

Bisakh Mondal
Roll: 001810501079 (A3)
BCSE-III
Computer Networks
Assignment-II (Flow Control)

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

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.

Deadline: 28 November 2020

Submission: 28 November 2020

DESIGN:

The purpose of the program is to simulate an end to end communication by transferring a byte stream between sender and receiver through the channel by mainly three protocols.

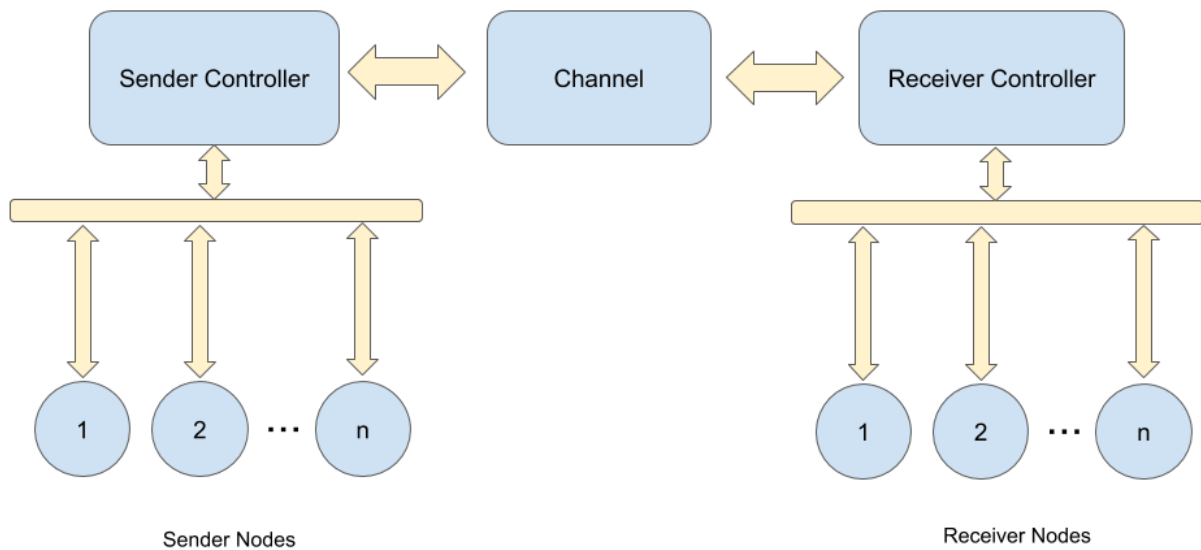
1. STOP N WAIT ARQ
2. GO BACK N ARQ
3. SELECTIVE REPEAT ARQ

To discuss the basic design principle I have created 2 Controller processes and one channel process which is responsible for providing a connection between two controller processes.

- Sender Controller
- Receiver Controller

Both receiver controllers and sender controllers are spawning N number of child processes/ nodes which are responsible for node to node communication.

The controllers are higher-level abstractions in **Data Link** layers like real-world scenarios where the sender and receiver nodes spawned from the controller can be regarded as connected devices like mobiles, laptops, and the controllers are the **router**. To be honest the controller is responsible for transmitting an Ethernet frame to the channel received from the sender node and returning the Acknowledgement frame received through the channel.



The **channel** is another higher-level abstraction considered here which is responsible for the host to host delivery by traversing the required path by itself internally. Due to which it can add a certain amount of delay or add noises while transmitting ethernet frames.

Input format: In practical cases, the data link layer gets datagram packets from the upper layer(Network layer). But for ease of simulation, I'm considering data of integer format converted to binary stream to be used for transmission(available in constants.hpp).

Output Format: It shows the ongoing transmission in terminal output and a message when the transmission is done.

IMPLEMENTATION:

Here, based on the proposed design, the whole problem heavily depends on IPC or interprocess communication.

In this context, I have used two types of interprocess communication here,

- **Shared Memory**
- **Named Pipes (FIFO)**

The program also hugely depends on the performance of the lightweight C++ threads used for faster communication between a particular sender and receiver nodes no matter what works it is currently performing.

- **DATA FORMAT**

Here for better generalization and uniformity, **IEEE 802.3** frame format has been used throughout the communication. The fields are

Preamble + SFD + DEST_MAC + SRC_MAC + SEQ_NO(custom addition) + DATA + CRC. 7 + 1 + 6 + 6 + 1 + 46 + 4.

Each data packet is encoded with **CRC32** for avoiding transmission error.

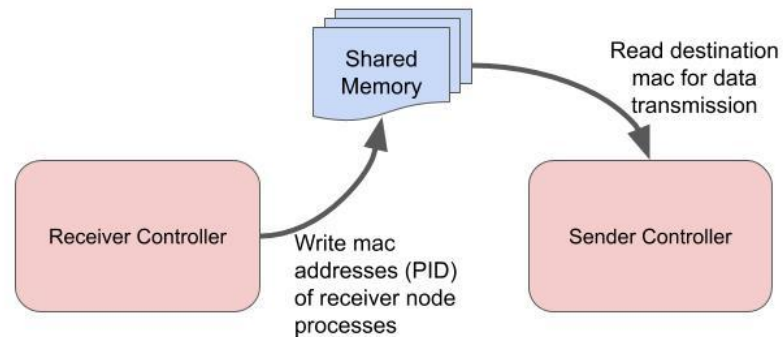
Preamble 7 Bits	SFD 1 Byte	Dest Mac 6 Byte	SRC Mac 6 Byte	Seq No 1 Byte	Data 46 Bits	CRC 4 Byte
--------------------	---------------	--------------------	-------------------	------------------	-----------------	---------------

Augmented IEEE 802.3 frame format

?? Now a serious question may arise what is the mac address here and how am I getting the destination address during transmission.

