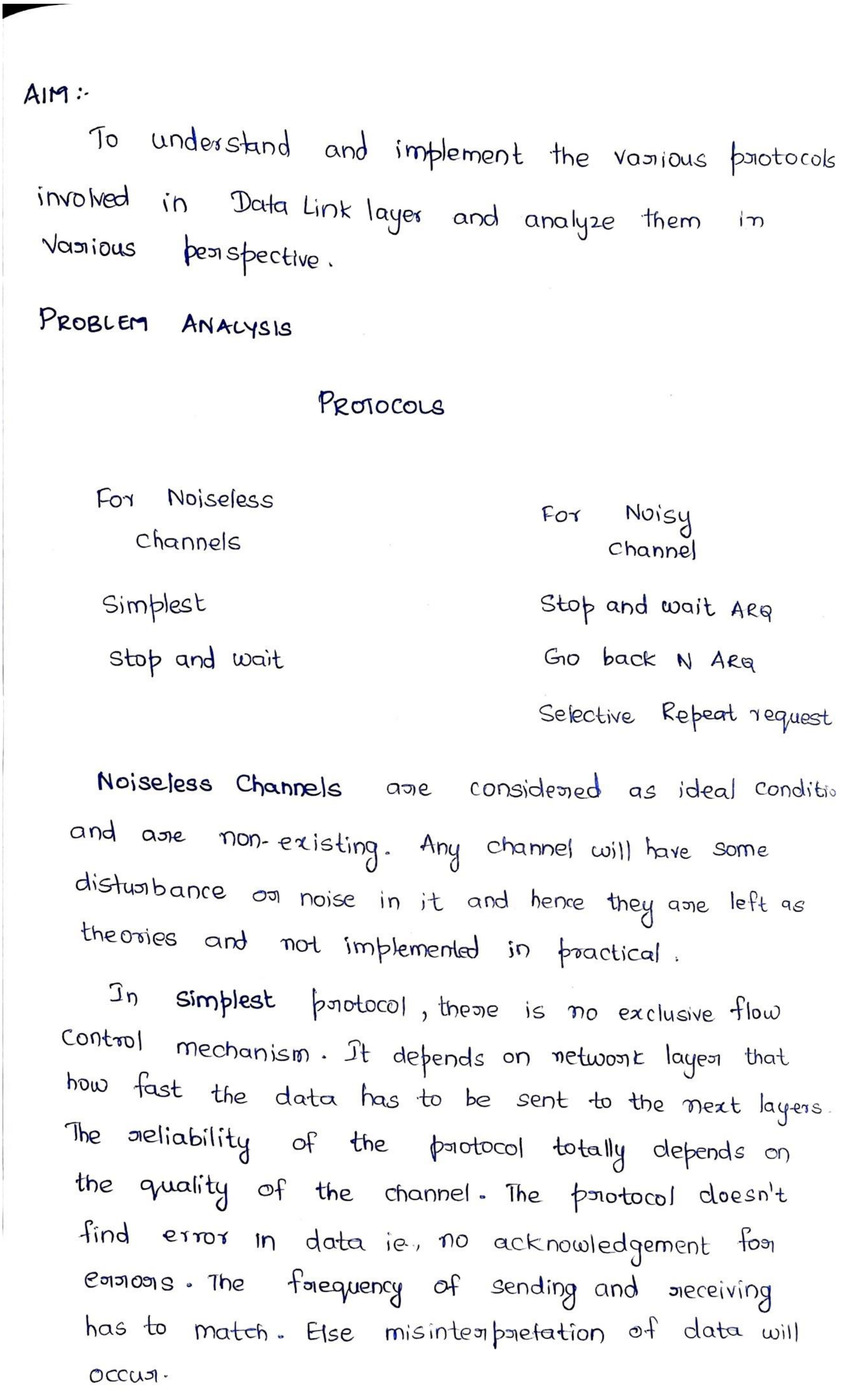
Data Communication and Network

Prashanth.S (19MID0020)



AIM :-

To understand and implement the various protocols involved in Data Link layer and analyze them in various perspective.

PROBLEM ANALYSIS

PROTOCOLS

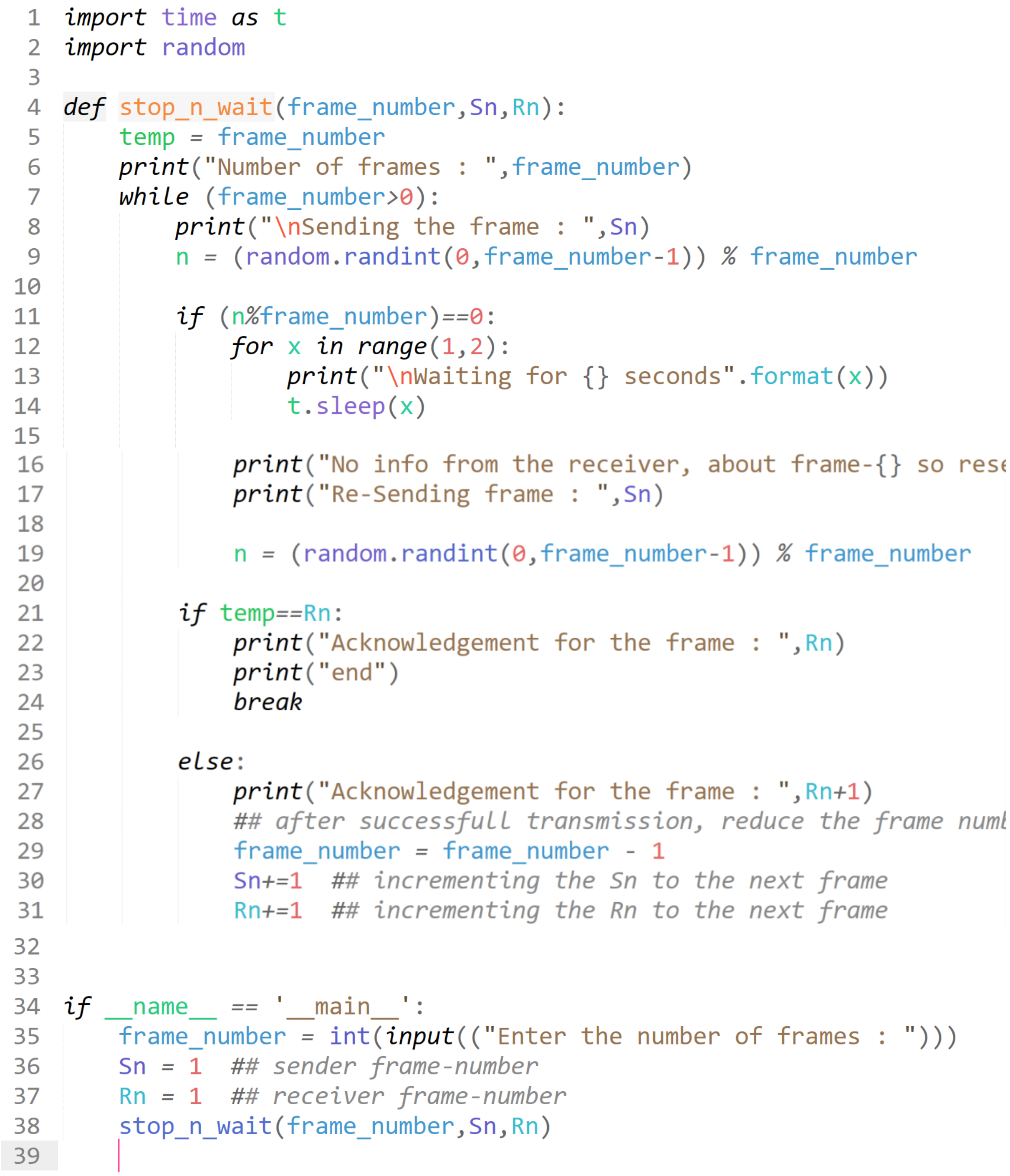
For Noiseless Channels	For Noisy Channel
Simplest	Stop and wait ARQ
stop and wait	Go back N ARQ
	Selective Repeat request

Noiseless Channels are considered as ideal conditio and are non-existing. Any channel will have some disturbance or noise in it and hence they are left as theories and not implemented in practical.

In **Simplest** protocol, there is no exclusive flow control mechanism. It depends on network layer that how fast the data has to be sent to the next layers. The reliability of the protocol totally depends on the quality of the channel. The protocol doesn't find error in data ie., no acknowledgement for errors. The frequency of sending and receiving has to match. Else misinterpretation of data will occur.

