# Networks Coursework

## Introduction

For this coursework, I implemented two different transport layer algorithms – Stop-and-Wait and Go-Back-N. Both algorithms are given data by the application layer and must transfer it across the network reliably and in order, although they have different ways of doing this. Broadly, Stop-and-Wait can only send one packet at a time while Go-Back-N allows multiple packets to be in the pipeline at once. The former is useful when a connection has a lot of interference, as packets will need resending frequently and there is a low cost to resend a single packet, while the latter is useful when a connection is over a large physical distance, as there is a high expected round trip time so having many packets ‘in the air’ at once is helpful.  
  
When programming, I aimed to use class variables rather than literal values wherever possible. This makes the code much more readable and means the specifics of implementation (i.e. the allowable sequence numbers) can be changed later in development. Some values, such as expected round trip time, won’t change during execution so are made constant. To further aid readability, I aimed to move any repeated (*incrementSequenceNumber*) or particularly clunky (*validateInPacketWindow*) snippets of code into their own methods. All methods and variables are given descriptive names so that the code is easy to understand, though the code is also commented with more detailed explanations.

## Stop and Wait

### Description

This protocol works by only allowing one packet to be sent at a time. Until the current packet is acknowledged, no other packets may be sent. In this implementation, for simplicity, any data sent by the application while a packet is still sending is ignored. This protocol ensures in order transfer by virtue of only sending one packet at a time. It ensures packets aren’t lost by resending packets after a specified amount of time has passed without acknowledgement. It ensures reliability by using a checksum (which is calculated on both the sender and receiver sides) to check for corruption – if a packet is corrupted, it’s simply treated as lost. To deal with duplicate message packets, the receiver ignores the message and resends the acknowledgement (as it was probably lost / corrupt). To deal with duplicate acknowledgement packets, the sender just ignores the packet.

### Design Decisions

To make sure only one packet is being sent at a time, class variable *currentlySending* is used. Initially it’s set to false, and as soon as a packet starts being sent it’s set to true. While it’s true, the *output* method does nothing – all the code in this method is contained within the statement *if(!currentlySending)*. Any data sent from the application while *currentlySending* is true is essentially lost, therefore. As soon as an uncorrupted acknowledgement is received for the current packet, *currentlySending* is set back to false and *output* is free to take a new message.  
  
With this limitation in place, there’s no need to implement a packet buffer. However, the sender does still have to store the *currentPacket*. This is necessary in case the packet is lost or corrupted, so that it can be resent without having to request the data from the application layer again. Current packet doesn’t need to be reset to null when sending is over as it’s contents are overwritten whenever a new packet is sent anyway.  
  
In this implementation, only two sequence numbers are used – 0 and 1. This is the minimum required to ensure that duplicate packets can be dealt with appropriately as it allows the protocol to differentiate between the current (expected) packet and the previous packet. The code can be changed to allow any sequence numbers such that MIN < MAX, using 0 and 1 was just preference.  
  
Checksumming was done by simply calculating the sum of the ASCII values of each character in the message and adding the sequence and acknowledgment numbers. This was done as it’s easy to calculate on both the sender and receiver sides and it ensures that corruption in any of the 4 fields in the packet will be caught.