While spawning the N number of nodes which are working as a receiver in the receiver controller, the process IDs are the identification of each node in both sender and receiver side.

The receiver controller writes the receiver node PIDs in shared memory and the sender controller reads if first and while spawning sender nodes provide the read PIDs to each sender to consider the PIDs as mac addresses during transmission of data packets.



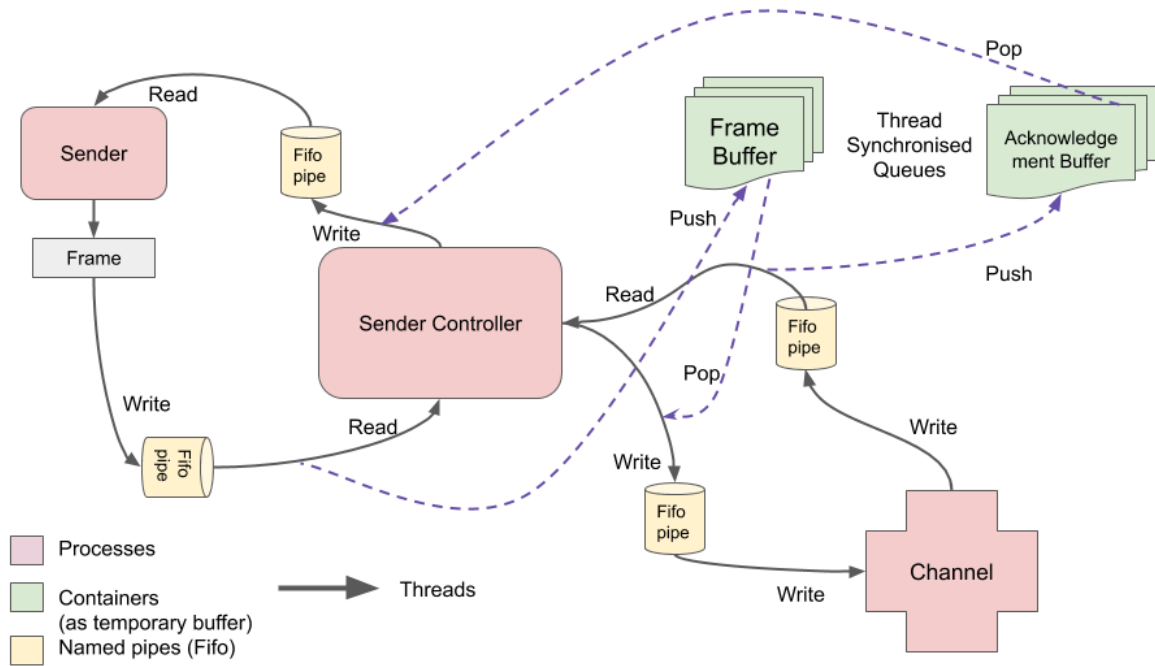
Both Sender Controller and Receiver Controller create a record for each process IDs as mac vs the FIFO pipes name as communication media. **So when a packet is received it first decodes the destination mac address and opens the named pipe and transmits the frame.**

❖ Addition of threads

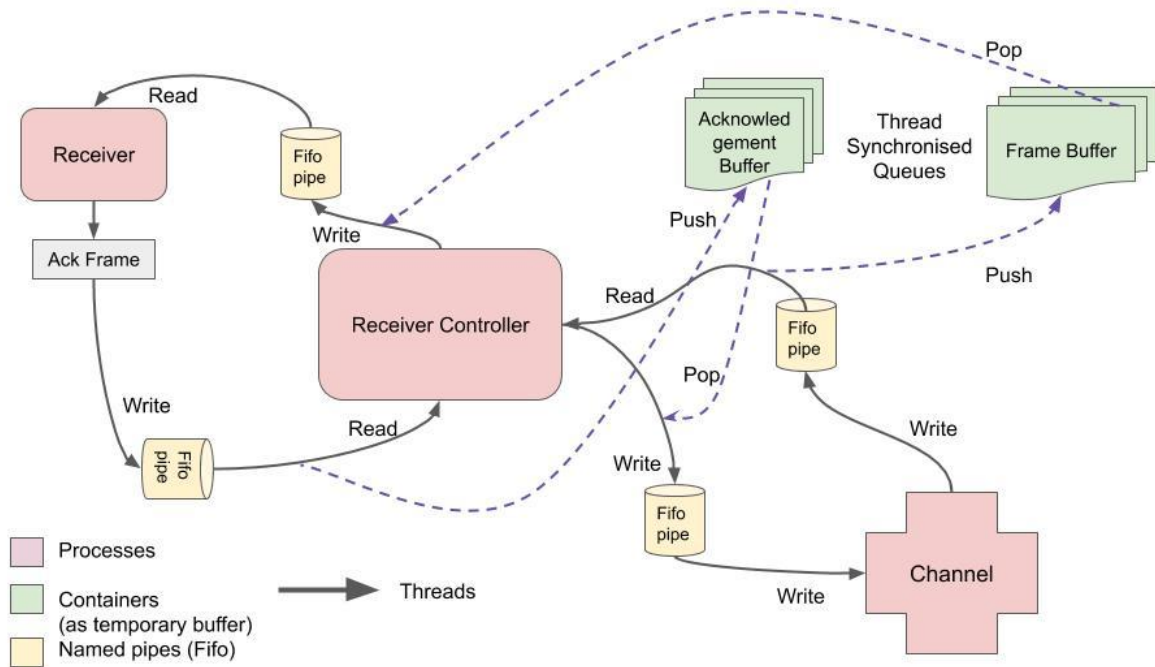
Since the simulation itself is the replication of the part of real-world data communication, implementation of each process in a single sequential program does not make any sense. Threads are lightweight, context switching is very computation friendly.