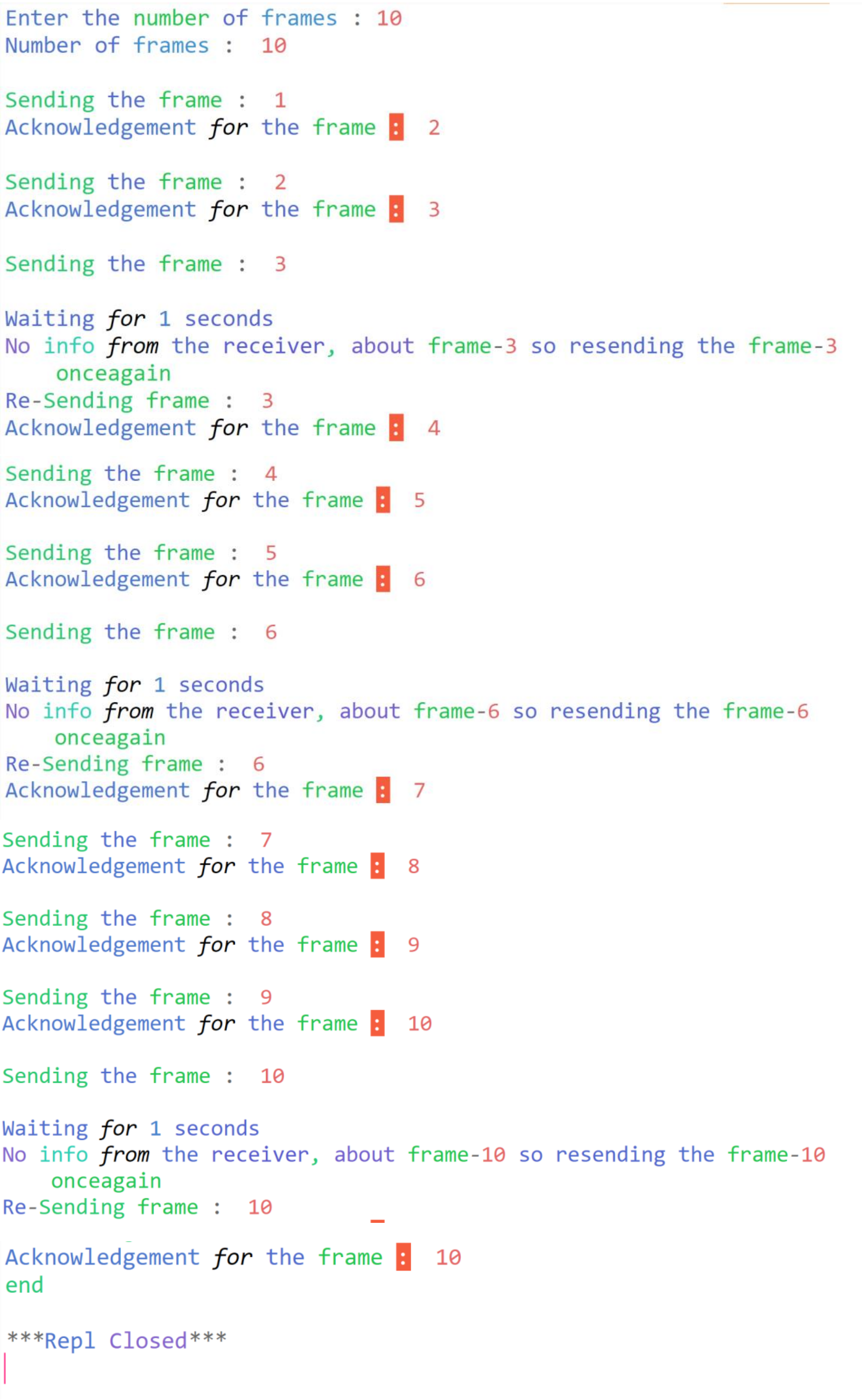
Stop and Wait ARQ

Code



```python
import time as t
import random

def stop_n_wait(frame_number,Sn,Rn):
    temp = frame_number
    print("Number of frames : ",frame_number)
    while (frame_number>0):
        print("\nSending the frame : ",Sn)
        n = (random.randint(0,frame_number-1)) % frame_number

        if (n%frame_number)==0:
            for x in range(1,2):
                print("\nWaiting for {} seconds".format(x))
                t.sleep(x)

            print("No info from the receiver, about frame-{} so rese
            print("Re-Sending frame : ",Sn)

            n = (random.randint(0,frame_number-1)) % frame_number

        if temp==Rn:
            print("Acknowledgement for the frame : ",Rn)
            print("end")
            break

        else:
            print("Acknowledgement for the frame : ",Rn+1)
            ## after successfull transmission, reduce the frame numb
            frame_number = frame_number - 1
            Sn+=1   ## incrementing the Sn to the next frame
            Rn+=1   ## incrementing the Rn to the next frame


if __name__ == '__main__':
    frame_number = int(input(("Enter the number of frames : ")))
    Sn = 1  ## sender frame-number
    Rn = 1  ## receiver frame-number
    stop_n_wait(frame_number,Sn,Rn)

```

Output



```
Enter the number of frames : 10
Number of frames :  10

Sending the frame :  1
Acknowledgement for the frame :  2

Sending the frame :  2
Acknowledgement for the frame :  3

Sending the frame :  3

Waiting for 1 seconds
No info from the receiver, about frame-3 so resending the frame-3
    onceagain
Re-Sending frame :  3
Acknowledgement for the frame :  4

Sending the frame :  4
Acknowledgement for the frame :  5

Sending the frame :  5
Acknowledgement for the frame :  6

Sending the frame :  6

Waiting for 1 seconds
No info from the receiver, about frame-6 so resending the frame-6
    onceagain
Re-Sending frame :  6
Acknowledgement for the frame :  7

Sending the frame :  7
Acknowledgement for the frame :  8

Sending the frame :  8
Acknowledgement for the frame :  9

Sending the frame :  9
Acknowledgement for the frame :  10

Sending the frame :  10

Waiting for 1 seconds
No info from the receiver, about frame-10 so resending the frame-10
    onceagain
Re-Sending frame :  10

Acknowledgement for the frame :  10
end

***Repl Closed***
```

Stop and Wait ARQ

- * stop and wait Automatic Repeat Request.
- * It is almost same to the stop and wait protocol of Noiseless channel except that it has an additional equipment clock.
- * Sender splits the data into frames and the frames are sent one by one, one at a time.
- * sender waits until acknowledgement pack is received from receiver side as previously discussed.
- * The disadvantage of stop and wait protocol is Solved here.
- * The sender resends the frame if the acknowledgement is not received from receiver side within a particular period of time monitored by the clock.
- * For example, if the threshold time period is 2ms, the sender doesn't wait more than 2ms to receive acknowledgement and retransmits the frame assuming that the frame hasn't reached receiver.
- * Even in the case, if acknowledgement pack is lost, if the timer timeouts the threshold time, it retransmits the frame.
- * Duplication is possible as same frame is resent again and again.
- * This will consume more time in sending the entire data frame by frame

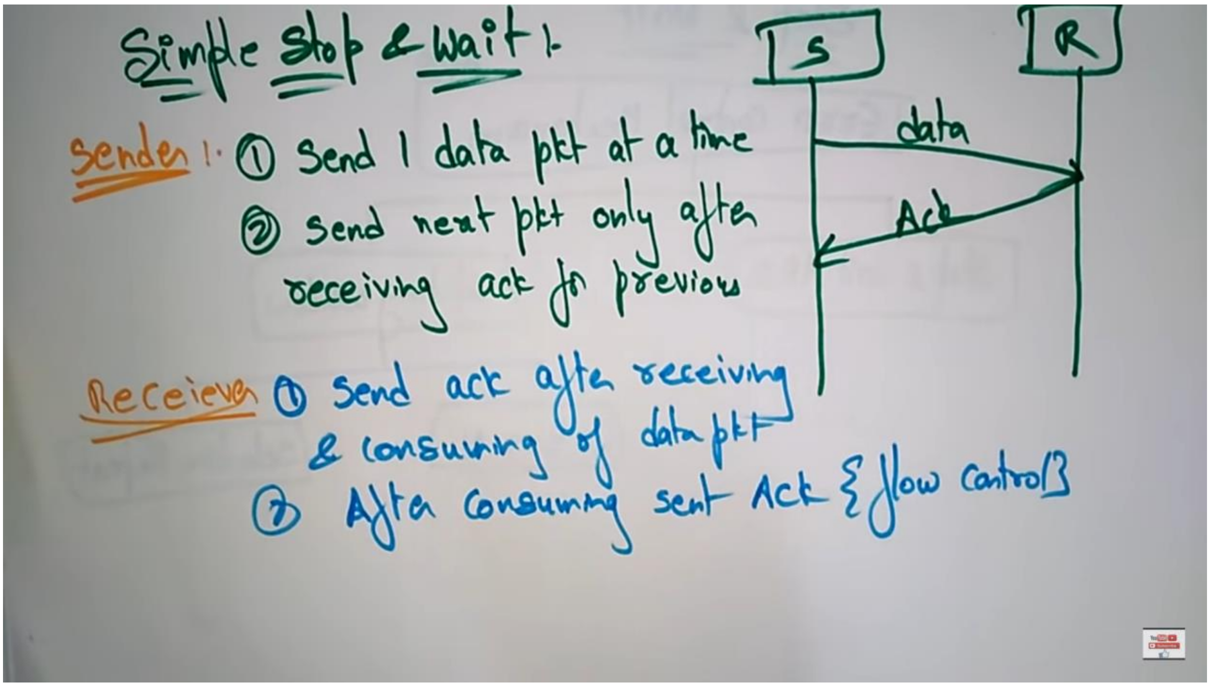
In **Stop and Wait** protocol, once the data is sent from the sender, the sender remains idle until a confirmation is received from receiver. From the receiver end, if the data is received properly without error, it sends an acknowledgement packet to the sender that it has received the data and proceed sending the further data.

It has a disadvantage. If the frame is lost in the channel, then the receiver won't get the data and it won't send any acknowledgement. In this case the sender and receiver will wait indefinitely.

In the other case if the acknowledgement pack is lost, the sender will keep on waiting for acknowledgement pack and again and both sender and receiver waits.

Since noiseless channels do not exist, this assignment has implementation of Noisy protocols alone. The working, Flowchart, implementation with examples are discussed clearly.

Sir I am attaching my notes also.



