Mohiuddin Mondal
Roll- 001910501043 (A2)

Computer Networks
3rd year 1st Sem BCSE UG
Assignment 2

Problem statement: Implement three data link layer protocols, Stop and Wait, Go Back N Sliding Window and Selective Repeat Sliding Windowfor flow control.

Description: Sender, Receiver and Channel all are independent processes. There may be multiple Transmitter and Receiver processes, but only one Channel process. The channel process introduces random delay and/or bit error while transferring frames. Define your own frame format or you may use IEEE 802.3 Ethernet frame format.

Design: Network communication is simulated through UDP client-server . Programms are written using C++ . Each of these , i.e. channel , sender and reciever has their own port to listen .

- *Channel*: It recieves any frame sent by other nodes. Then it injects burst error randomly (only for 5% frames. Rest of the frames are kept intact). Then it looks at the header section of the frame and finds destination port address. If the port address is changed (due to error injection) and the new destination port is out of network(that is , I didn't run any node with that port) , then the frame is dropped. Otherwise the frame is sent to that port address. This procedure is followed for each packet in a separate thread.
- *Sender*: It first takes a file name from user. Opens it in binary mode. Then reads **64B** data to make frames. It also puts an additional header structure at the beginning. It's discussed later. Frame is sent following either of the above mentioned protocols. For each sender , there should be one reciever using the same protocol. There's address (port number) is hard coded in the programme. Then it waits for acknowledgement or resends frame after certain timeout.
- *Reciever*: It primarily waits for incoming data. Before starting the server, it asks user output file name. File extension should be same as sent by sender. When packets arrives, it checks the seq no. and checks if its corrupted or not. If seqno matches and frame not corrupted, it extracts the data, and saves in the given file. Then it sends acknowledgement.

Input/Output : 64B or 512bit data is read from the file to prepare the packet. An additional header object is kept in the beginning of hte frame:

```
struct DataHeader{
    char startByte; // 0b01010101
    int sourcePort;
    int destPort;
    NetworkDataType type;
    long long seqNo;
    int length; // Length of actual data + 32 bit crc flag
};
```

NetworkDataType is defined as following enum:

```
enum NetworkDataType{ RAW_DATA, ACK, NCK, COMPLETION_ACK };
```

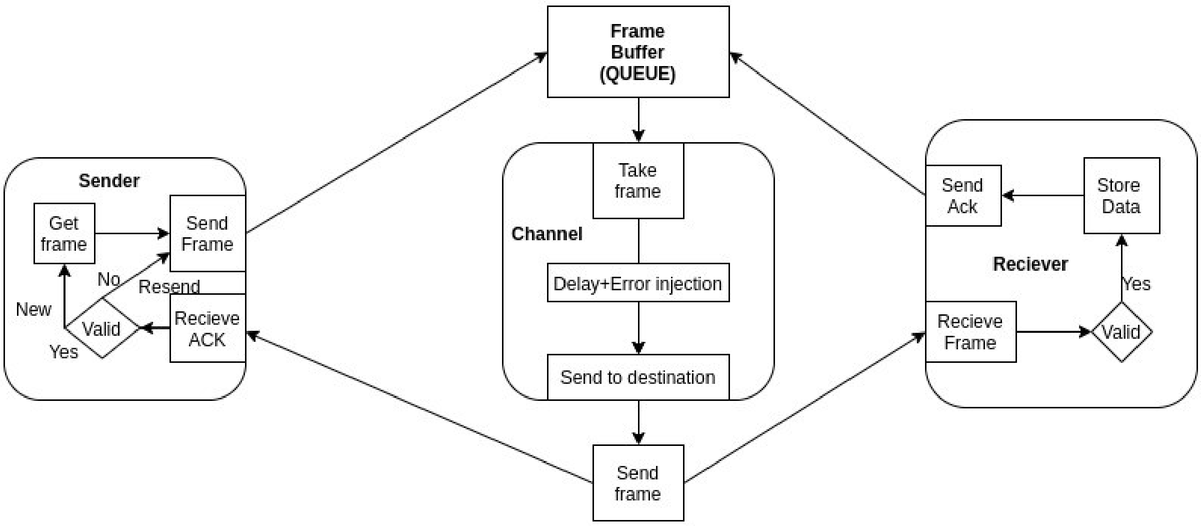
Frame format :

32 Byte Data header | 64 Byte data | 32 bit CRC

Total 100 Byte frame.