So for an N number of sender and receiver nodes,

- Each sender Node/process uses a minimum of **two threads**.
- Each receiver Node uses a minimum of two threads.
- Sender Controller (and Receiver Controller) uses **N+3** nodes, **N nodes assigned to read ethernet frames from each node, 1 node to transmit received acknowledgement, and 2 threads to write and read to and from the channel respectively.**
- The Channel uses 4 threads, **2 dedicated for the sender controller and receiver controller each** for reading and writing.

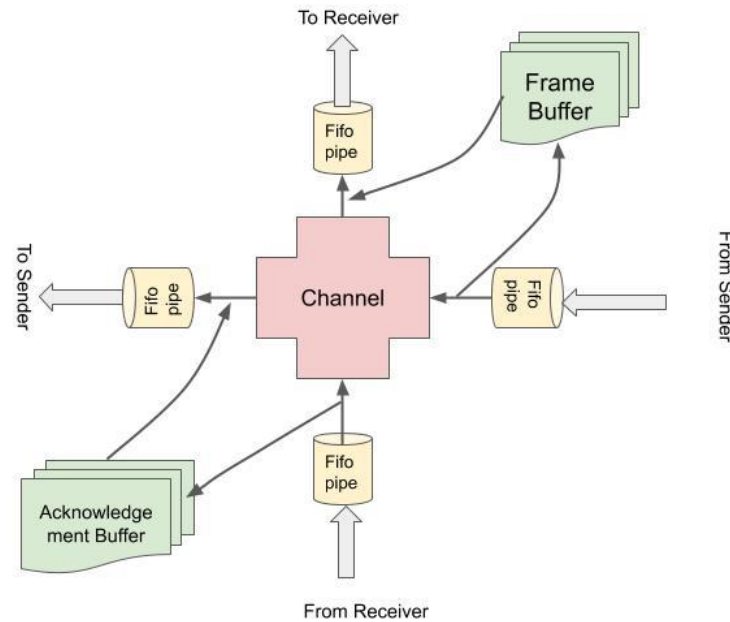


A Single sender, Sender controller with channel involved communication diagram.



→ The Channel

It comprises two **syncd buffer** and **4 threads** and is responsible for two way communication between two controllers.



→ Thread Safe Queue as Buffer.

All operations that have been mentioned here need to maintain buffers, since holding pipe for too long to retransmit the data in controllers and channels does not make sense. But also racing is not preferred. That's why a thread-safe queue has been implemented to restrict access of concurrent threads.

```
template<class T>
class SyncdQueue{

    std::queue<T> q;
    std::condition_variable c;
    mutable std::mutex m; //thread safe mutex.

public:
    void push(T val){
        std::lock_guard<std::mutex> lock(m);
        q.push(val);
        c.notify_one();
    }
};
```

```

    }

    T pop(){
        std::unique_lock<std::mutex> lock(m);

        while(q.empty()){
            c.wait(lock); //release the lock until it gets filled
            with a value by other workers.
        }
        T popped = q.front();
        q.pop();
        return popped;
    }
    int len(){
        std::lock_guard<std::mutex> lock(m);
        return q.size();
    }
};

```

→ Shared Memory module along with Binary Semaphore:

The shared memory library in C and the semaphore library consists of low-level APIs, the following items are rewritten to avail efficient APIs throughout the program.

shared memory.hpp

```

class SharedMemory{
    int fd;
    const char* backingfile;
    int bytesize;

public:
    void* memptr;
    void* SharedMEMInit(){
        this->fd = shm_open(backingfile, O_RDWR | O_CREAT, 0644);

        if(fd<0){
            report_and_exit("shared memory initialization
failed");

```



```

        if (memptr == NULL) report_and_exit("Can't access
segment...");

        return memptr;
    }

    SharedMemory(std::string file, int Bytes, std::string
mode="init"){
        backingfile= file.c_str();
        bytesize= Bytes;
        if(mode=="init")
            this->memptr = SharedMEMInit();
        else
            this->memptr = SharedMEMOpen();
    }

    ~SharedMemory(){
        munmap(this->memptr, this->bytesize);
        close(fd);
        unlink(backingfile);
    }
};

