INF-2300

Assignment 3, Reliable Transport Protocol

Magnus Dahl-Hansen

November 2022

Introduction

This report describes the implementation of the *Go Back-N* algorithm used by the transport layer for reliable data transfer over an unreliable channel. The Algorithm is mainly used to handle three problems that can occur during data transfers, which is: p*acket loss*, *latency/delay* and *corruption.*

Technical background

Before reading how the code was implemented, the reader of this report should be familiar with the *Go Back-N* (GBN) protocol.

In the GBN protocol the sender is allowed to send multiple packets of data before receiving acknowledgment from the receiver. When the receiver receives a packet it will either acknowledge it and send an acknowledgment message back to the sender if it was received in the correct order, or it will discard the packet because it was corrupted or it arrived out of order.

If the sender receives an acknowledgment for the lowest package number it sent, it will replace the package in the window with the next package that needs to be sent. However should the sender not receive an acknowledgment for the lowest package number sent, a timer will expire after an agreed upon time limit which in turn will tell the sender to re-send all the packages in the current sending window. The timer is then reset when the first package is sent.

If the sender receives an acknowledgment message for a package that is not the lowest package number in the sending window, it will again re-send all packages in the sending window because this indicates that the receiver may not have received a package.

Design&implementation

In this simulation Alice is the sender and Bob is the receiver of the data. The simulation starts by checking whether or not Bob's received data is equal to Alice's original data. If Bob has received all data then the program will exit, but if not Alice will divide her data into chunks and transfer it to the transport layer from the application layer. To keep track of how many packages has been sent, received, acknowledged and which package can not be sent three variables is used: *base, next sequence number* and *window size*. *Base* is used to keep track of the lowest package number that has been sent but is not yet acknowledged, *next sequence number* points to the next available space in the sending window and *window size* indicates how many packages can be placed inside the sending window. In combination, these three variables is able to tell which package that is yet to receive an acknowledgment, if the sending window can receive another packet, how many packages that is sent, if the sending timer should be reset and when all packets have been sent and acknowledged.

When a package reaches the transport layer it is marked with an unique ID before it is placed inside the sending window. In a case where the sending window is already full, a check is made to see if Alice has received an acknowledgment message for the lowest package ID in the window. If she have received an acknowledgment message the packet is replaced and the new packet is sent. Should however this not be the case, then the packet is stored in a buffer and sent when an acknowledgment is received and there has opened up an available space in the window. If the new packet needs to be buffered, then all the packages in the current window is sent again because the last sent package may not have been received by Bob or the acknowledgment message which Bob sent may be lost.
Should there be space in the sending window then the package is placed inside the window and sent to Bob. During these processes a check is made to see if the *base* number is equal to the *next sequence number.* If this is the case it means that the lowest package number in the window is sent and that the timer should be reset.

A packet that is sent contains a *receiver* variable which tells who is going to receive this package. If Bob is the receiver he will proceed to check if the package ID/number is the one he is expecting (Bob is receiving packages in sequential order starting from id: 0). If Bob receives a packet number that is expected he will proceed to check if the package ID is the last one that Alice has sent. This check is only necessary due to how the pre-implemented code delivers packets to Alice and is not how a program like this would work in a real situation. However if the package ID is correct then Bob will deliver the package's data to the application layer and "receive" the data, then proceed to change the packets *acknowledgment* variable to *True* and send the package back to Alice.
If the package that Bob receives is out of order, or Alice has not received acknowledgment for the last packet she sent, then Bob will proceed to send a package to Alice with the ID of the last package that he has received and acknowledged.

When Alice is the receiver of a packet it can only be an acknowledgment packet from Bob. So Alice will check if the package ID is one that is contained in the sending window. If the received package has the lowest ID in the sending window then it will be replaced with a new packet when it arrives from Alice’s application layer. If the package that Alice receives from Bob is out of order or it does not arrive at all, then Alice will wait for the timer to expire and re-send all the packages in the sending window.

The current state of the program will in total send 6 packets with a sending window size of 5 packets. The drop and delay chance for a packet is set to 25% and 50% while the amount of time that Alice will wait for an acknowledgment message is 5 seconds. In addition, each time a new tick occurs in the program the simulation is delayed for 2 seconds so that the user can easily see what actually happens during the simulation.

Results&discussion

The implemented code is able to handle the sending and receiving of packages between two parts when the packets are being randomly dropped, delayed and when the sending window is both equal to and less than the total number of packets sent. It is however not able to handle corrupt packages.

When the code is run with a chance of packets being dropped it will run correctly until Bob has received all packages. However the program is struggling to exit because the last acknowledgment message that Alice receives (which will quit the program) is never registered correctly. This minor problem is most likely caused by a “*if*-sentence” being written incorrectly which in turn causes Alice’s timer to expire and she will keep re-sending packages from the sending window until the program ended by the user.

There has not been made an attempt to implement a code that handles corrupt messages due to time restraints. Still, had there been more time this could have been solved by a simple *string compare* check. The pre-implemented function which corrupts a packet’s data will modify the data (which originally only contains uppercase letters) to lower case letters. So a simple check for any lower case letters contained in the packet that Bob receives will tell Bob that the packet is corrupted. In such a case the packet would be discarded and Bob would add a *corrupted* variable to the packet and send it back to Alice.

A last mention of this section is that the implemented code is not an exact implementation of the Go-Back-N protocol. The reason for this is that the pre-code will always send only one packet at a time from the transport layer to the network layer, which in turn will cause Bob as the receiver to handle one packet at a time before Alice can send more packets. In a real case scenario Alice would be able to send multiple packets to Bob before he handles the packets and send them back. The way that the pre-code is implemented has, in my opinion, made this task more difficult to solve than necessary. The reason is that the order of how packets should be received and handled becomes completely different than if all the packets had first been sent by Alice and then handled by Bob. To try and solve this issue and make the Go-Back-N protocol work somewhat identical to a real life scenario there have been created variables in the *config* file which is globally accessible for both Bob and Alice so they can check each others status for received and acknowledged packets.

Conclusion:

This assignment involved implementing a reliable transport protocol where Go-Back-N has been implemented. This implementation is able to handle the sending and receiving of packages between two parts where the packets may be randomly dropped and delayed. If a package is corrupt it will not be handled, though a possible solution for this has been discussed in the *Result&Discussion* section.