N.B. - UDP protocol allows 65kB max size for the frame.

Diagram:



Flow of data depends on underlying protocol implemented by both sender and reciever. Channel can handle multiple node and creates new thread for each arrived frame. So frames doesn't collide or wait.

Note: Delay is introduced in each node just before sending the frame to channel. There's no delay introduced by channel in this case. This is done to maintain sequential transmission of data packets. Though they can drop in the channel due to error injection.

Implementation:

File Structure:

./header:
error_delay.hpp errorDetection.hpp flowcontrol.hpp networknode.hpp

./lib:
channel gbn_reciever GBN_reciever.cpp GBN_sender PlatformChannel.cpp sendData2.jpg srsoutput.jpg SRS_reciever.cpp SRS_sender swoutput.jpg SW_reciever sw_sender
gbnoutput.jpg GBN_reciever gbn_sender GBN_sender.cpp sendData1.jpg sendData3.jpg srs_reciever srs_sender SRS_sender.cpp sw_reciever SW_reciever.cpp SW_sender.cpp

error_delay.hpp :

It includes two function:

- *injectErrorRandom():* It takes the frame as a reference object. Then flips the bit randomly. It injects error only 5% cases. It flips at least one it and at most 25% bits.

- *RandomDelay():* It makes the current thread sleep for a certain micro second of time. This delay is at most 5000 microsecond. Distribution is uniform.

errorDetection.hpp :

It includes only one class with two public static function:

- *CRC::encodeData():* It takes the frame (header + binary data) as argument and genearates 32 bit remainder using the polynomial ($x^{32} + x^{31} + x^{30} + x^{28} + x^{27} + x^{25} + x^{24} + x^{22} + x^{21} + x^{20} + x^{16} + x^{10} + x^{9} + x^{6} + 1$). It pads the redundant bits at the end and returns new frame.

- *CRC::hasError():* It takes the frame (with redundant bits) and determines if it has error. Returns true /false based on the result.

networknode.hpp :

It contains all necessary function related to establishing udp node (it can both send and recieve), DataHeader as mentioned before and NetworkDataType enums.

- *MakeHeader():* It takes source node's port number, destination node's port number, actual data length in bits, data type (NetworkDataType) , and sequence number. It feels the data in a structure and returns the structure.
- *Class ComputerNode:* It contains the node's listening port number and methods to send and recieve data.
    - *SendData():* It takes the complete frame (with header and crc bits) and an default arguement sendToChannel(default true). If sendToChannel is true, then frame is sent to Channel node despite their destination address written in the header. Channel then sends it to the actual destination.
    - *RecieveData():* It takes a refference to buffer and an default arguement blocking(=false). If blocking is false and no data is available , it returns immediately with return value -1. Otherwise , it blocks the current thread untill new data is available. It returns number of bytes read and -1 if error occurs.

flowcontrol.hpp :

It defines two abstract class and necessary methods : SenderNodeFlow and RecieverNodeFlow

- *SenderNodeFlow:* It stores various state variables of a sender node that includes: port address of the node, destination port, input file, a map of seq no to corresponding frame etc. All the following methods are protected:
    - *GetData():* It reads 64B data and puts in an array (vector).
    - *MakeFrame():* It takes previously read data , adds header and then encode using CRC. Stores the complete frame in another variable.
    - *StoreFrame():* It takes seq no as an arguement and maps the frame (made in the previous step) to the seq no. It's stored to be resent in case of timeout.
    - *PurgeFrame():* It erase a frame corresponding to given seq no.
    - *sendFrame():* It finally sends the frame associated with given seq no . It also introduces randomDelay in the thread. It uses an ComputerNode object previously created.
    - *RecvFrame():* It recieves frames as acknowledgement and stores in an array (protected variable).