```

The Binary Semaphore:...

```

class BinSemaphore{
    std::string name;
    int val;

    public:
    sem_t * semptr;
    sem_t* semInit(){
        sem_t* semptr = sem_open(name.c_str(), O_CREAT,
S_IRUSR|S_IWUSR, val);

        if(semptr == SEM_FAILED){
            report_and_exit("sem init failed");
        }
    }
};

```

```

    }
    return semptr;
}

sem_t* semOpen(){
    sem_t * semptr = sem_open(name.c_str(), O_EXCL, 0);
    if(semptr == SEM_FAILED){
        report_and_exit("sem open failed");
    }
    return semptr;
}

//semPOST
int semPOST(){
    return sem_post(semptr);
}

//semWAIT
int semWAIT(){
    return sem_wait(semptr);
}

int semVal(){
    int k;
    sem_getvalue(semptr, &k);
    return k;
}

//2 modes one for initialization another for opening.
BinSemaphore(std::string n, int v=0, std::string mode="init"):
name(n), val(v){
    if(mode=="init")
        this->semptr = semInit();
    else
        this->semptr = semOpen();
}

~BinSemaphore(){
    // std::cout<<"des: "<<std::endl;

```

```

        sem_unlink(name.c_str());
        sem_destroy(&semptr);
    }
};

```

→ Channel Process

As discussed earlier the channel process is only responsible for frame transmission from one end to another. It has one read and writes function, four threads on separate pipes use the methods. A random error injector injects random error by flipping the bits at a random position.

```

class Channel{
    fc::syncedBuffer_t toReceiver, toSender;
public:
    void fifoRead(string fifoname, int frameSize, bool
    bysender=true){
        MakeFiFO(fifoname);
        cout<<fifoname<<endl;
        while(true){
            fc::descriptor_t fd = open(fifoname.c_str(), O_RDONLY);
            char data[frameSize+1];
            int status;

            if((status=read(fd, data, frameSize+1))>0){

                close(fd);

                if(status==-1){
                    cerr<<"fifo read failed"<<endl;
                    continue;
                }

                if(bysender){
                    cout<<"read from sender"<<endl;
                }else{
                    cout<<"read from receiver"<<endl;
                }
                fc::bytestream_t d(data);
            }
        }
    }
};

```

```

        if(bysender){
            toReceiver.push(d);
        }else{
            toSender.push(d);
        }
    }
}

}

}

void fifoWrite(string fifoname, int framesize, bool
bysender=true){
    MakeFiFO(fifoname);
    cout<<fifoname<<endl;
    while(true){
        fc::descriptor_t fd = open(fifoname.c_str(), O_WRONLY);

        fc::bytestream_t data;
        if(bysender){
            data = toReceiver.pop();
        }else{
            data = toSender.pop();
        }
        char d[framesize+1];
        strcpy(d, (char*)data.c_str());
//        add noises in 20%cases
        if(rand()%10>=8){
            addNoise(d, framesize);
        }
        if(write(fd, d, framesize+1)==-1){
            cerr<<"fifo write failed"<<endl;
//            if(bysender){
//                toReceiver.push(data);
//            }else{
//                toSender.push(data);
//            }
        }

        if(bysender){
            cout<<"write to receiver"<<endl;

```

```

        }else{
            cout<<"write to sender"<<endl;
            cout<<"buffer size "<<toSender.len()<<"
"<<toReceiver.len()<<endl<<endl;
        }
        close(fd);

    }

}

void addNoise(char * frame, int framesize){
    int numtimes = rand()%10;
    while(numtimes--){
        int pos = rand()%framesize;
        frame[pos] = frame[pos]=='1' ? '0':'1';
    }
}

void run(){
    thread t1(&Channel::fifoRead, this, fc::S2CFIFO,
fc::FrameSize, true);
    thread t2(&Channel::fifoWrite, this, fc::C2RFIFO,
fc::FrameSize, true);
    thread t3(&Channel::fifoRead, this, fc::R2CFIFO, fc::ACKSIZE,
false);
    thread t4(&Channel::fifoWrite, this, fc::C2SFIFO,
fc::ACKSIZE, false);

    t1.join();
    t2.join();
    t3.join();
    t4.join();
}

};

```

→ Receiver Controller

After spawning N receiver processes it writes to the shared memory and lets the sender controller be acquainted with the destination mac address.