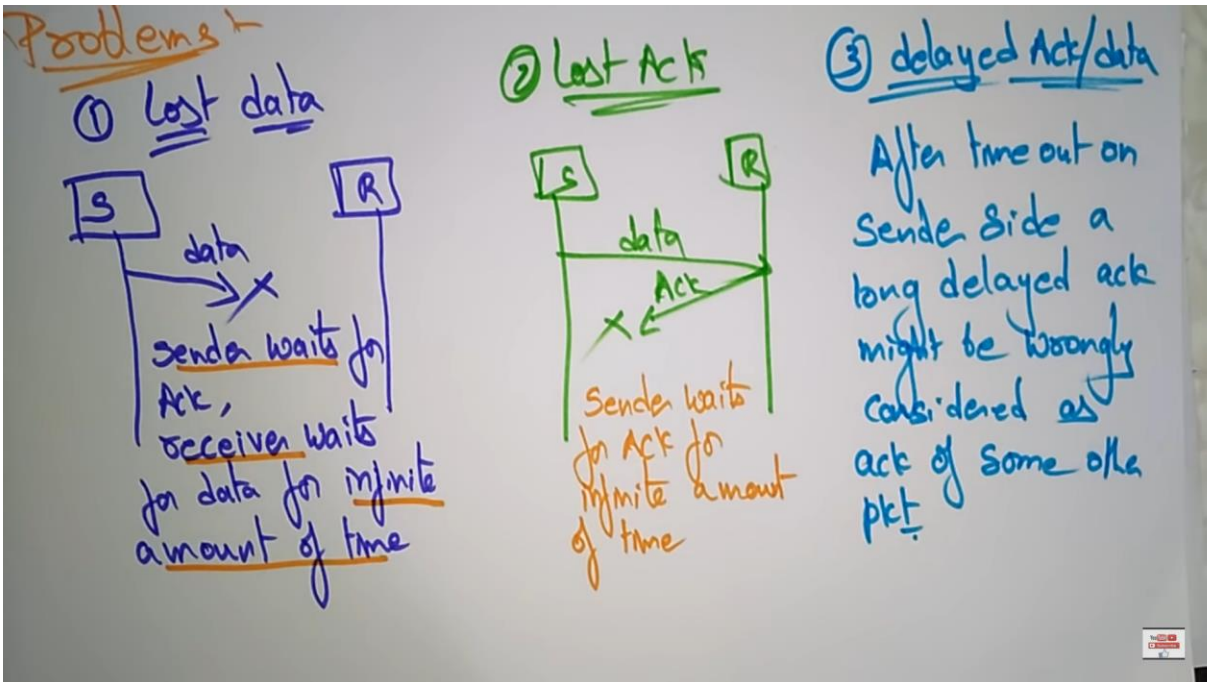
Simple Stop & Wait :-

Sender :- ① Send 1 data pkt at a time

② Send next pkt only after receiving ack for previous

Receiever ① Send ack after receiving & consuming of data pkt

② After consuming sent Ack {flow control}



Problems :-

① Lost data

sender waits for Ack, receiver waits for data for infinite amount of time

② Lost Acks

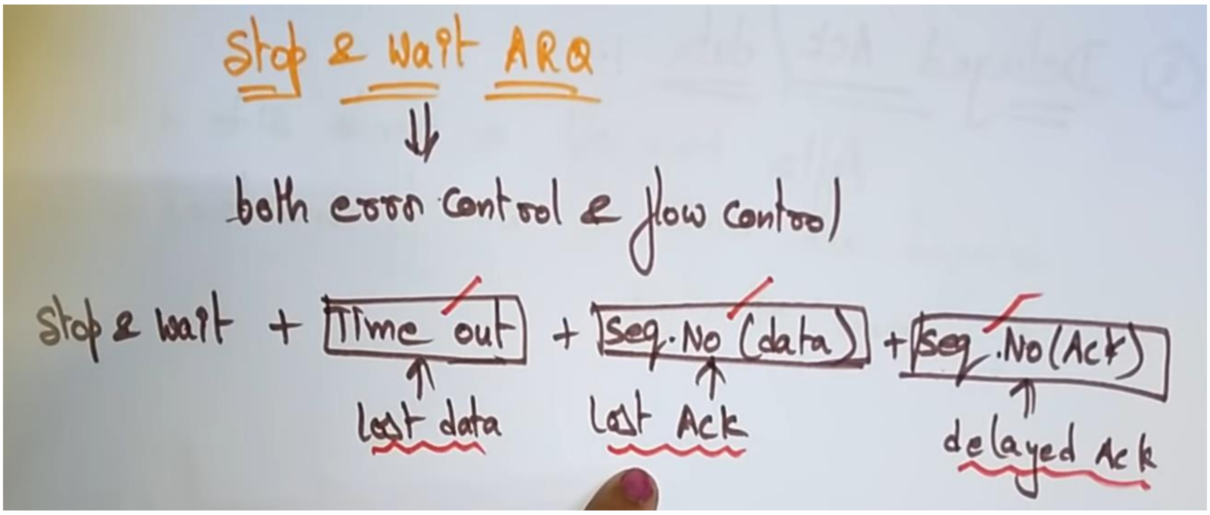
Sender waits for Ack for infinite amount of time

③ delayed Ack/data

After time out on Sender side a long delayed ack might be wrongly considered as ack of some other pkt.

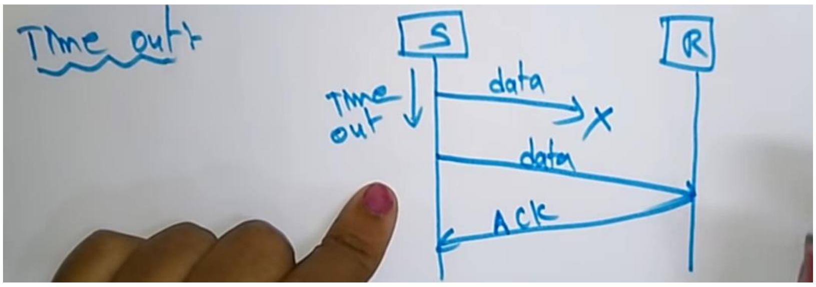
S
R
data
S
R
data
Ack

Since there is no buffer on both the sender and receiver side due to delay the sender doesn’t know / doesn’t keep track of sending the data bits.


Stop & Wait ARQ
both error control & flow control
Stop & wait + Time out + Seq.No (data) + Seq.No(Ack)
lost data
Lost Ack
delayed Ack

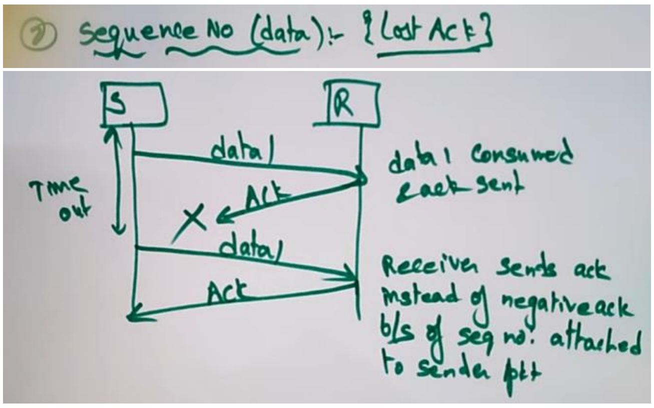
With **Time out** we can overcome the **problem of Lost data**.
With **Sequence No(data)** we can overcome the **Lost acknowledgement.**
With **Sequence No(acknowledgement)** we can overcome the **delayed acknowledgement.**

Time out



The sender will keep a clock(i.e 2s), within that 2s if the acknowledgement is not received by, the sender will once again send the same data once again. With this the problem of losing the data is prevented.

Sequence No(data)

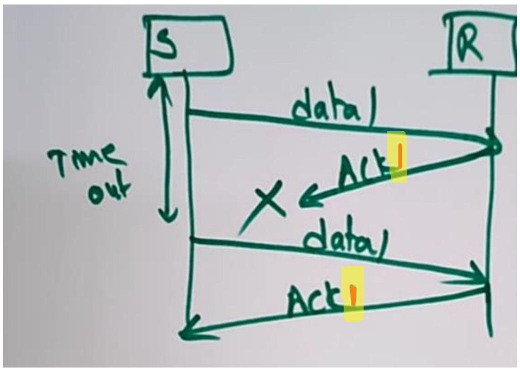

② Sequence No (data):- {Lost Ack}
S
R
data1
Ack
data1
Ack
Time out
data1 consumed & ack sent
Receiver sends ack instead of negative ack b/s of seq no: attached to sender pkt

This is resolved by introducing the sequence number for sender data.
Here the acknowledgement of data1 is not received by the sender. So data 1 is

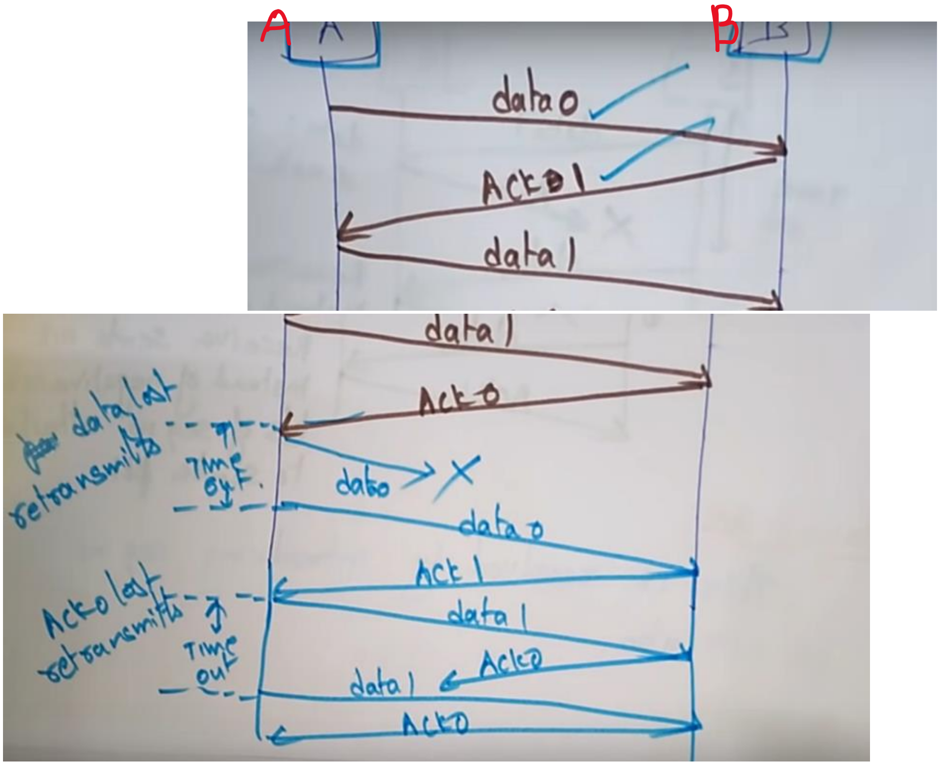
sent once again, on receiving the acknowledgement, the sender sends data2. With this the problem of lost acknowledgement is prevented.

Sequence No(acknowledgement)

This is resolved by introducing the sequence number for acknowledgement also.



If the acknowledgement is lost or data is lost, the sender will resend the data once-again.



Advantages:

* Noisy channel.

* Flow and error control mechanism.

* Timer implementation.

Disadvantages:

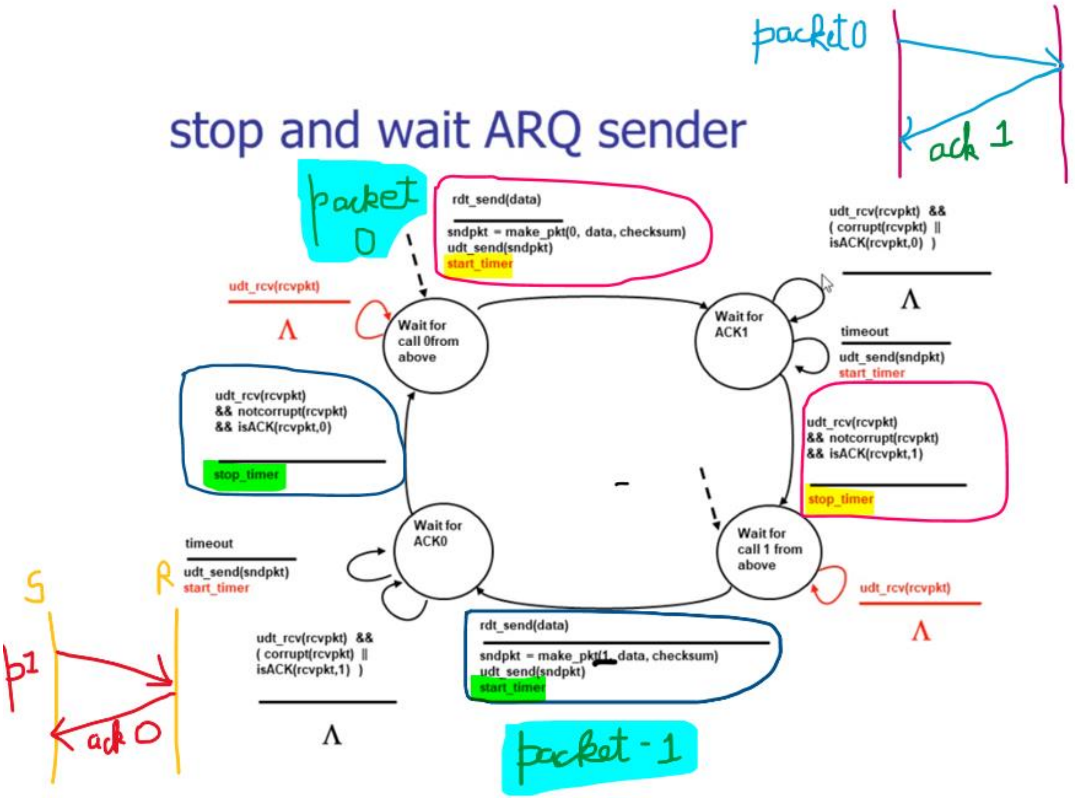
* Efficiency is very less.

* Only 1 frame is sent at a time.

* No process of sending the packets together at a same time.