- ***Run()***: It’s an abstract function with no defination. Child class must implement it according to protocol.

- *RecieverNodeFlow:* It stores various state variables of a reciever node that includes: port address of the node,   destination port (that is sender node) , input file, a map of seq no to corresponding frame etc. All the following methods are protected:
  - *sendFrame():* It sends acknowledgment frame or Not acknowledged frame.
  - *RecieveFrame():* It recieves frame. Blocking state can be controlled by providing an optional arguement.
  - *ExtractData():* It removes the header and CRC redundant bits and extracts actual data.
  - *DeliverData():* It writes the actual data to the output file.
  - **Run():** It’s an abstract function. Child class must implement it following the protocols.

N.B. - All above libraries uses system dependent APIs. That is, these program should be run on a linux machine.

Driving programmes:

- SW_sender.cpp: It implements, Stop and Wait protocol for sender side. It basically waits for three events : send new frame or request, recive acknowledgement when arrives, and resend frames when timeout occurs. It uses modulo 2 addition to increament the sequence number. It defines StopNWaitSender class inheriting SenderNodeFlow class and overrides run() as follows:

```
void run(){
        sn = 0;
        canSend = true;
        totalConsecutiveTimeout = 0;

        thread tsend(&StopNWaitSender::sendNewFrame, this);
        thread trecieve(&StopNWaitSender::recieveAck,this);
        thread ttimeout(&StopNWaitSender::handleTimeOut,this);

        tsend.join();
        trecieve.join();
        ttimeout.join();

}
```

  Here, we also count number of consecutive timeouts. If timeout event occurs 30 times in a row, we assume that reciever is down somehow. So we stop sending data and exit the program. Upper mentioned 3 events are defined as follows:

sendNewFrame() is as follows:

```
void sendNewFrame(){
   while(true){
       std::unique_lock<std::mutex> lk(mut);
       if(!eventRequestToSend){
           cout<<"Seccessfully transmitted!\n";
           exit(0);
       }

        //Wait untill the condition becomes true
       data_cond.wait(
           lk, [this]{return (eventRequestToSend && canSend);});

       getData();
       makeFrame(sn);
       storeFrame(sn);
       sendFrame(sn);
       startTimer();
       sn = (sn + 1) % MODULO;
       canSend = false;

       totalConsecutiveTimeout = 0;

       lk.unlock(); //Unlock resources for other threads to use
   }
}
```

recieveAck() is implemented as follws:

```
void recieveAck(){
        while(true){
            if (!eventRequestToSend){
                cout << "Seccessfully transmitted!\n";
                exit(0);
            }

             // Blocking socket is used to recive frame
            if (recvFrame(true) > 0){
                totalConsecutiveTimeout = 0;
                DataHeader h;
                int status = extractAck(h);

                //Frame not corrupted and ack==Sn
                if (status == sn){
                    {
                        std::lock_guard<std::mutex> lk(mut);
                        stopTimer();
                        purgeFrame((sn - 1 + MODULO) % MODULO);
                        canSend = true;
                        if (h.type == COMPLETION_ACK){
                            eventRequestToSend = false;
                        }
                    }

                     // Recieve sending thread to send new frame
                    data_cond.notify_one();
                }
            }
        }
    }
```

handleTimeout() is as follows:

```
void handleTimeOut(){
        while(true){
            std::lock_guard<std::mutex> lk(mut);

            if (isTimeOut()){
                startTimer();
                sendFrame((sn - 1 + MODULO) % MODULO);
                totalConsecutiveTimeout++;

                if (totalConsecutiveTimeout > 30){
                    cout << "Failed to recieve response 30 times consecutively!\nAborting programm!\
n";

                    exit(1);
                }

                std::this_thread::sleep_for(75ms);
            }
        }
    }
```