The shared memory section..

```
void memory_Write(fc::SharedMemory &shm){
    for(auto id: receiveridlist){
        *((pid_t*)shm.memptr) = id;

        shm.memptr = (void *)((pid_t*)shm.memptr +1);
    }
}

fc::SharedMemory shm(fc::SHMBACKINGFILE, fc::ALLOCATE_BYTES);
fc::BinSemaphore sem(fc::SHMMUTEX);
//synchronus memory write using semaphore.
memory_Write(shm);
if(sem.semPOST()<0) fc::report_and_exit("sempost error");
```

At Sender Controller

```
void memory_Read(fc::SharedMemory &shmm){
    using namespace fc;

    for(int j=0;j<NUM_NODES;j++){
        pid_t k = *((int*)shmm.memptr);
        receiveridlist.push_back(k);
        cout<<k<<endl;
        shmm.memptr = (void *)((int*)shmm.memptr +1);
    }
}

{
    fc::SharedMemory shm(fc::SHMBACKINGFILE, fc::ALLOCATE_BYTES,
"open");
```

```

fc::BinSemaphore sem(fc::SHMMutex, 0, "open");

//getting the receiver mac addresses synchronously.
// cout<<"sem count" <<sem.semVal()<<endl;
if(!sem.semWait()){
    memory_Read(shm);
}

if(sem.semPost()<0) fc::report_and_exit("sempost error");
}

```

→ Controllers acquiring frames from the channel

The two member functions of the controller classes running on two separate detached threads are responsible for reading and writing to the channel.

```

//the main process will execute this. ack receiving from channel
void mainProcessC(){
//    cout<<"mainProcess to chan"<<endl;
    MakeFIFO(fc::C2SFIFO.c_str());

    while(true){
        fc::descriptor_t fd = open(fc::C2SFIFO.c_str(),
O_RDONLY);
        if(fd==-1){
            cerr<<fc::C2SFIFO<<" open error"<<endl;
        }
        char ackFrame[fc::ACKSIZE+1];
        while(read(fd, ackFrame, fc::ACKSIZE+1)>0){
//            cout<<" ack received from channel(controller)
"<<endl;

            if(!fc::frame::checkFrameIntegrity(fc::bytestream_t(ackFrame))){
                cout<<"corrupted acknowledgement frame
received...discarding..."<<endl;
                continue;
            }
        }
    }
}

```



```

        ackBuffer.push(string(ackFrame));
    }
    close(fd);
}
}

void mainProcessS(){
//    cout<<"mainProcess from chan"<<endl;
    MakeFiFO(fc::S2CFIFO.c_str());

    while(true){
        fc::descriptor_t fd = open(fc::S2CFIFO.c_str(),
O_WRONLY);
        fc::bytestream_t frame = buffer.pop();
        char ip[fc::FrameSize+1];
        strcpy(ip, (char *)frame.c_str());
        if(write(fd, ip, fc::FrameSize+1)<0){
            cerr<<"frame sending to channel failed"<<endl;
            buffer.push(frame);
        }
        close(fd);
    }
}
}

```

TEST CASES :

For testing of the three protocols, two types of test have been performed on each one

- The channel is Noiseless
- Noisy Channel.

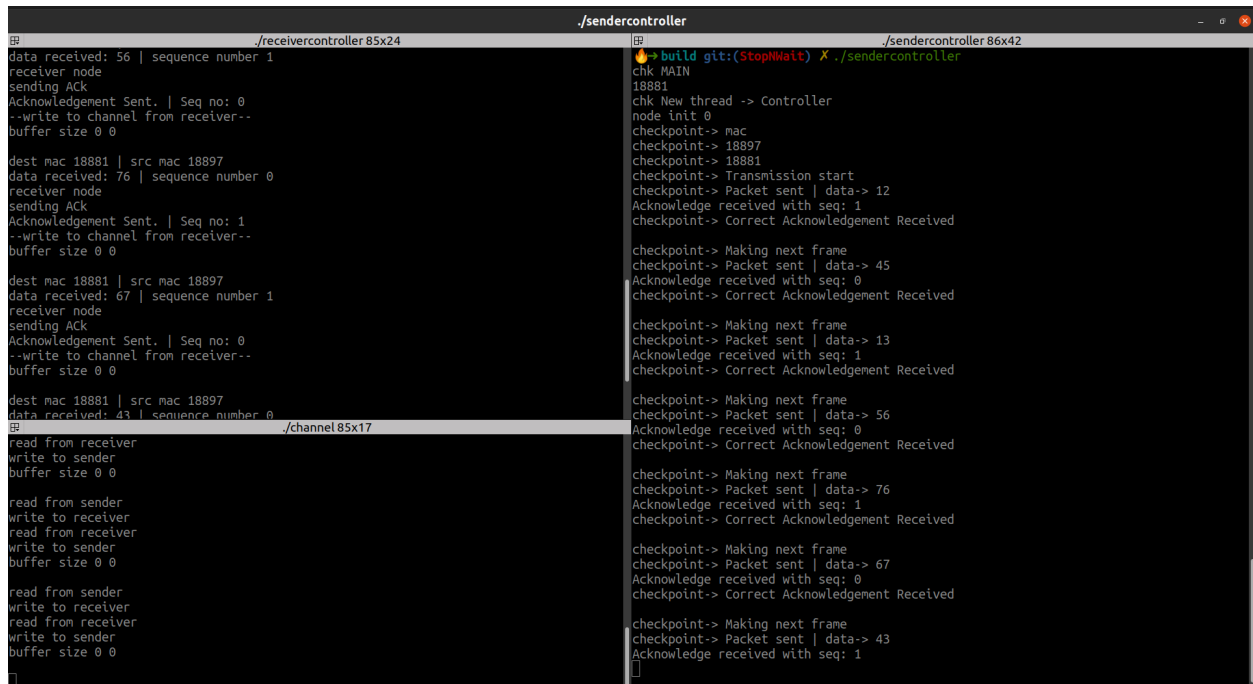
And as discussed earlier, since the data link layer is responsible for adding header and trailers (the CRC32) with the data packet received from the upper Network layer, for ease to readability and performance checking an array of integers have been transmitted.

dataArray = {12, 45, 13, 56, 76, 67, 43, 23, 15};

1. STOP N WAIT:

It is the simplest protocol among three, sends only one data packet and waits for the acknowledgement and transmits the next packet.