S R

Finite State Machine (FSM)



packet0
ack 1
stop and wait ARQ sender
packet 0
rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
udt_rcv(rcvpkt) && (corrupt(rcvpkt) \|\| isACK(rcvpkt,0))
Λ
udt_rcv(rcvpkt)
Λ
Wait for call 0from above
Wait for ACK1
timeout
udt_send(sndpkt)
start_timer
udt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
udt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
timeout
udt_send(sndpkt)
start_timer
Wait for ACK0
Wait for call 1 from above
udt_rcv(rcvpkt)
Λ
S
R
p1
ack 0
udt_rcv(rcvpkt) && (corrupt(rcvpkt) \|\| isACK(rcvpkt,1))
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer
packet - 1

Wait for call0 from above:

rdt_send(data)

sndpkt = makepkt(0,data,checksum) → Sending data0

udt_send(sndpkt)

start_timer

Wait for ACK1: (self-loop)

udt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0)) → receiving data0

timeout → If the sender doesn't receive the acknowledgment, it will resend the data and

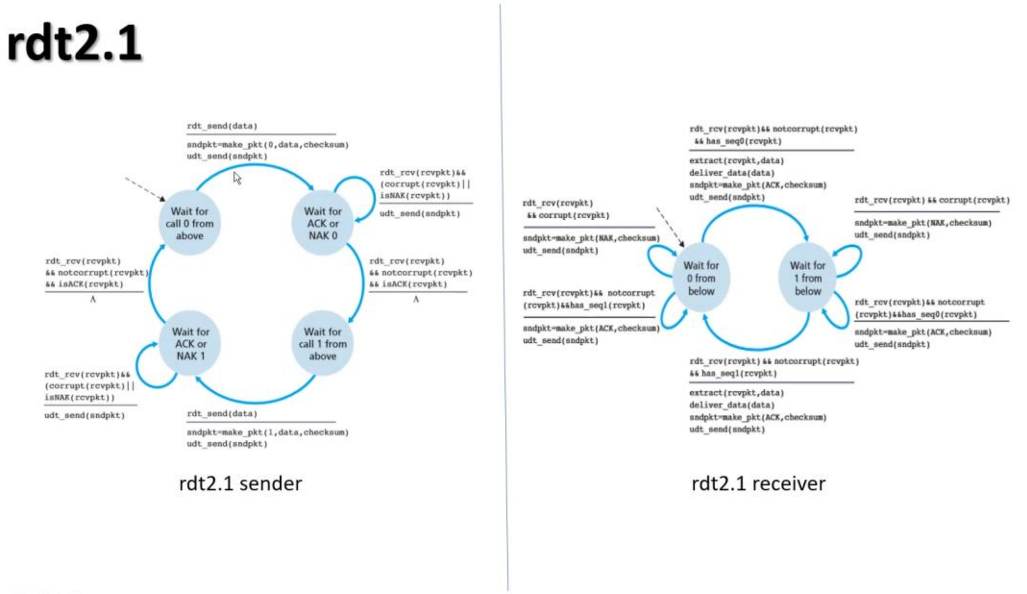
udt_send(sndpkt) start the timer

start_timer

Wait for ACK1: (from 1 state to another)

udt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)

stop_timer

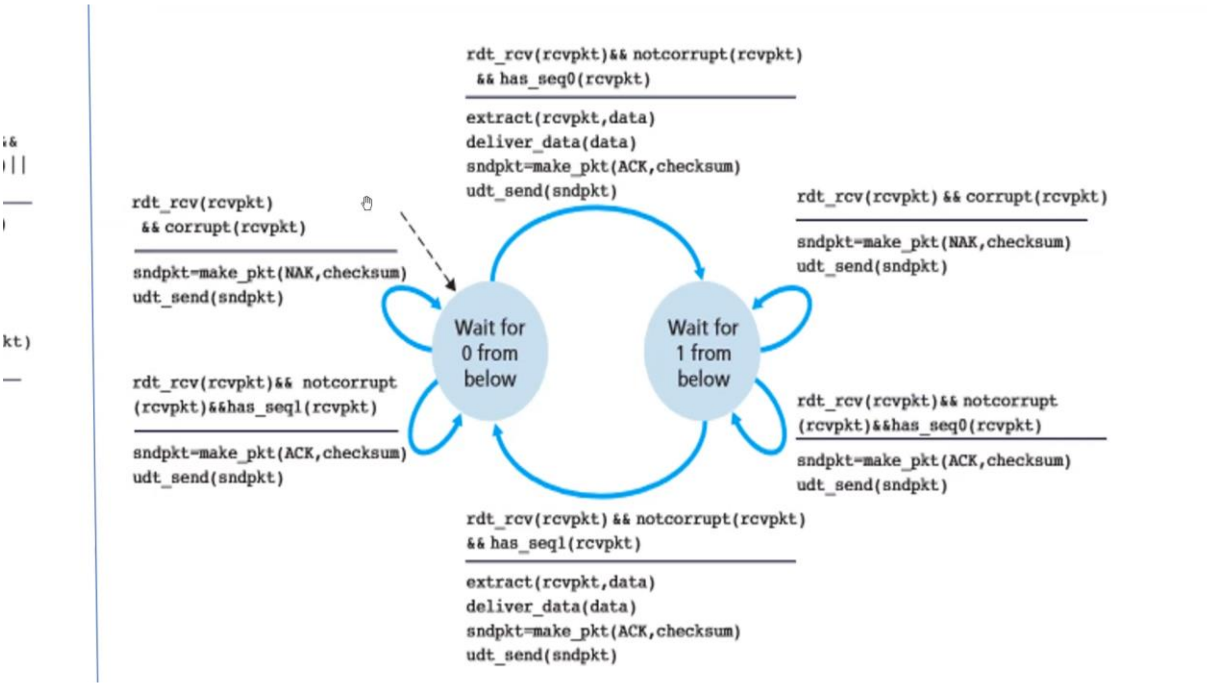


rdt2.1

rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt)&& (corrupt(rcvpkt)|| isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
Wait for ACK or NAK 1
Wait for call 1 from above
rdt_rcv(rcvpkt)&& (corrupt(rcvpkt)|| isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt)&& notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt)&& notcorrupt (rcvpkt)&&has_seq1(rcvpkt)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt)&& notcorrupt (rcvpkt)&&has_seq0(rcvpkt)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)

rdt2.1 sender

rdt2.1 receiver



rdt_rcv(rcvpkt)&& notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt)&& notcorrupt (rcvpkt)&&has_seq1(rcvpkt)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt)&& notcorrupt (rcvpkt)&&has_seq0(rcvpkt)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)

Wait for 0 from below
Will except the data with sequence number 0

Will accept sequence1 also send ACK, but will not extract the data from the packets received.

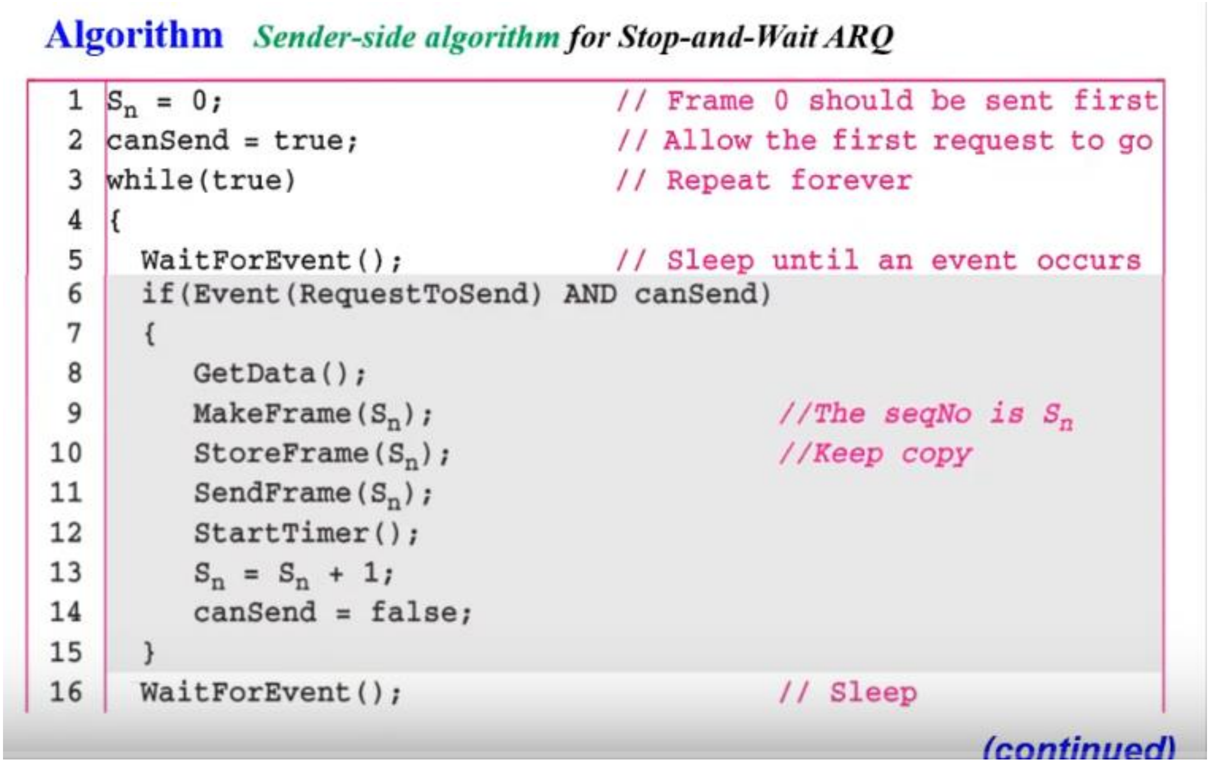
Wait for 1 from below
Will except the data with sequence number 1

Implementation

Sender

1)Data must be sent from the network layer to physical layer

Data will be coming in the form of packets, these packets must be converted into frames and then sent to the physical layer



Algorithm *Sender-side algorithm for Stop-and-Wait ARQ*

```
S_n = 0;                        // Frame 0 should be sent first
canSend = true;                 // Allow the first request to go
while(true)                     // Repeat forever
{
  WaitForEvent();               // Sleep until an event occurs
  if(Event(RequestToSend) AND canSend)
  {
     GetData();
     MakeFrame(S_n);                    //The seqNo is S_n
     StoreFrame(S_n);                   //Keep copy
     SendFrame(S_n);
     StartTimer();
     S_n = S_n + 1;
     canSend = false;
  }
  WaitForEvent();                       // Sleep
```

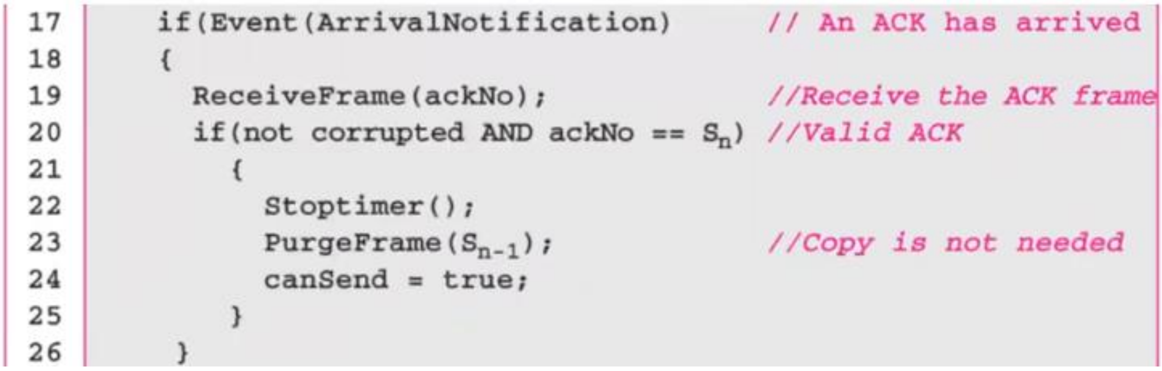
(continued)

After getting the data from network layer, we will create a frame with sequence number and store the frame, in-case if the frame is lost we have to resend the frame once again.