- SW_reciever.cpp:- It implements reciever side algorithm for Stom n Wait protocol. It runs a single thread. It waits for a frame ad if the seq no matches, it sends back an acknowledgement. If only a header object is recieved with type COMPLETION_ACK, then that means sender has completed transmission and reciever can close the output file . Algorithm is implemented in the run() of StopNWaitReciever class which inherits above mentioned RecieverNodeFlow. It’s as follows:

```
  void run(){
        sn = 0;
        while (eventRequestToRecieve){
            int i = recvFrame(true);
            if (i >= 0 && !CRC::hasError(frame)){
                cout << "Recieve status: " << i << endl;

                DataHeader h;
                extractData(h);
                if (h.seqNo == sn){
                    deliverData();
                    sn = (sn + 1) % MODULO;
                }

                // Send acknowledgement
```

```
            sendFrame(sn);
            if (h.type == COMPLETION_ACK) eventRequestToRecieve = false;
        }

    }
}
```

- GBN_sender.cpp: This defines GoBackNSender class (that inherits SenderNodeFlow) and implements GoBackN protocol's sender side algorithm in run() method. It also runs three concurrent threads that raise three events: send new frame, recieve acknowledgement, and timeout. Synchronization between them is done using internal variables and mutex lock. Here we consider m=4 or window length of 2^m-1 = 15 packets. We used twopointer to mark the window : sw and sn. Sw points to first not-acknowledged frame and sn points to first frame that is not sent yet. Sw is increamented only when and an ack arrives (where sw<ack<=sn) . Now , an unique situation occurs: increament is done here by modulo 16 addition . So , after recieving frame with seq no = 15, reciever will send an ack = 0 . Which is out of sender side window. So, we assume that frames ones sent, will either always reach sequentially, or it will be dropped by channel. That is , say seq no 1,2,3 are sent, then they arrives at that order or some may get droppped (say 1 and 3 arrives, 2 get dropped) but the order remains same , or it will never happen that 3 arrives before 2. Thus if ack=0 arrives , we can assume safely that reciever succesfully recieved all the frames upto 15 and we start sf from 0 again. This is implemented in recieveAck() thread. Here also three threads maintain the flow and sychronizes using internal variables and mutex. These are ass follows:

sendNewFrame():

```
void sendNewFrame(){
    while (true){
        std::unique_lock<std::mutex> lk(mut);
        if (!eventRequestToSend){
            cout << "Seccessfully transmitted!\n";
            exit(0);
        }

        data_cond.wait(
            lk, [this]{return (eventRequestToSend && canSend);});

        getData();
        makeFrame(sn % MODULO);
        storeFrame(sn % MODULO);
        sendFrame(sn % MODULO);
        sn = (sn + 1);
        startTimer();

        if(sn % MODULO == 0){
            canSend = false;
        }

        totalConsecutiveTimeout = 0;

        lk.unlock();
        cout << "Sending new frame, sf:" << sf << "  sn:" << sn << " sw:" << sw << "\n";
    }
}
```

recieveAck():

```
void recieveAck(){
        while (true){

            if (!eventRequestToSend){
                cout << "Seccessfully transmitted!\n";
                exit(0);
            }

            if (recvFrame(true) > 0){
                totalConsecutiveTimeout = 0;
                DataHeader h;
                int ackNo = extractAck(h);

                if(ackNo <= -1) continue;//Corrupted frame
```

```
                cout << "Recieved ack: " << ackNo << "\n";

                //Frame not corrupted and valid ack
                if ((ackNo + sf) <= (sn)){
                    {
                        std::lock_guard<std::mutex> lk(mut);
                        cout << "sf: " << sf << "  " << MODULO << "    " << (sf % MODULO) << "\n";
                        cout << "Recieved__ ack: " << (ackNo) << "\n";

                        while ((sf % MODULO) < ackNo){
                            cout << "Purging frame: " << (sf % MODULO) << " " << sf << "\n";
                            purgeFrame(sf % MODULO);
                            sf = (sf + 1);
                            stopTimer();
                        }

                        if (ackNo == 0 && storage.find(0) == storage.end()){
                              //Previous batch all acknowledged
                            canSend = true;
                            sf = sn;
                            storage.clear();
                        }

                        if (h.type == COMPLETION_ACK){
                            eventRequestToSend = false;
                        }
                    }
                    // Notify other threads
                    data_cond.notify_one();
                }
            }
        }
    }
```