For *noiseless* channel:



```
./receivercontroller 85x24
data received: 56 | sequence number 1
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 0
--write to channel from receiver--
buffer size 0 0

dest mac 18881 | src mac 18897
data received: 76 | sequence number 0
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 1
--write to channel from receiver--
buffer size 0 0

dest mac 18881 | src mac 18897
data received: 67 | sequence number 1
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 0
--write to channel from receiver--
buffer size 0 0

dest mac 18881 | src mac 18897
data received: 43 | sequence number 0

./channel85x17
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver
read from receiver
write to sender
buffer size 0 0

./sendercontroller 86x42
build git:(StopWait) X ./sendercontroller
chk MAIN
18881
chk New thread -> Controller
node init 0
checkpoint-> mac
checkpoint-> 18897
checkpoint-> 18881
checkpoint-> Transmission start
checkpoint-> Packet sent | data-> 12
Acknowledge received with seq: 1
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 45
Acknowledge received with seq: 0
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 13
Acknowledge received with seq: 1
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 56
Acknowledge received with seq: 0
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 76
Acknowledge received with seq: 1
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 67
Acknowledge received with seq: 0
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 43
Acknowledge received with seq: 1
```

For *Noisy* Channel

The channel is adding noise to 20 % of its writing to both ends. The results are

```

./sendercontroller
./receivercontroller 85x24
Error in communication
corrupted ethernet frame received...discarding...
dest mac 19403 | src mac 19426
data received: 67 | sequence number 1
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 0
--write to channel from receiver--
buffer size 0 0

dest mac 19403 | src mac 19426
data received: 43 | sequence number 0
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 1
--write to channel from receiver--
buffer size 0 0

Error in communication
corrupted ethernet frame received...discarding...
Error in communication
corrupted ethernet frame received...discarding...
]

./channel 85x17
read from sender
write to receiver
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver
read from sender
write to receiver

./sendercontroller 86x42
checkpoint-> Packet sent | data-> 12
Acknowledge received with seq: 1
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 45
Acknowledge received with seq: 0
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 13
Acknowledge received with seq: 1
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 56
Acknowledge received with seq: 0
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 76
checkpoint-> Retransmitting old packet
checkpoint-> Packet sent | data-> 76
Acknowledge received with seq: 1
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 67
checkpoint-> Retransmitting old packet
checkpoint-> Packet sent | data-> 67
Acknowledge received with seq: 0
checkpoint-> Correct Acknowledgement Received

checkpoint-> Making next frame
checkpoint-> Packet sent | data-> 43
Error in communication
corrupted acknowledgement frame received...discarding...
checkpoint-> Retransmitting old packet
checkpoint-> Packet sent | data-> 43
checkpoint-> Retransmitting old packet
checkpoint-> Packet sent | data-> 43

```

2. Go BACK N

It sends a window of the frame with Sf and Sn to denote the first and last bits entry transmitted from a sliding window. If $Sf - Sn \geq 2^w - 1$ or no acknowledgement received the protocol sends all the windows to the receiver on a note to receive acknowledgement in future.

For noiseless channel:

```

./sendercontroller
./receivercontroller 85x24
data received: 43 | sequence number 6
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 7
--write to channel from receiver--
buffer size 0 0

dest mac 27011 | src mac 27038
data received: 23 | sequence number 7
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 8
--write to channel from receiver--
buffer size 0 0

dest mac 27011 | src mac 27038
data received: 15 | sequence number 8
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 9
--write to channel from receiver--
buffer size 0 0

]

./channel 85x17
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver
read from receiver
write to sender
buffer size 0 0

./sendercontroller 86x42
^C
build git:(StopMatt) X ./sendercontroller
chk MAIN
27011
chk New thread -> Controller
node init 0
checkpoint-> mac
checkpoint-> 27038
checkpoint-> 27011
checkpoint-> Transmission start
checkpoint-> Packet sent | data-> 12
checkpoint-> Packet sent | data-> 45
checkpoint-> Packet sent | data-> 13
Acknowledge received with seq: 1
checkpoint-> Packet sent | data-> 56
Acknowledge received with seq: 2
checkpoint-> Packet sent | data-> 76
Acknowledge received with seq: 3
checkpoint-> Packet sent | data-> 67
Acknowledge received with seq: 4
checkpoint-> Packet sent | data-> 43
Acknowledge received with seq: 5
checkpoint-> Packet sent | data-> 23
Acknowledge received with seq: 6
checkpoint-> Packet sent | data-> 15
Acknowledge received with seq: 7
checkpoint-> Retransmitting Frames
checkpoint-> Packet sent | data-> 15
Acknowledge received with seq: 8
checkpoint-> Retransmitting Frames
checkpoint-> Packet sent | data-> 15
Data transmission complete
Acknowledge received with seq: 9

