Module-2 Transport Layer

Introduction & Transport Layer Services

A transport-layer protocol provides logical communication blw app' processes running on different hosts.

On the sender side, the transport layer receives the msg from the app' layer, split them into chunks & adds header, which is called as segments. These segments are passed to the nlw layer, where nlw layer header is encapsulated and sent to the receiver side.

Then in the receiver side the packet/segment is processed by the nlw & transport layer & reaches the app' layer.

Relationship blw Transport & Nlw Layers

Transport-layer protocol provides logical conn' blw processes running on different hosts, a nlw layer protocol provides logical conn' blw hosts

eg: 2 Home 1. Bangaloxe 2. Mumbai kids in each home are cousins of kids in other home.

Kids in Bangaloxe home will send & receive letters through post to & from kids in Mumbai home. A Home have 200 children.

A kid (Bob) will collect all the letters from the home & give the letters to the post office.

The (post office) postal service will send the letter to the respective home.

Again a child/kid (Alice) will get all the letters & distribute it to all the kids.

App'-Msg → letters, Host- Home Process- Cousins/kids

Transport-layer protocol- Bob & Alice. Nlw-layer protocol- Postal service.

Routers doesn't examine any info' that trans' layer have added to the app' layer msgs.

Overview of the Transport Layer in the Internet

UDP (User Datagram Protocol) - provides unreliable, connectionless service.

TCP (Transmission ctrl Protocol) - provides reliable, connection oriented service to the invoking app'.

TCP packet - segment UDP - Datagram.

Nlw Layer protocol - IP (Internet Protocol) → Model is a best-effort delivery service.

But it doesn't guarantee i) segment delivery, ii) orderly delivery of segments iii) integrity of the data.

So, IP is said to be an unreliable service.

Extending host-to-host delivery to process-to-process delivery is called transport-layer multiplexing & demultiplexing.

UDP & TCP also provide integrity checking by including error detection fields in their segment's headers.

TCP → Reliable using flow ctrl, sequence no's, acks, & timers.

Congestion ctrl. is also provided.

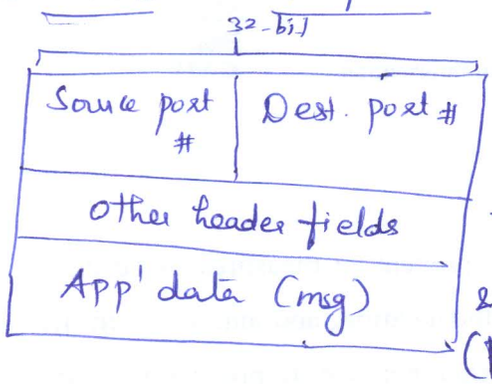
Multiplexing & Demultiplexing

Gathering data chunks from diff' sockets, creating segments by adding header inf' to each chunk & passing the segments to nlw layer is called as multiplexing.

Delivering the segment to the correct socket (eg: multiple sockets will be there one for FTP, one for HTTP, SMTP, ..) is called as Demultiplexing.

Bob performs - Multiplexing Alice performs - Demultiplexing

Source & dest' port-no' fields in a transport-layer segment



Socket id'no - port no' → 16 bit.

Range - 0 to 65535, 0 to 1024 - Reserved

The transport layer examines the segment & directs the segment to the specified (port no) socket.

Connectionless Multiplexing and Demultiplexing

In the UDP Socket, in client side. a Socket is created by giving

```
DatagramSocket ds = new DatagramSocket ();
```

The transport layer will automatically assigns a port no' in the range 1024 to 65535 or can also be given manually.

Whereas in case of server side the port no' should be given.

```
DatagramSocket ds = new DatagramSocket (3000);
```

If a process in Host A, with UDP port 2556, want to send a chunk of app' data to a process in Host B, with UDP port 3556, the transport layer in Host A creates a transport layer segment with port no of source, destination & 2 two other values.

The transport layer then passes the resulting segment to the n/w layer.

Once the segment reaches the transport layer of the receiving side, the transport layer delivers the segment to the socket identified by the port no' (3556).

port no, then the 2 segments will be directed to the same destination process via the same destination socket.

If If server wants to send data to the client then client port no' - is used.

Connection-oriented Multiplexing & Demultiplexing

TCP socket is identified by a 4-tuple: Source IP address, Port no', destination IP add', port no'.

In constrast to UDP, 2 TCP segments with different source IP add' or port no' & same destination IP add' & port no' will be directed to 2 different sockets (except connection establishment req').