handleTimeout():

```
void handleTimeOut(){
        while (true){

            {
                std::lock_guard<std::mutex> lk(mut);
                if (isTimeOut()){
                    startTimer();

                    int temp = sf;
                    while (temp < sn){
                        sendFrame(temp % MODULO);
                        temp = (temp + 1);
                        cout << "Re-Sending new frame, sf:" << sf << " sn:" << sn << endl;
                    }
                    totalConsecutiveTimeout++;

                    if (totalConsecutiveTimeout > 30){
                        cout << "Failed to recieve response 30 times consecutively!\nAborting
programm!\n";
                        exit(1);
                    }

                }
            }

            std::this_thread::sleep_for(150ms);
        }
    }
```

- GBN_reciever.cpp: It defines GoBackNReciever class which inherits RecieverNodeFlow. Run() method implements reciever side algorithm for Go back N protocol. It's also single threaded. It's as follows:

```
void run(){
        sn = 0;
        long long j = 0;
        while (eventRequestToRecieve){
            cout << "Iteration count: " << (j++) << endl;
            cout << "err: " << errno << endl;

            int i = recvFrame(true);
            if (i >= 0 && !CRC::hasError(frame)){

                DataHeader h;
                extractData(h);
                cout<<"Recieved seq: "<<h.seqNo<<endl;

                if (h.seqNo == (sn % MODULO)){
```

```
                deliverData();
                sn = (sn + 1) ;
            }

            sendFrame(sn % MODULO);  //send acknowledgement
            cout<<"sneding ack: "<<sn << " - "<< (sn%MODULO) <<endl;
            if (h.type == COMPLETION_ACK){
                eventRequestToRecieve = false;
                cout<<"COMPLETEION_ACK recieved, terminationg the program\n";
            }
        }
    }

}
```

- SRS_sender.cpp: It defines SelectiveRepeatSender class inheriting senderNodeFlow. It uses un ordered map to store frame and timer data corresponding to different seq no. Here also m=4 or window length = 2^{m-1} = 8. Similar to GoBackN protocol, here also two pointers are used to mark the current window. Sf points to first of not-acknowledged frame and sn points to first not sent frame. If reciever sends an ack that is out of window , then we assume that all the previously frames are recieved successfully, as explained earlier. It also runs 3 concurrent threads and synchronizes them using internal variables and mutex. These are as follows:

sendNewFrame():

```
void sendNewFrame(){
        while (!makingNewFrameComplete){
            std::unique_lock<std::mutex> lk(mut);
            if (!eventRequestToSend){
                cout << "Seccessfully transmitted!\n";
                exit(0);
            }

            // wait untill condition becomes true
            data_cond.wait(
                lk, [this]{return (eventRequestToSend && canSend);});

            getData();
            makeFrame(sn);
            storeFrame(sn);
            sendFrame(sn);
            startTimer(sn);
            sn = (sn + 1) % MODULO;

            // when sf>8, sn becomes 0<=sn<8 due to modulo adddition
            if ((sn - sf == sw) || (MODULO - sf + sn) == sw){   // that is |sf-sn| == sw
                canSend = false;
            }

            totalConsecutiveTimeout = 0;

            lk.unlock();
        }
    }
```