Send the frame and the start the timer, we will increase the sequence number.

flag canSend=False (until we receive the acknowledgment next frame can't be sent)

2) When the acknowledgement have arrived

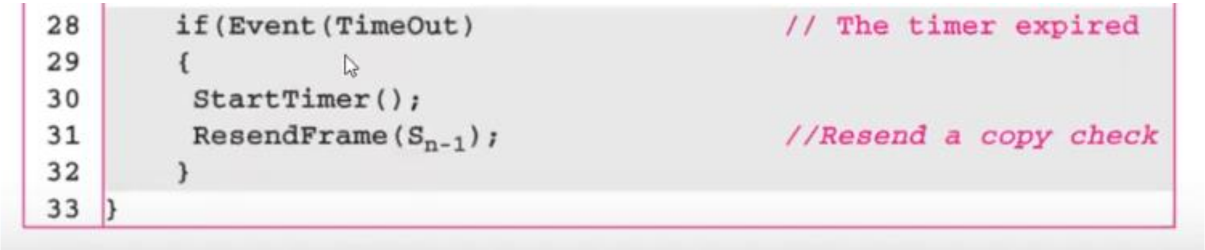


```
    if(Event(ArrivalNotification)      // An ACK has arrived
    {
      ReceiveFrame(ackNo);             //Receive the ACK frame
      if(not corrupted AND ackNo == S_n) //Valid ACK
        {
          Stoptimer();
          PurgeFrame(S_n-1);           //Copy is not needed
          canSend = true;
        }
    }
```

First we Should check if the frame is not corrupted and the acknowledgment number is equal to the next frame which we want to send.

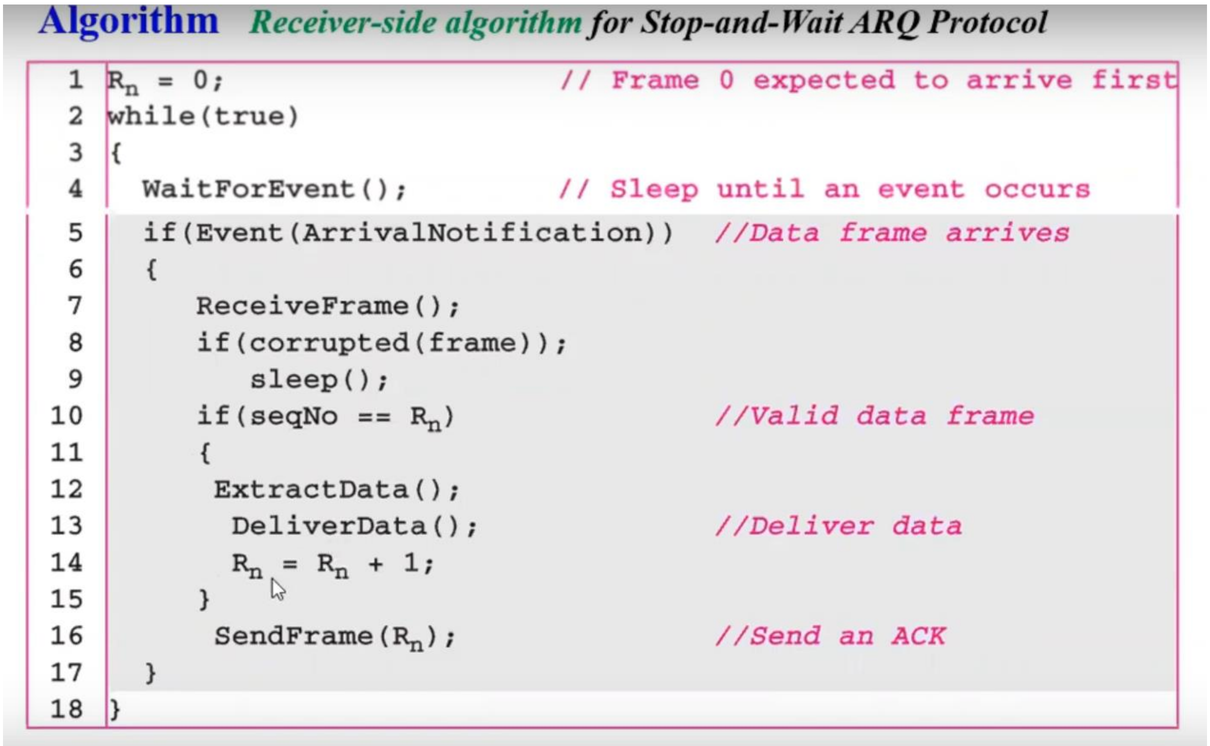
We will stop the timer, purge the frame which was sent and make the flag true, indicating the completion of the event.

3) When the timer was expired



```
    if(Event(TimeOut)                          // The timer expired
    {
     StartTimer();
     ResendFrame($S_{n-1}$);                   //Resend a copy check
    }
}
```

Receiver

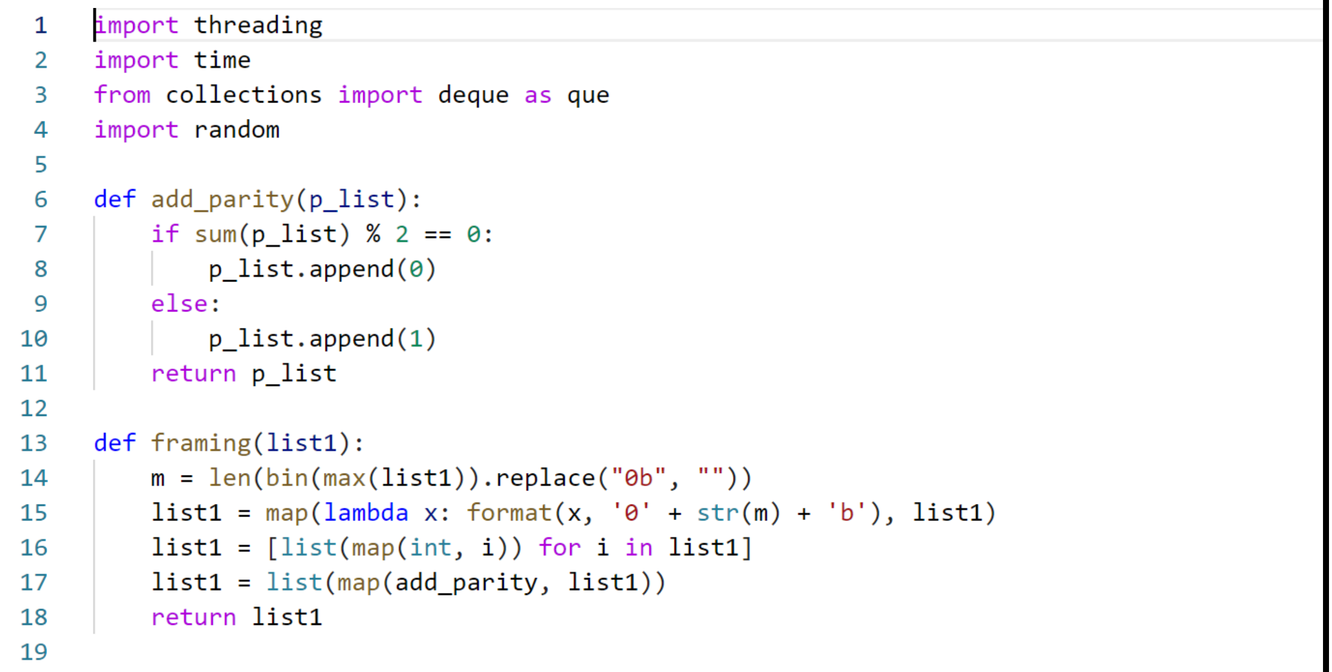


Algorithm *Receiver-side algorithm for Stop-and-Wait ARQ Protocol*

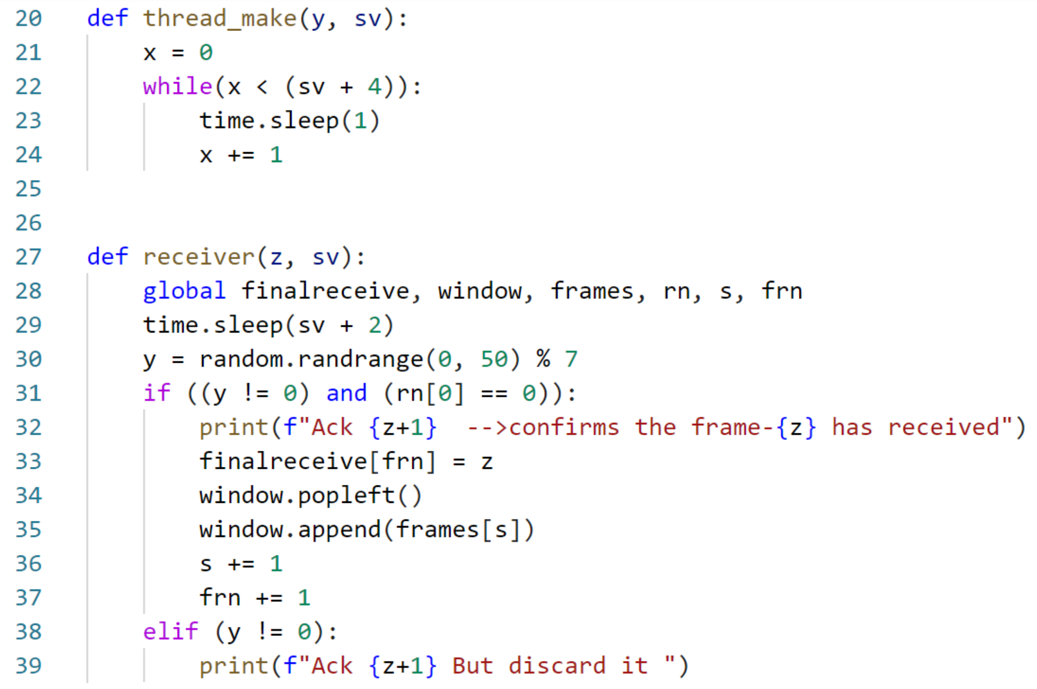
```
$R_n$ = 0;                        // Frame 0 expected to arrive first
while(true)
{
  WaitForEvent();                 // Sleep until an event occurs
  if(Event(ArrivalNotification))  //Data frame arrives
  {
     ReceiveFrame();
     if(corrupted(frame));
        sleep();
     if(seqNo == $R_n$)           //Valid data frame
     {
      ExtractData();
       DeliverData();             //Deliver data
       $R_n$ = $R_n$ + 1;
     }
      SendFrame($R_n$);           //Send an ACK
  }
}
```