Client-side:

```
Socket s = new Socket ("localhost", 6666);
```

Server-side:

```
ServerSocket ss = new ServerSocket (6666);
Socket s = ss.accept(); // accepts the connection req'
```

All segments whose source port, IP add', destination port, IP add' match the specified one will be demultiplexed to this socket.

eg:

Host C initiates 2 HTTP sessions to server B.

Host A " 1 " session "

Host A, B & C have their unique IP add'.

Host C assigns 2 different source port no' (26145 & 7532) to its HTTP connections. Host A may also assign source port no' - 7532.

But B, will be able to correctly demultiplex the 2 connections, since the 2 connections have different source IP add'.

Web Servers & TCP

When client send segments to server (both connection req & message) the port no' will be 80. But, the server will identify the segments by the client port no' & ip add'.

Non-persistent conn' → new socket is created/closed, not suitable for busy server.

Persistent conn' → single server socket is enough.

Connectionless Transport: UDP

UDP - Connectionless, Unreliable.

Used in DNS. The client app' sends query to the DNS server if it doesn't get the response, it will send req' to some other server, or it informs the invoking app' that it can't get a reply.

TCP is reliable, but UDP is still suited for many app' for the following reasons:

i) Finer app'-level ctrl over what data is sent & when:

Real-time app' often require a min' sending rate, do not want to overly delay segment transmission, & can tolerate some packet loss.

ii) No Connection establishment.

UDP doesn't introduce any delay to establish a connection.

iii) No Connection state

UDP doesn't maintain connection state & does not track the parameters like send & receive buffers, congestion-ctrl parameters & ack no' parameters.

iv) Small packet header overhead

TCP segment header overhead - 20 bytes

UDP segment " - 8 bytes

UDP is used in DNS, RIP (Routing Protocol), SNMP, Internet telephony, Multimedia.

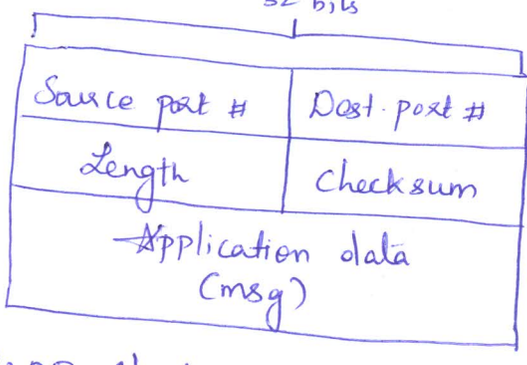
Since RIP updates are sent periodically, lost updates will be replaced by more recent updates, thus making the lost, out-of-date update useless.

UDP is un-reliable & has no congestion ctrl.

Building reliability directly into the app' allows the app' to have "its cake & eat it too".

ie. App' processes can communicate reliably without being subjected to the transmission-rate constraints imposed by TCP's congestion ctrl mechanism.

UDP Segment Structure



Data field - DNS → req (query) | res, Audio - audio samples

The length field specifies - the no' of bytes in the UDP segment (header plus data).

header (8 bytes each)

→ used by the receiving host to check whether errors have been introduced into the segment

UDP Checksum → provides error detection.

The checksum is used to determine whether bits within the UDP segment have been altered as it moved from source to destination.

UDP at the sender side performs the 1s complement of the sum of all the 16-bit words in the segment, with any overflow encountered during the sum being wrapped around. This result is put in the checksum field of the UDP segment.

In the receiver side the checksum is added along with the data & if the result is 111....1 then there is no error. Not else some error is present.

eg: 3 - 16-bit words.

0 1 1 0 0 1 1 0 0 1 1 0 0 0 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 0 0 1 1 1 1 0 0 0 0 1 1 0 0.

Sender

The sum of 1st 2 words is

0 1 1 0 0 1 1 0 0 1 1 0 0 0 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
---
1 0 1 1 1 0 1 1 1 0 1 1 0 1 0 1 → sum.

Adding 3rd word with the sum

1 0 1 1 1 0 1 1 1 0 1 1 0 1 0 1
1 0 0 0 1 1 1 1 0 0 0 0 1 1 0 0
---
1 0 1 0 0 1 0 1 0 1 1 0 0 0 0 0 1
↳ 1 (wrapped around)
---
0 1 0 0 1 0 1 0 1 1 0 0 0 0 1 0
---
1 0 1 1 0 1 0 1 0 0 1 1 1 1 0 1 1st complement (checksum).

Receiver

0 1 1/3 16-bit words + checksum.