recieveAck():

```
void recieveAck(){
        while (true){
            if (!eventRequestToSend){
                cout << "Seccessfully transmitted!\n";
                exit(0);
            }

            if (recvFrame(true) > 0){
                totalConsecutiveTimeout = 0;
                DataHeader h;
                int ackNo = extractAck(h);
                if (ackNo < 0) continue;//Corrupted frame

                if(h.type == NCK && storage.find(ackNo)!=storage.end()){
                    std::lock_guard<std::mutex> lk(mut);
                    cout<<"Recieved NCk: "<<ackNo<<endl;
                    sendFrame(ackNo);
                    startTimer(ackNo);
                    continue;
                }

                //Frame not corrupted and valid ack
                // sf<ack<16
                if (sf < ackNo && (ackNo <= sn || ackNo < (sf+sw))){
```

```
                    // if ((ackNo > (sf % MODULO)) && (ackNo <= (sn % MODULO))){
                    {
                        std::lock_guard<std::mutex> lk(mut);
                        cout << "Recieved__ ack: " << (ackNo) << "\n";

                        while (sf < ackNo){
                            cout << "Purging frame: " << (sf) << "\n";
                            purgeFrame(sf);
                            stopTimer(sf);
                            sf = (sf + 1) % MODULO;
                            if(sf==0) break;
                        }

                        if (sn - sf < sw){
                            canSend = true;
                        }

                        if (h.type == COMPLETION_ACK){
                            eventRequestToSend = false;
                        }
                    }
                    data_cond.notify_one();
                }
                // 0<=ack<sn , i.e. when sf=11,sn=3 and ack = 1
                else if (ackNo <= sn && sf > sn){
                    // Reciver window slided to the next frame slot, therefore, ack < Sf
                    {
                        std::lock_guard<std::mutex> lk(mut);
                        cout << "Recieved__ ack: " << (ackNo) << "\n";

                        while (sf < MODULO){
                            cout << "Purging frame: " << (sf)<< "\n";
                            purgeFrame(sf);
                            stopTimer(sf);
                            sf = (sf + 1) % MODULO;
                            if (sf == 0) break;
                        }

                        //Now sender window also arived at next slot
                        while (sf < ackNo){
                            cout << "Purging frame: " << (sf)  << "\n";
                            purgeFrame(sf);
                            stopTimer(sf);
                            sf = (sf + 1) % MODULO;
                        }

                        if (sn - sf < sw){
                            canSend = true;
                        }

                        if (h.type == COMPLETION_ACK){
                            eventRequestToSend = false;
                        }
                    }
                    data_cond.notify_one();
                }
            }
        }
    }
```

handleTimeout():

```
void handleTimeOut(){
        while (eventRequestToSend){
            vector<int> tout = isTimeOut();
            for (int f : tout){
                std::lock_guard<std::mutex> lk(mut);
                startTimer(f);
                sendFrame(f);
                cout << "Re-Sending new frame, f:" << f  << endl;
            }

            if (tout.size()) totalConsecutiveTimeout++;

            if (totalConsecutiveTimeout > 30){
                cout << "Failed to recieve response 30 times consecutively!\nAborting programm!\n";
                exit(1);
            }

            std::this_thread::sleep_for(50ms);
        }
    }
```

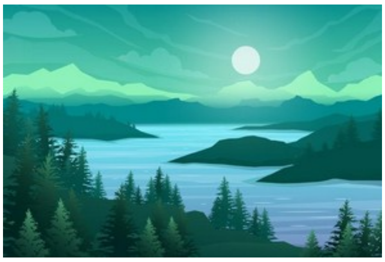
- SRS_reciever.cpp: It defines SelectiveRepeatReciever class inheriting RecieverNodeFlow. It implements run() function as follows:

```
void run(){
     sn = 0;
     long long j = 0;
     nackSent = false;
     ackNeed = false;
     frameWindow.clear();

     while (eventRequestToRecieve){
         cout << "Iteration count: " << (j++) << endl;

         int i = recvFrame(true);

         bool corruptedFrame = CRC::hasError(frame);

         if (i >= 0 && corruptedFrame){
             sendFrame(sn,true); //send NACK
             nackSent = true;
             continue;
         }

         if (i >= 0 && !corruptedFrame){

             DataHeader h;
             extractData(h);
             cout << "Recieved seq: " << h.seqNo << endl;

             if(h.seqNo != sn ){
                 sendFrame(sn,true); //sendNAck
             }

             if(withinWindow(h.seqNo) && frameWindow.find(h.seqNo)==frameWindow.end()){
                 frameWindow[h.seqNo] = data; //store and mark the seqNo

                 while(frameWindow.find(sn)!=frameWindow.end()){
                     deliverData(sn);
                     frameWindow.erase(sn);
                     sn = (sn+1)%MODULO;
                     ackNeed = true;
                 }

                 if(ackNeed){
                     sendFrame(sn);
                     ackNeed = false;
                     nackSent = false;
                 }

                 if (h.type == COMPLETION_ACK){
                     eventRequestToRecieve = false;
                     cout << "COMPLETEION_ACK recieved, terminationg the program\n";
                 }
             }
         }

     }
 }
```