Go Back and ARQ

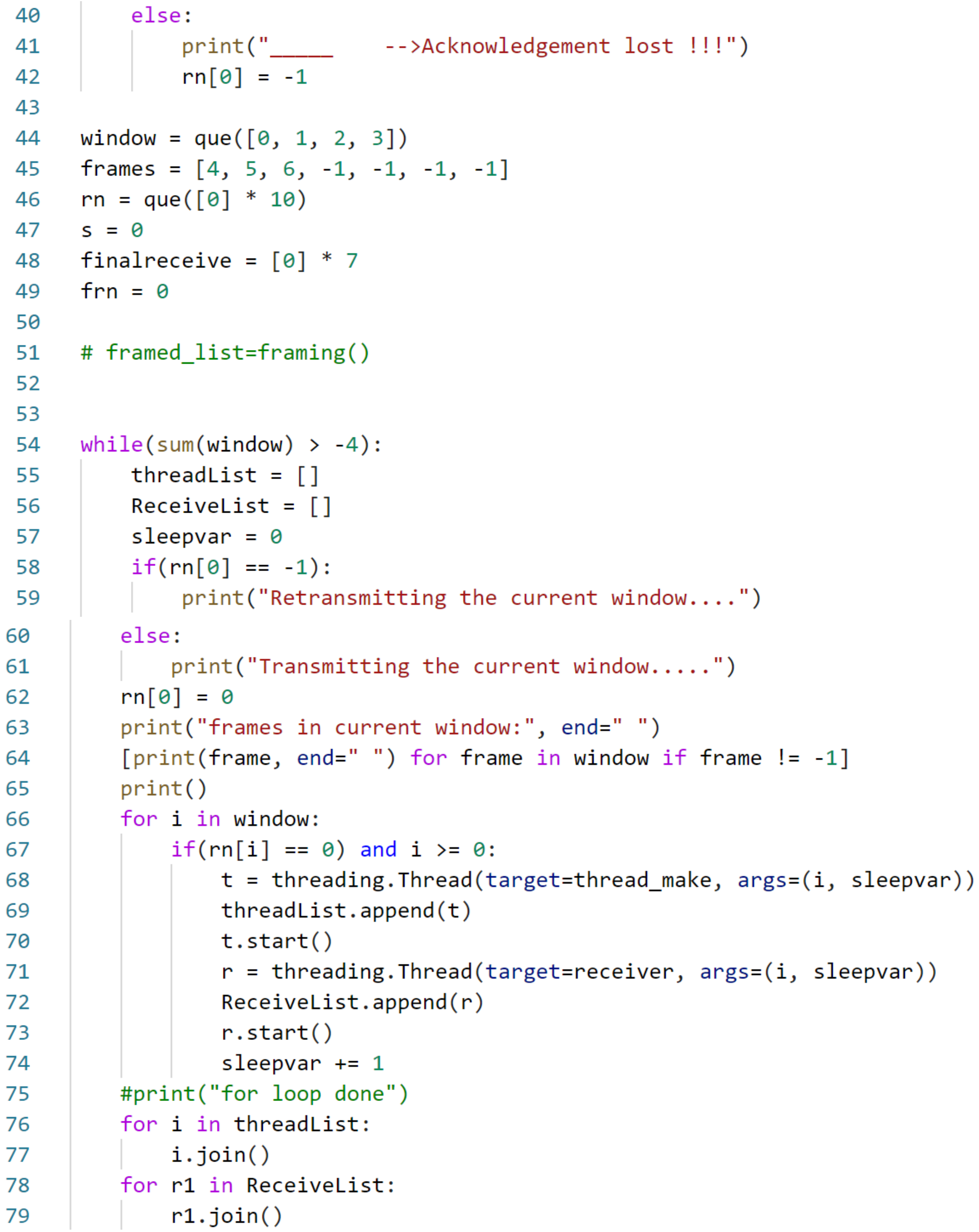
Code



```python
import threading
import time
from collections import deque as que
import random

def add_parity(p_list):
    if sum(p_list) % 2 == 0:
        p_list.append(0)
    else:
        p_list.append(1)
    return p_list

def framing(list1):
    m = len(bin(max(list1)).replace("0b", ""))
    list1 = map(lambda x: format(x, '0' + str(m) + 'b'), list1)
    list1 = [list(map(int, i)) for i in list1]
    list1 = list(map(add_parity, list1))
    return list1
```

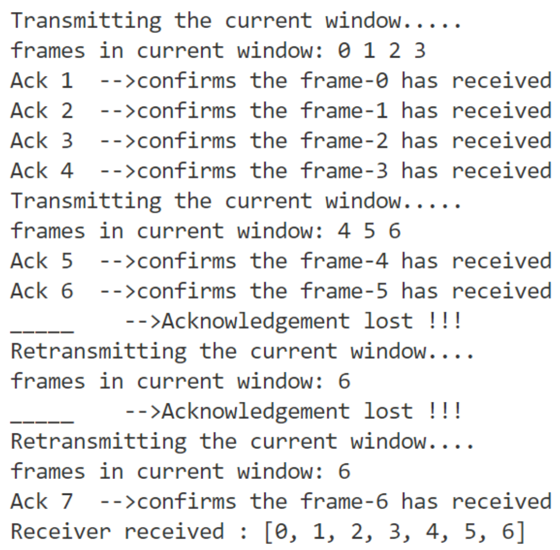


```python
def thread_make(y, sv):
    x = 0
    while(x < (sv + 4)):
        time.sleep(1)
        x += 1


def receiver(z, sv):
    global finalreceive, window, frames, rn, s, frn
    time.sleep(sv + 2)
    y = random.randrange(0, 50) % 7
    if ((y != 0) and (rn[0] == 0)):
        print(f"Ack {z+1}  -->confirms the frame-{z} has received")
        finalreceive[frn] = z
        window.popleft()
        window.append(frames[s])
        s += 1
        frn += 1
    elif (y != 0):
        print(f"Ack {z+1} But discard it ")
```



```python
    else:
        print("_____    -->Acknowledgement lost !!!")
        rn[0] = -1

window = que([0, 1, 2, 3])
frames = [4, 5, 6, -1, -1, -1, -1]
rn = que([0] * 10)
s = 0
finalreceive = [0] * 7
frn = 0

# framed_list=framing()


while(sum(window) > -4):
    threadList = []
    ReceiveList = []
    sleepvar = 0
    if(rn[0] == -1):
        print("Retransmitting the current window....")
    else:
        print("Transmitting the current window.....")
    rn[0] = 0
    print("frames in current window:", end=" ")
    [print(frame, end=" ") for frame in window if frame != -1]
    print()
    for i in window:
        if(rn[i] == 0) and i >= 0:
            t = threading.Thread(target=thread_make, args=(i, sleepvar))
            threadList.append(t)
            t.start()
            r = threading.Thread(target=receiver, args=(i, sleepvar))
            ReceiveList.append(r)
            r.start()
            sleepvar += 1
    #print("for loop done")
    for i in threadList:
        i.join()
    for r1 in ReceiveList:
        r1.join()
```

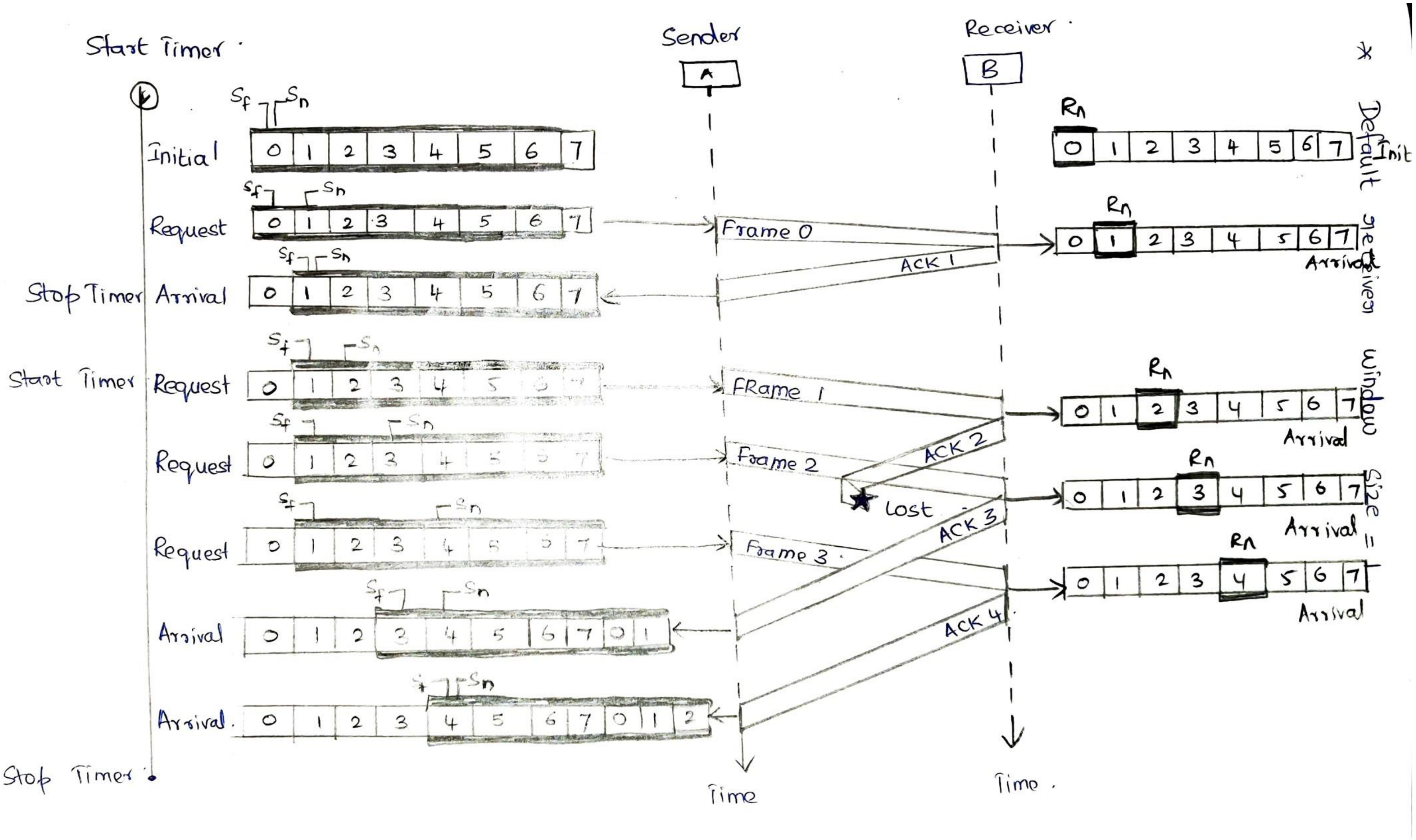
Output



```
Transmitting the current window.....
frames in current window: 0 1 2 3
Ack 1  -->confirms the frame-0 has received
Ack 2  -->confirms the frame-1 has received
Ack 3  -->confirms the frame-2 has received
Ack 4  -->confirms the frame-3 has received
Transmitting the current window.....
frames in current window: 4 5 6
Ack 5  -->confirms the frame-4 has received
Ack 6  -->confirms the frame-5 has received
_____    -->Acknowledgement lost !!!
Retransmitting the current window....
frames in current window: 6
_____    -->Acknowledgement lost !!!
Retransmitting the current window....
frames in current window: 6
Ack 7  -->confirms the frame-6 has received
Receiver received : [0, 1, 2, 3, 4, 5, 6]
```