```

For Noisy Channel

The channel is adding noise to 10 % of its writing on both ends. The results are

```
./receivercontroller 85x22
buffer size 0 0
dest mac 27775 | src mac 27824
data received: 43 | sequence number 6
receiver node
dest mac 27775 | src mac 27824
data received: 67 | sequence number 5
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 6
--write to channel from receiver--
buffer size 0 0

dest mac 27775 | src mac 27824
data received: 43 | sequence number 6
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 7
--write to channel from receiver--
buffer size 0 0

./channel
fifos git:(StopNWait) X cd ..
build git:(StopNWait) X ./sendercontroller
chk MAIN
27775
chk New thread -> Controller
node init 0
checkpoint-> mac
checkpoint-> 27824
checkpoint-> 27775
checkpoint-> Transmission start
checkpoint-> Packet sent | data-> 12
checkpoint-> Packet sent | data-> 45
checkpoint-> Packet sent | data-> 13
Acknowledge received with seq: 1
checkpoint-> Packet sent | data-> 56
Acknowledge received with seq: 2
checkpoint-> Packet sent | data-> 76
Acknowledge received with seq: 3
checkpoint-> Packet sent | data-> 67
checkpoint-> Packet sent | data-> 43
Acknowledge received with seq: 4
checkpoint-> Retransmitting Frames
checkpoint-> Packet sent | data-> 76
checkpoint-> Packet sent | data-> 67
checkpoint-> Packet sent | data-> 43
Acknowledge received with seq: 5
checkpoint-> Packet sent | data-> 23
checkpoint-> Packet sent | data-> 15
Error in communication
corrupted acknowledgement frame received...discarding...
checkpoint-> Retransmitting Frames
checkpoint-> Packet sent | data-> 43

./channel 85x19
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver
read from sender
write to receiver
read from receiver
```

3. Selective Repeat

This protocol incorporates the goodness of the Go back N at sender side but at the receiver side it updates a few changes, which includes sending negative acknowledgement when a frame is received whose sequence number is greater than the desired. The sender itself retransmits frames on which timeout occurred.

For noiseless channel:

```
build git:(StopNWait) X ./receivercontroller
11381
receiver node
dest mac 11381 | src mac 11391
dest mac 11381 | src mac 11391
dest mac 11381 | src mac 11391
data received: 12 | sequence number 0
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 1
--write to channel from receiver--
buffer size 0 0

dest mac 11381 | src mac 11391
data received: 45 | sequence number 1
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 2
--write to channel from receiver--
buffer size 0 0

dest mac 11381 | src mac 11391
data received: 13 | sequence number 2
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 3
--write to channel from receiver--

fifos/sender2channel
fifos/receiver2channel
fifos/channel2receiver
fifos/channel2sender
read from sender
read from sender
write to receiver
write to receiver
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver

build git:(StopNWait) X ./channel
build git:(StopNWait) X ./sendercontroller
chk MAIN
10423
chk New thread -> Controller
node init 0
checkpoint-> mac
checkpoint-> 10432
checkpoint-> 10423
checkpoint-> Transmission start
checkpoint-> Packet sent | data-> 12
checkpoint-> Packet sent | data-> 45
Acknowledge received with seq: 1
checkpoint-> Packet sent | data-> 13
Acknowledge received with seq: 2
checkpoint-> Packet sent | data-> 56
Acknowledge received with seq: 3
checkpoint-> Packet sent | data-> 76
Acknowledge received with seq: 4
checkpoint-> Packet sent | data-> 67
Acknowledge received with seq: 5
checkpoint-> Packet sent | data-> 43
Acknowledge received with seq: 6
checkpoint-> Packet sent | data-> 23
Acknowledge received with seq: 7
Data transmission complete
```

For noisy channel:

The channel is adding noise to 10 % of its writing on both ends. The results are

```

dest mac 11381 | src mac 11391
data received: 45 | sequence number 1
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 2
--write to channel from receiver--
buffer size 0 0

dest mac 11381 | src mac 11391
data received: 13 | sequence number 2
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 3
--write to channel from receiver--
buffer size 0 0

Error in communication
corrupted ethernet frame received...discarding...
dest mac 11381 | src mac 11391
data received: 56 | sequence number 3
receiver node
sending ACK
Acknowledgement Sent. | Seq no: 4
--write to channel from receiver--
buffer size 0 0

Error in communication
corrupted ethernet frame received...discarding...