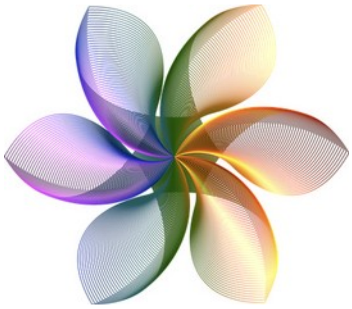
Test cases:

Following files are sent by sender algorithm and rebuilt by reciever side algorithm.

sendData*.jpg is used as input file in all three cases. These are folowing:



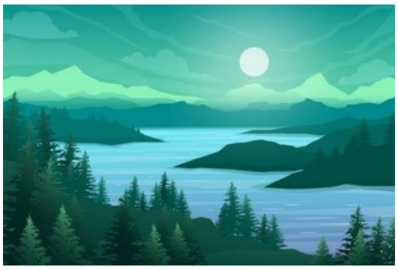
sendData1.jpg – 43.5kB



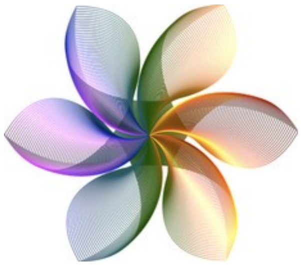
sendData2.jpg – 162.8kB

sendData3.jpg – 9.9kB

These are recieved as follows:



swoutput.jpg(100% transmission)



srsoutput.jpg(100%transmission)

gbnoutput.jpg(100% transmission)

Reuslts and Analysis:

Performance metrics- Receiver Throughput (packets per time slot), RTT, bandwidth-delay product, utilization percentage.

Delay and errors are introduced as mentioned earlier.

Few notes:-

- Available bandwidth: Bandwidth is measured by max data sent from one node to another in 1s. Now for UDP, packet size is limited 65kB. After introducing random delay manually, we get 94.22 Mbps bandwidth. But, in this case, we have taken max frame size as 100Bytes. So we'll consider *maximum bandwidth: **148.66 kbps.***
- *RTT is assumed same for all, Or RTT is considered independent of the protocol used by either sender/reciever. It is calculated in StopNWait protocol. Consequtively , approx delay is taken as half of RTT .*

Results:

- Following are almost same for all protocols. These measures are taken during StopNWait protocol testing.
  - **RTT**: 10100 micro seconds or **0.0101s.**
  - **Delay:** Delay of data to reach destination from source = RTT/2 = 5050 micro seconds.
  - **Bandwidth-Deay product**: Max bandwidth * delay = ~751 bits
- Stop and Wait protocol:

    - Reciever throughput: 3.808 kbps
    - Bandwidth utilization: (Reciever throughput / max bandwidth) = ~ 2.6%
- Go-Back-N protocol:
    - Reciever throughput: 71.409 kbps
    - Bandwidth utilization: (Reciever throughput / max bandwidth) = ~ 48%
- Selective repeat protocol:
    - Reciever throughput: 123.7 kbps
    - Bandwidth utilization: (Reciever throughput / max bandwidth) = ~ 83%

Conclusion:

We can increase reciever throughput by decreasing number of ack/nck and caching frames which are out of order. Increasing the window size also may improve the utilization.