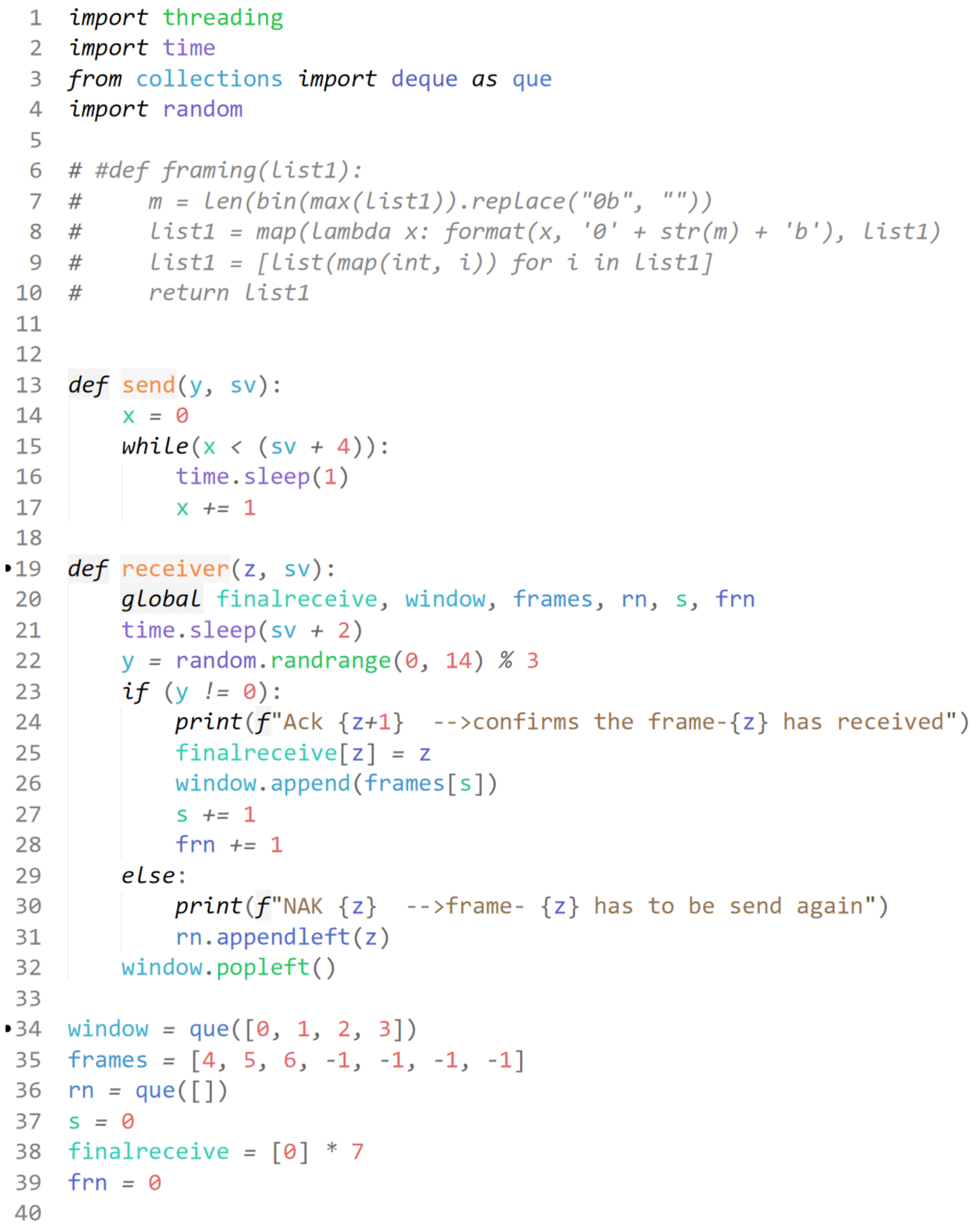
Go Back N ARQ

* This is a sliding window protocol
* The window moves one position as and when it ~~re~~$\overset{ac}{}$quires an acknowledgement.
* The window size will be fixed and it indicates the no of frames to be sent continuously at a stretch.
* Unlike stop and wait ARQ, it doesn't wait for receiver's acknowledgement back for each frame
* This protocol also uses timer to keep track of time taken to receive the Ack pack.
* The protocol starts sending and receiving frames and acknowledgement parallely less than or equal to size of window. Else the sender will keep on sending all the frames and if acknowledgement is not received the frames has to be resent (all frames in this case)
* If the sender has sent all the frames in window, it will wait to receive all the acknowledgements from the receiver's side.
* If the time expires, then the sender will resend frames from current window ie it will send frames based on the recent acknowledgement
* Here the disadvantage is, eventhough the sender has sent a particular frame, it will resend it again from where the acknowledgement packet weren't received.

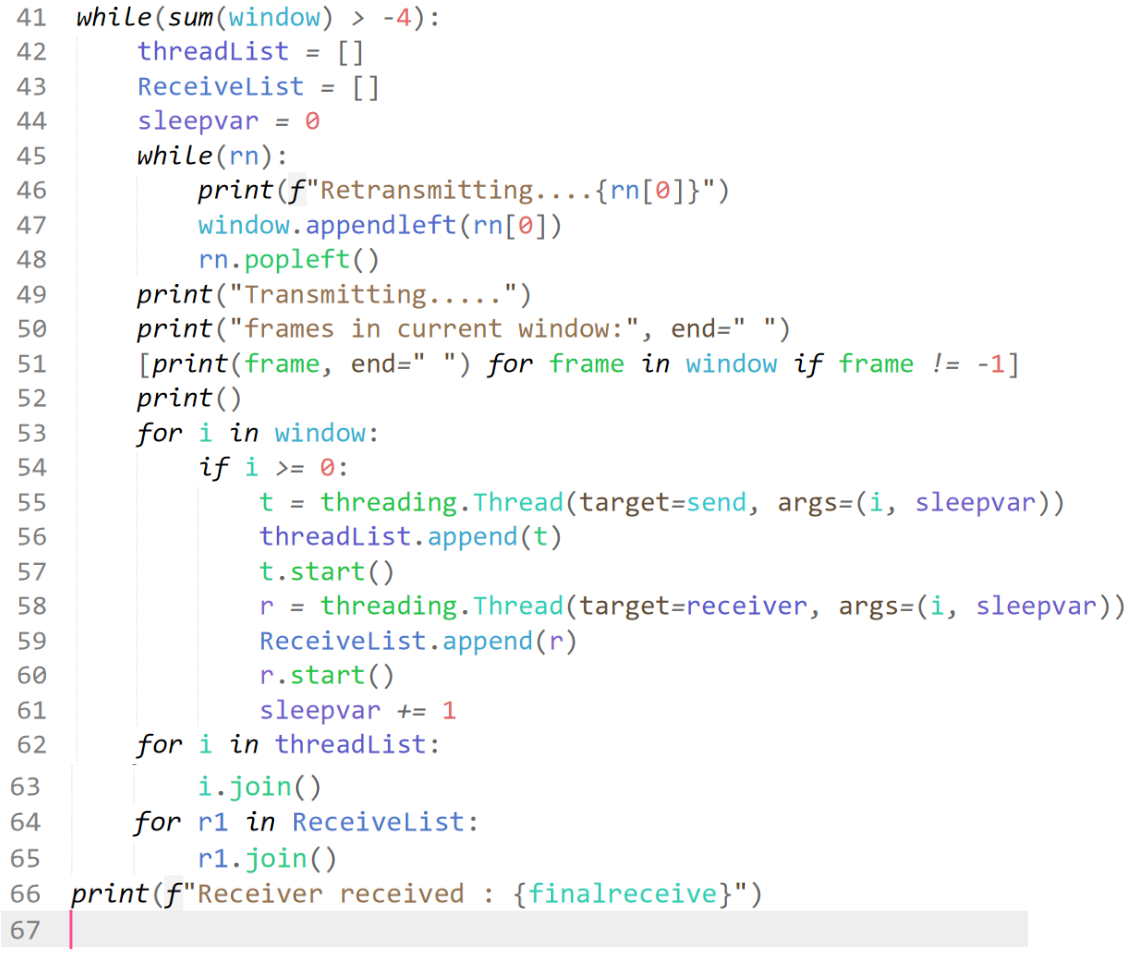

Start Timer
Sender
A
Receiver
B
* Default receiver window size =
Sf
Sn
Initial
0 1 2 3 4 5 6 7
Request
Stop Timer
Arrival
Start Timer
Request
Request
Request
Arrival
0 1 2 3 4 5 6 7 0 1
Arrival
0 1 2 3 4 5 6 7 0 1 2
Stop Timer
Frame 0
ACK 1
FRame 1
ACK 2
Frame 2
Lost
ACK 3
Frame 3
ACK 4
Rn
Init
Arrival
Time
Time

Selective Repeat Request

Code

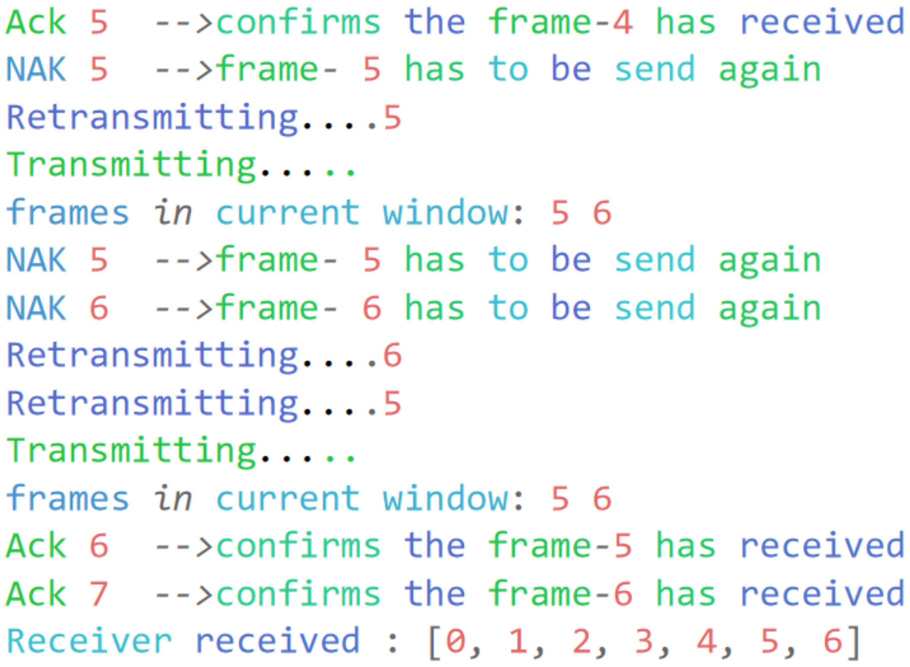


```python
import threading
import time
from collections import deque as que
import random

# #def framing(list1):
#     m = len(bin(max(list1)).replace("0b", ""))
#     list1 = map(lambda x: format(x, '0' + str(m) + 'b'), list1)
#     list1 = [list(map(int, i)) for i in list1]
#     return list1


def send(y, sv):
    x = 0
    while(x < (sv + 4)):
        time.sleep(1)
        x += 1

def receiver(z, sv):
    global finalreceive, window, frames, rn, s, frn
    time.sleep(sv + 2)
    y = random.randrange(0, 14) % 3
    if (y != 0):
        print(f"Ack {z+1}  -->confirms the frame-{z} has received")
        finalreceive[z] = z
        window.append(frames[s])
        s += 1
        frn += 1
    else:
        print(f"NAK {z}  -->frame- {z} has to be send again")
        rn.appendleft(z)
    window.popleft()

window = que([0, 1, 2, 3])
frames = [4, 5, 6, -1, -1, -1, -1]
rn = que([])
s = 0
finalreceive = [0] * 7
frn = 0