```

```

fifos/receiver2channel
fifos/sender2channel
read from sender
read from sender
write to receiver
write to receiver
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver
read from receiver
write to sender
buffer size 0 0

read from sender
write to receiver
read from sender
write to receiver
buffer size 0 0

```

```

checkpoint-> mac
checkpoint-> 11245
checkpoint-> 11235
checkpoint-> Transmission start
checkpoint-> Packet sent | data-> 12
checkpoint-> Packet sent | data-> 45
checkpoint-> Packet sent | data-> 13
*-> build git:(StopNWait) x ./sendercontroller
chk MAIN
11381
chk New thread -> Controller
node init 0
checkpoint-> mac
checkpoint-> 11391
checkpoint-> 11381
checkpoint-> Transmission start
checkpoint-> Packet sent | data-> 12
checkpoint-> Packet sent | data-> 45
checkpoint-> Packet sent | data-> 13
Acknowledge received with seq: 1
checkpoint-> Packet sent | data-> 56
Acknowledge received with seq: 2
checkpoint-> Packet sent | data-> 76
Acknowledge received with seq: 3
checkpoint-> Packet sent | data-> 67
checkpoint-> Packet sent | data-> 43
Acknowledge received with seq: 4
checkpoint-> Packet sent | data-> 23

```

ANALYSIS

- The Stop N Wait ARQ is the simplest method among all, easy to implement yet inefficient in practical cases.
- The synchronisation is an issue during the whole implementation. So using **mutexes** and semaphores does not completely annihilate the racing condition. The blocking nature of **FIFO pipes** (which requires writing on the other end of the pipe to be opened for reading). As a result, the execution of that thread stalls.
- Two things to be noted, there is high chances of deadlock in the receiver controller if operated on a single pipe, if the read end(for ack frame) is acquired first the fifo hangs the execution until a write end opens but eth frame need to be written on the same pipe to receiver node for ACK frame to be received. And another is if not careful, the same process may write and read to the same pipe thinking it is communicating with another process.
- GO BACK N is comparatively faster in a little noisy channel too, but the whole goodness is swept away by the retransmission of the full window if the ACK frame of the current Sf is not received.
- Selective repeat is very efficient in a noisy environment, instead of sending the whole window it selectively sends the frame on which a NAK is received or for which timeout has occurred.

Round Trip Time:

Propagation Delay: 1 Sec per frame.

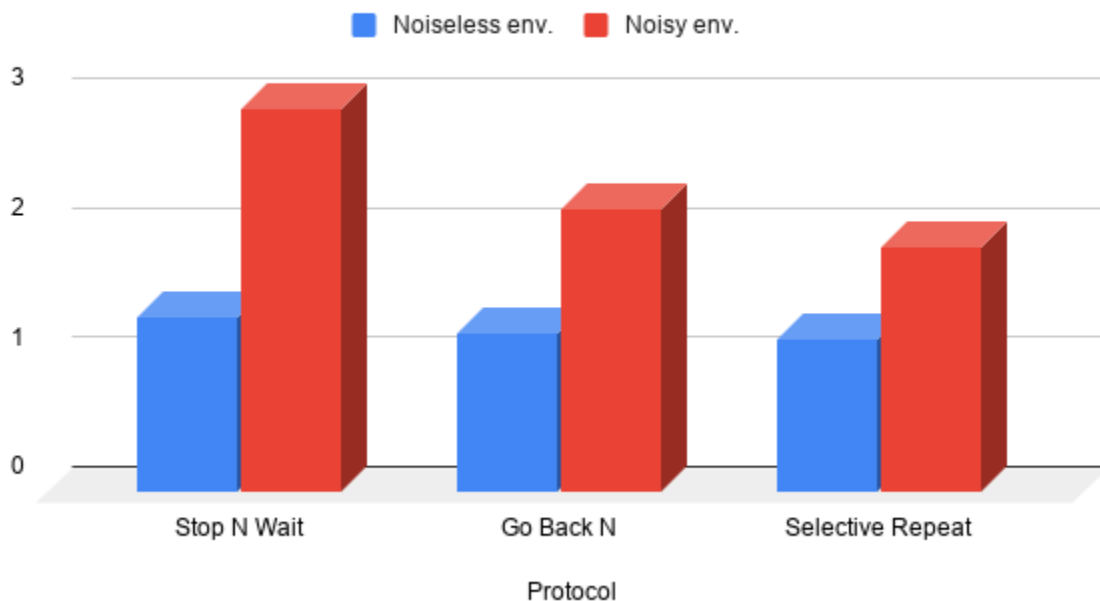
- **StopNwait**

- In Noiseless channel : 1.35 sec
- In Noisy channel: 2.96 Sec (10 % transmitted frames are noisy)

- **GO BACK N**
 - In Noiseless channel : 1.22 sec
 - In Noisy channel: 2.19 Sec (10 % transmitted frames are noisy)

- **Selective Repeat**
 - In Noiseless channel : 1.18 sec
 - In Noisy channel : 1.89 (10 % transmitted frames are noisy)

Round trip time per frame.



COMMENTS:

It was indeed a nice assignment to use the theoretical knowledge as well as the existential coding knowledge to test it and of course, it helps me in clearing the understanding of those Flow Control methods. Overall it was a great learning experience, especially working with IPC using barebone system calls is quite interesting.

For the difficulty, I'd say implementation of those control mechanism is easy but developing the idea to have a communication, managing synchronization, the controller programs, handling a lot of threads, processes, using IPC and the low-level APIs of C and C++ was indeed very *time consuming, tedious and not to mention debugging was on another level.*

Thank you.