Computer Networks Coursework 1

Description of StopAndWait Implementation

The stop and wait protocol works by sending one data packet and verifying that it is uncorrupted on receipt before moving on to the next packet in the data stream,. The sequence then changes to 1 or 0 dependent on its current number to ensure that a previous data packet is not being referenced. Due to this method, any incoming data coming from the app layer will be discarded, which is why 1000 was choosen for the frequency of incoming messages, in order for all the messages to be sent and received. If the data packet that was sent is lost or corrupted the data packet is re-sent following a timeout or a negative acknowledgement or NAK. The code is described below in order of appearance:

- Sender.java:
  - Variables:
    - seqNum – This keeps track of the sequence number of the packet
    - sndpkt – The current Packet being sent to the reciever, used to resend if need be
    - state – corresponds to the finite state machine's states given in the lecture slides. This is implemented in order to keep track and know what to do with any inputs/outputs to the protocol
    - increment – the increment for the timer, how long it waits before resending a packet, set to 40 as suggested in the brief.
  - Methods:
    - check(Packet p) – checks the incoming packet from the reciever, for errors. It does this by adding the ACK, Sequence and Checksum numbers together and checking the result to be -1, returning true if this is the case.
    - getChecksum(int sNum, int aNum, String data) – produces a checksum for the new packet using the data of the packet, the sequence number and acknowledgement number. It uses the TCP checksum method by adding the numbers together, the ascii values of the data is used here, and carrying out a 'ones complement' on the result, which is the checksum. This method makes the process more simple to check as seen above, and is also reliable due to the components involved.
    - init() - initialises the integer variables to zero. These being Sequence number and state.

        - output(Message message) – This is called when the app layer has a message to send. If this is called at the correct state, 0 or 2, where we are not in the process of looking for an ACK from the current packet, then we create a new packet and set this to be the current packet or sndpkt. We then send this to the reciever, start the timer and increase the state. If more than 3 then we set the state back to 0, as we have looped around the FSM.
        - input(Packet packet) – This is called when a packet is sent from the receiver. It checks that it is the correct state to receive an ACK packet, and calls check(Packet p). If it is in sequence, not corrupt and an ACK (not a NAK), then the timer is stopped to increase the sequence number and state number, and reset to 0 if they are more than 1 or 3 respectively.
        - timerInterrupt() - stops the timer, sends the current sndpkt and then starts the timer.
- Receiver.java
    - Variables:
        - seqNum – This keeps track of the sequence number of the current packet.
    - Methods:
        - check(Packet p) – checks the incoming packet from the sender for errors. It does this by adding the ACK, Sequence, Ascii data values and Checksum numbers together and checking the result to be -1, returning true if this is the case.
        - getChecksum(int sNum, int aNum) – produces a checksum for the acknowledgement packet, using the the sequence number and acknowledgement number. It uses the TCP checksum method of adding the numbers together and doing 'ones complement' on the result, which is the checksum. This method makes the process more simple to check as seen above, and is also reliable due to the components involved.
        - init() - initialises seqNum to 0.
        - input(Packet packet) – When a packet is sent from the sender this method calls check(Packet p). If it is in sequence and not corrupt, the data is delivered to the app layer, and an ACK packet is returned, increasing the seqNum (making sure it flips to 0 or 1). If it isn't in sequence and not corrupt an ACK packet is sent back. If it is corrupted a NAK packet is sent back. All packets call getChecksum(int sNum, int aNum) when created.

Description of GoBackN Implementation

The Go back N protocol is slightly different from Stop and Wait. It is quicker than Stop and Wait, due to the way it controls the sending and receiving of packets. The protocol has a window size, in this case 8. It sends up to 8 packets at a time to the reciever, dependant on the incoming messages from the app layer. The reciever then sends back the highest sequence numbered packet it received correctly. The sender then assumes that all packets before this are accepted by the sender and queues more packets and so on. This protocol requires a buffer as well, in order to cope with the large volume of requests from the app layer. This is to ensure no messages are lost, this implementation has a buffer size of 50, which updates everytime a message is removed in-order to welcome the new messages from the app layer.

- Sender.java
  - Variables:
    - nxtSeqNum – the next sequence number for the next new packet. Starts at 0, due to the ease of use with arrays in this implementation.
    - sndpkt - the current array of packets sent to the reciever, updates every time one is added and removed, i.e. added from app layer or received ACK for.
    - base – Tracks the current smallest seqence number packet in the process of being sent to the reciever. Starts at 0 for the same reason as nxtSeqNum.
    - windowSize – The window size of the sent packet window. Set to 8.
    - increment - the increment for the timer, how long it waits before resending a packet, set to 40 as suggested in the brief.
    - maxBuffered – the maximum amount that can be buffered at any one time, set to 50.
    - msgCount – the current amount of messages that have been revceived from the app layer.
    - buffer – the array of buffered messages from the app layer.
  - Methods:
    - check(Packet p) – same as in Stop and Wait protocol - checks incoming packets from the receiver for errors
    - getChecksum(int aNum, int sNum, String data) – same as in Stop and Wait protocol – returns calculated checksum number.
    - out(Message message) – if nxtSeqNum is less than the sum of base and windowSize, this sends the next new packet to the reciever, and starts the timer.

If base is equal to the next sequence packet, it is also increased. This is called in output() and input(), when there are still messages in the buffer array.

- init() - sets nxtSeqNum, base and msgCount to 0. Creates new array of windowSize and maxBuffered for sndpkt and buffer respectively.
- output(Message message) – adds the new message from app layer to the buffer array, increases msgCount and calls out(Message message) with this current message.
- input(Packet packet) – if the packet is not corrupt and is an ACK then the current array is copied from the packet's seqNum, into a temp array, and then copied back to remove reduant packets. Base is then increased by the packet's seqNum + 1; if base then equals nxtSeqNum the timer is stopped. Otherwise, if there are still messages in the buffer, call out() with the specified message denoted by the nxtSeqNum modded by the max number of buffered messages, and finally start the timer.
- timerInterrupt() - start the timer, and send all packets currently in the sndpkt array.

- Receiver.java
  - Variables:
    - expectedSeqNum – the number that the sequence number is currently expected to be.
    - sndpkt – the current sndpkt that is sent when there are no new ACK packets to send back.
  - Methods:
    - check(Packet p) – same as Stop and Wait protocol - returns true if incoming packet is not corrupted.
    - getChecksum(int seq, int ack) – same as Stop and Wait protocol - returns checksum value given sequence number and acknowledgement number.
    - init() - sets expectedSeqNum to 0, and creates a new packet for sndpkt.
    - input(Packet packet) – if the incoming packet is not corrupted and the packet is the expected sequence number the data is delivered to the app layer, then a new sndpkt is created with this sequence number, and sent back. Expected sequence number is increased by 1. Otherwise the current sndpkt is sent back.