your other sender-side methods are called.

The input method, with signature input, is also used by both sender and receiver hosts for both implementation. For Stop-And-Wait receiver host, this method will be called whenever a packet sent from the sender(i.e. as a result of a udtSend() being called by the Sender) arrives at the receiver. The argument "packet" is the (possibly corrupted) packet sent from the sender, so a corruption check and duplicate check is carried out (by calling the private methods). If the packet is a duplicate or corrupt the sequence number is updated and re-transmitted, otherwise if it isn't corrupt or a duplicate, deliver with correct ACK number. For the sender host for Go-Back-N, If ACK isnt corrupted or a duplicate, transmission done. Updated base used to remove Acked packets from pipeline, and stops timer if no more packets in pipelines, otherwise restart timer.

The input method used for Stop-And-Wait receiver host check If transmission is complete by seeing whether the packet was a duplicate or corrupted. If OK, current packet set to null, current sequence number updated (using modulo formula shown above), and the timer is stopped.

The rest of the public methods are only used by the sending hosts for both implementation. The output method for Stop-And-Wait, it will only process incoming message, as long as no packet in transmission, the packet data is set up, saves current packet in case of the need for re-transmission and Sends packet to receiver and starts timer. For the Go-back-N protocol, the output method also takes the message as an input. The job of the protocol is to ensure that the data in such a message is delivered in-order, and correctly, to the receiving application layer, which this method does. First ensures the message is only processed if unacked packets are less than the window size. If so, the packet data is set up, the packet is created, and added to pipeline for re-transmission. Then sends packet to receiver, if no packets in pipeline, then start the timer, finally updating the current available sequence number.

The timer interrupt method is used by both sending hosts. This method will be called when the senders's timer expires (thus generating a timer interrupt) and this method is used to control the retransmission of packets. For Go-Back-N Protocol, If one packet was lost or delayed heavily, restart timer, retransmit packet waiting for Ack.