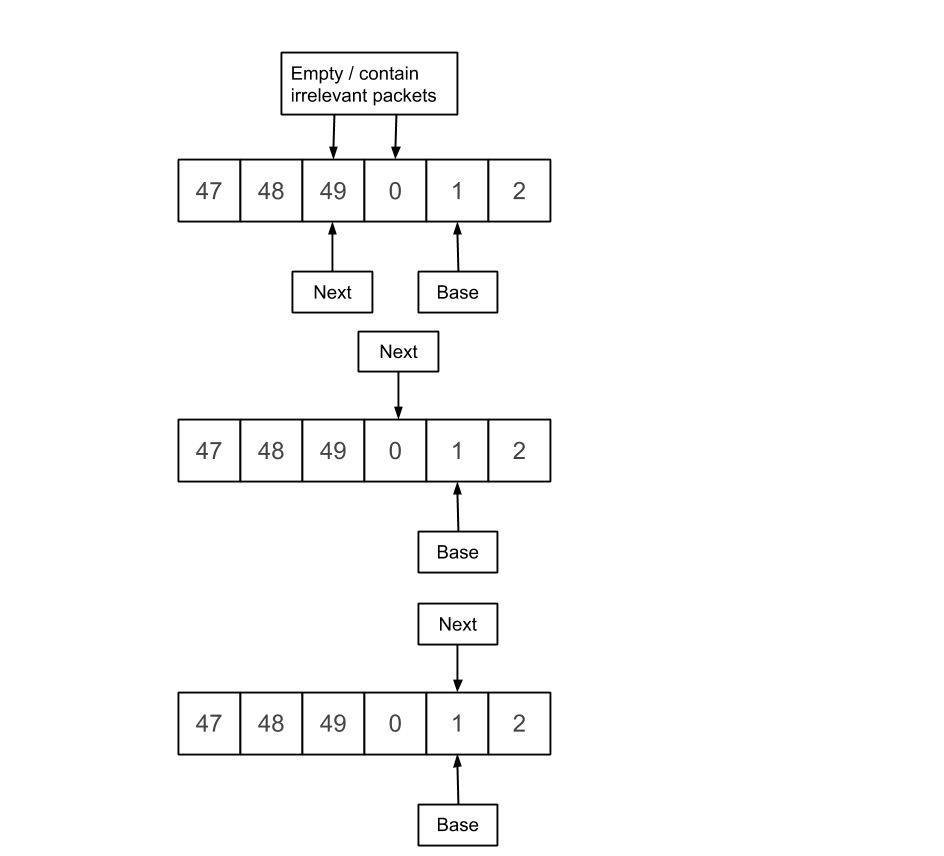
## Go Back N

### Description

This protocol works by allowing N packets to be sending (waiting on acknowledgement) at a time. In this case, a window size of 8 is used. A packet buffer is used on the sender side to hold packets that are currently in the air or waiting to be sent. Packets must still be sent in order, so the receiver keeps track of the expected sequence number and sends an acknowledgment only when a packet with the correct sequence number arrives. On the sender side, acknowledgements are understood to be cumulative – if an ack number is out of order, then all the packets before that must have also been accepted. To account for loss, a timer is started when the base packet (first packet of the window) is sent and when the timer expires all packets within the window are resent. To account for corruption, checksums are calculated on both the sender and receiver sides and any corrupt packets are treated as lost. To deal with duplicate packets, the receiver would ignore the packet as it doesn’t have the expected ack number and the sender would ignore the packet as the ack number isn’t in the packet window, so it isn’t for one of the packets currently in the air.

### Design Decisions

To implement this protocol, I used an array of size 50 as the buffer. I used the sequence numbers 0 to 49 in order to use them to index the array, as well as to treat it circularly. I had to implement various functions to do with the packet window, such as check whether a packet is within the window or get the sequence number of the end of the packet window. This is necessary due to the wrap around from 49 to 0 – simply using *baseSequenceNumber + WINDOWSIZE* for these calculations would have created array index out of bounds errors, as numbers could have exceeded 49.  
  
To deal with the problem of having a finite buffer, the program terminated once the buffer is full. Programmatically, the array having a packet stored in all indexes doesn’t mean it’s full since we’re using it circularly. We never actually delete any old information, we just keep track of where relevant information is, so the array may be ‘full’ while only a small portion contains relevant packets. To tell when we run out of space for new relevant packets, we check whether incrementing *nextSequenceNumber* would make it *baseSequenceNumber*. As the *next* number is always after the *base* number, this would indicate that the *next* number has move through every available sequence number and circled back to the *base*, so the array is full. At this point an error message is printed and the program terminated.



In this final case, a packet overwrites the contents of index 0 and next is incremented. After it’s incremented it’s equal to base, so this is when execution would need to be halted as the buffer is full.

Here, index 49 has been overwritten with a new packet (with data from the application layer). After Next is incremented, it doesn’t equal base, so there is still room in the buffer and execution can continue.

In this case, everything is fine, execution can continue. Index 49 and 0 are treated as empty, so new packets added to the buffer will overwrite their contents.