```

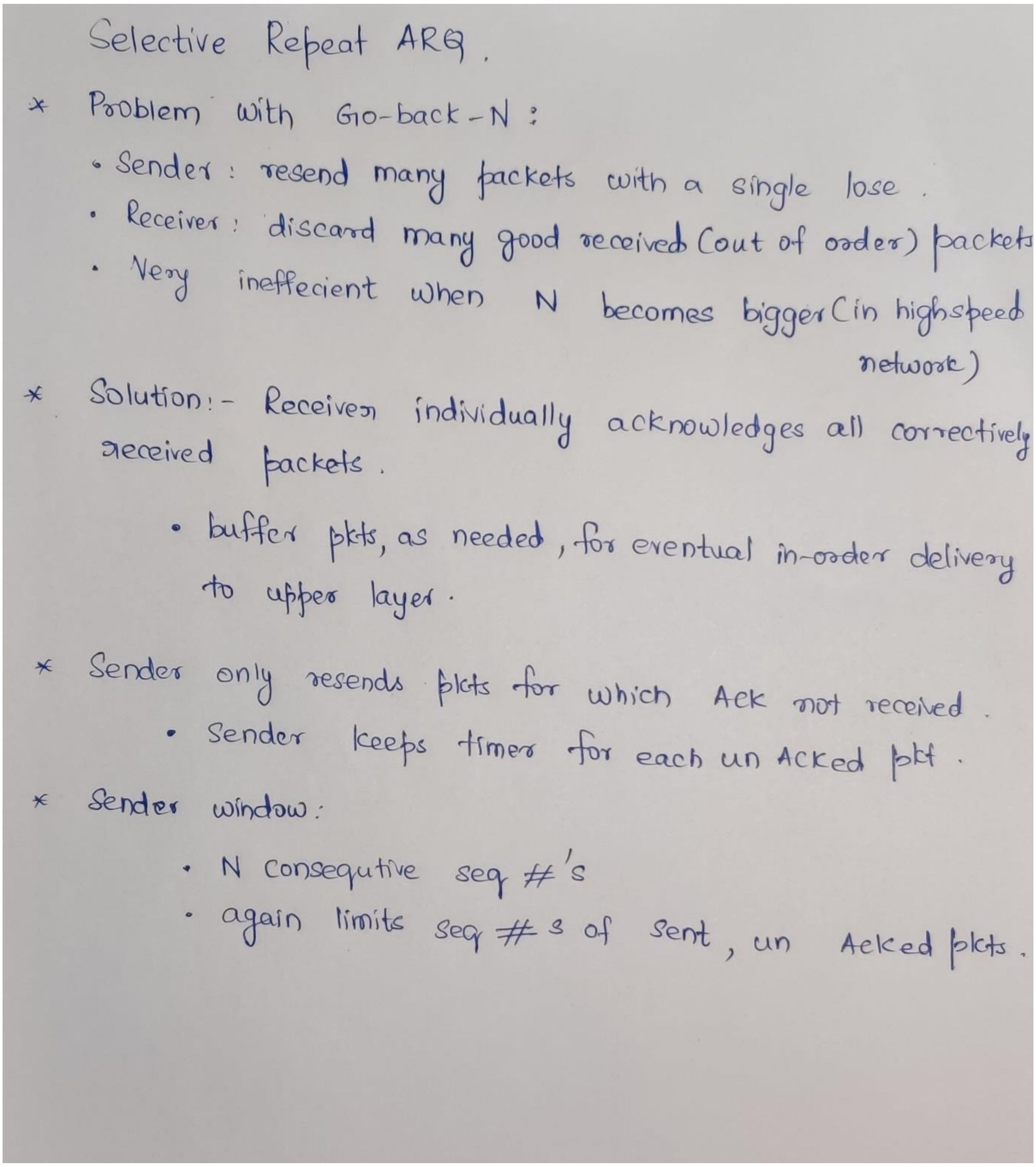


```
while(sum(window) > -4):
    threadList = []
    ReceiveList = []
    sleepvar = 0
    while(rn):
        print(f"Retransmitting....{rn[0]}")
        window.appendleft(rn[0])
        rn.popleft()
    print("Transmitting.....")
    print("frames in current window:", end=" ")
    [print(frame, end=" ") for frame in window if frame != -1]
    print()
    for i in window:
        if i >= 0:
            t = threading.Thread(target=send, args=(i, sleepvar))
            threadList.append(t)
            t.start()
            r = threading.Thread(target=receiver, args=(i, sleepvar))
            ReceiveList.append(r)
            r.start()
            sleepvar += 1
    for i in threadList:
        i.join()
    for r1 in ReceiveList:
        r1.join()
print(f"Receiver received : {finalreceive}")
```

Output



```
Ack 5  -->confirms the frame-4 has received
NAK 5  -->frame- 5 has to be send again
Retransmitting....5
Transmitting.....
frames in current window: 5 6
NAK 5  -->frame- 5 has to be send again
NAK 6  -->frame- 6 has to be send again
Retransmitting....6
Retransmitting....5
Transmitting.....
frames in current window: 5 6
Ack 6  -->confirms the frame-5 has received
Ack 7  -->confirms the frame-6 has received
Receiver received : [0, 1, 2, 3, 4, 5, 6]
```



Selective Repeat ARQ.

* Problem with Go-back-N :

- Sender : resend many packets with a single lose.
- Receiver : discard many good received (out of order) packets
- Very ineffecient when N becomes bigger (in highspeed network)

* Solution:- Receiver individually acknowledges all correctively received packets.

- buffer pkts, as needed, for eventual in-order delivery to upper layer.

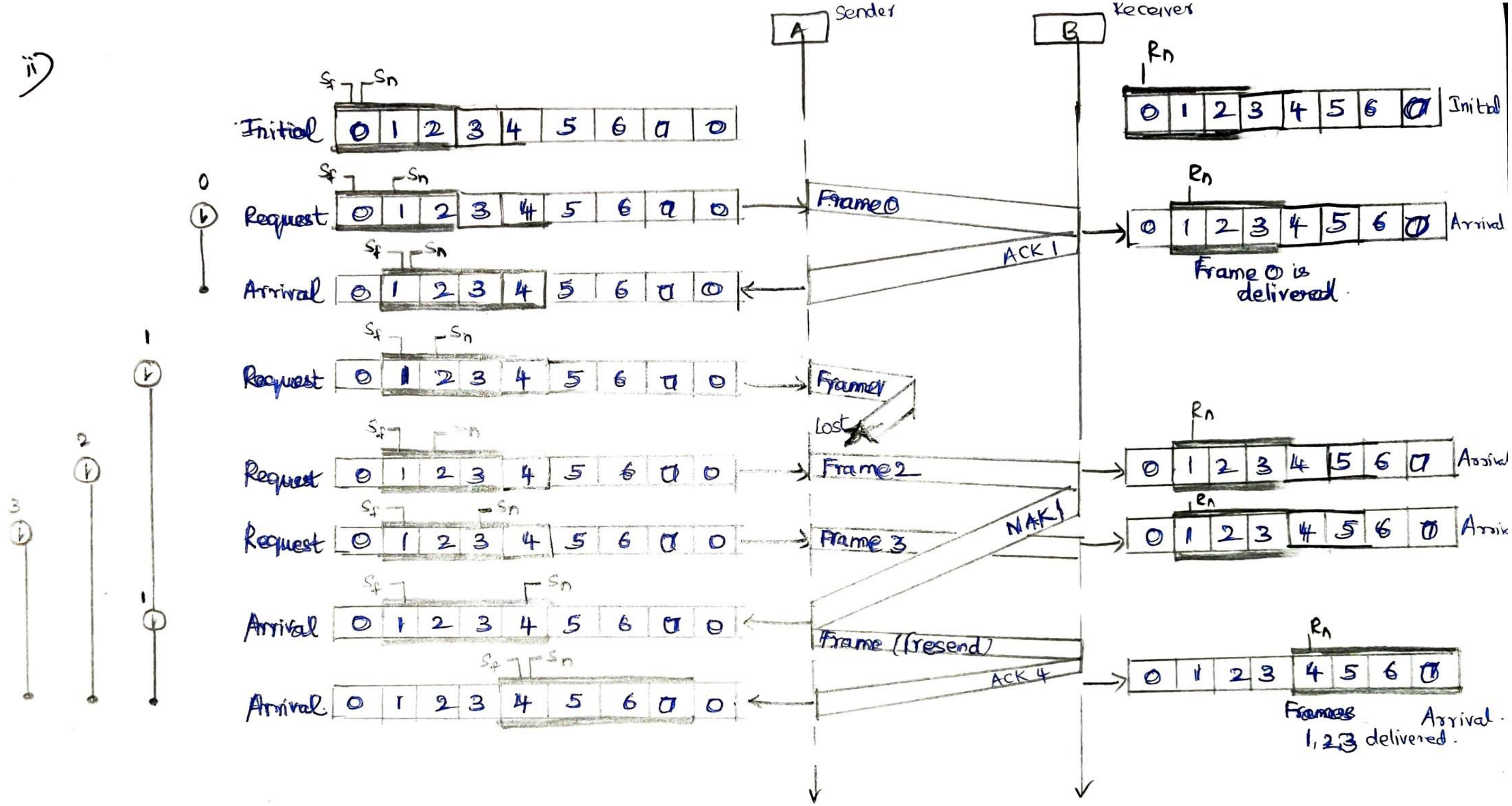
* Sender only resends pkts for which Ack not received.

- Sender keeps timer for each un ACKed pkt.

* Sender window:

- N consequtive seq #'s
- again limits seq #s of sent, un Acked pkts.

ii)


A
Sender
B
Receiver
Initial
Request
Arrival
Request
Request
Request
Arrival
Arrival
Sf
Sn
Rn
0 1 2 3 4 5 6 7 0
0 1 2 3 4 5 6 7
Frame 0
ACK 1
Frame 1
Lost
Frame 2
Frame 3
NAK 1
Frame 1 (resend)
ACK 4
Initial
Arrival
Frame 0 is delivered.
Arrival
Arrival
Frames 1,2,3 delivered.
Arrival.
0
1
2
3
1