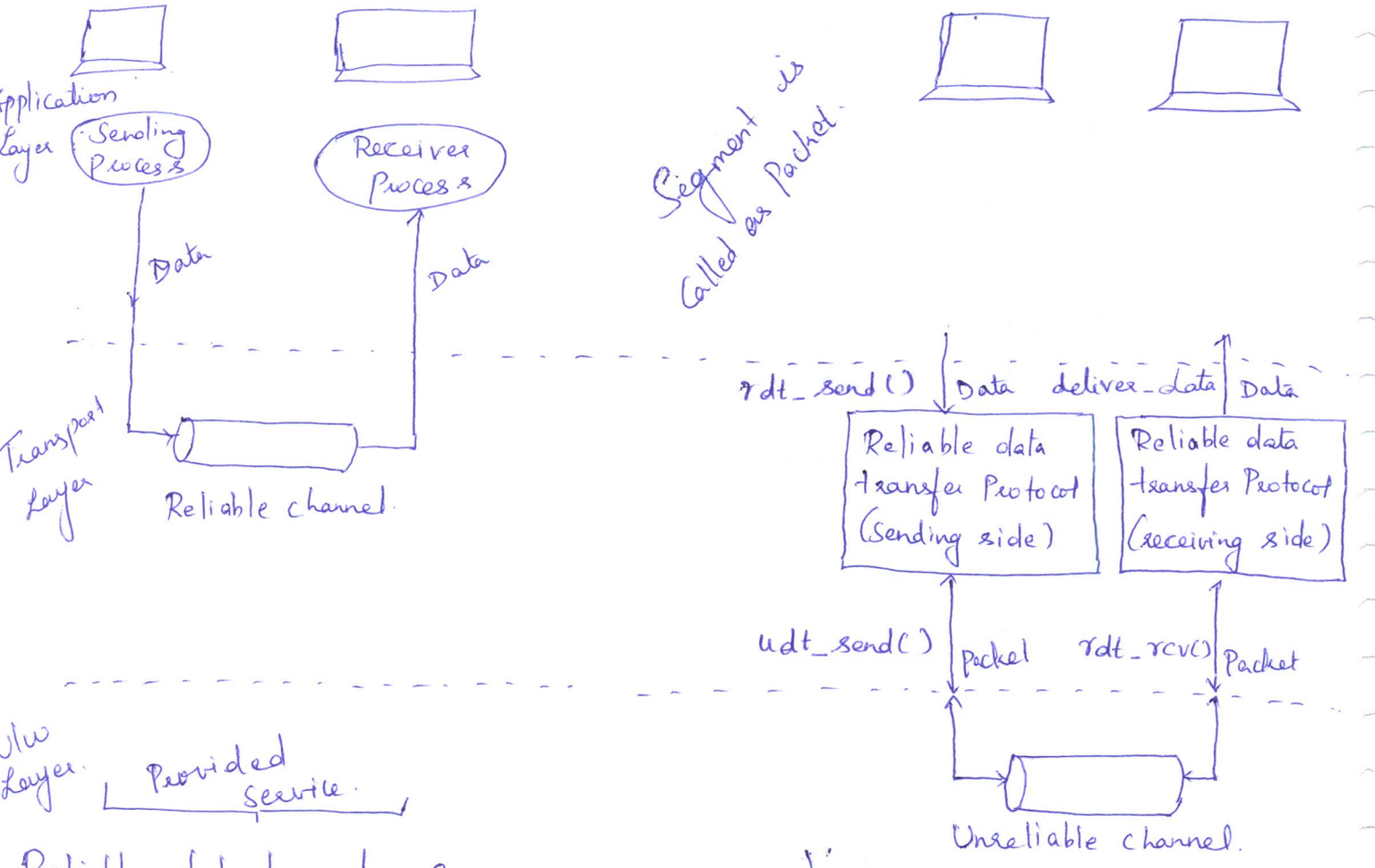
1 0 0 0 1 1 1 1 0 0 0 0 1 1 0 0 → 3rd word
1 0 1 1 0 1 0 1 0 0 1 1 1 1 0 1 → checksum.
---
1 0 1 0 0 0 1 0 0 0 1 0 0 1 0 0 1
↳ 1
---
0 1 0 0 0 1 0 0 0 1 0 0 1 0 1 0 sum 3rd word + checksum
1 0 1 1 1 0 1 1 1 0 1 1 0 1 0 1 sum 1st word + 2nd word
---
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

→ No error.

UDP provides checksum (error checking) in transport layer. As there is no guarantee that all the links b/w source & destination provide error checking. (data-link layer).

End-end-Principle: Functions placed at the lower levels may be redudant or of little value when compared to the cost of providing them at the higher level. Error could also introduced when the segment is stored in router's m/y. UDP doesn't provide error correction. It simply discards the damaged segment / gives warning to the app.

Principles of Reliable Data Transfer



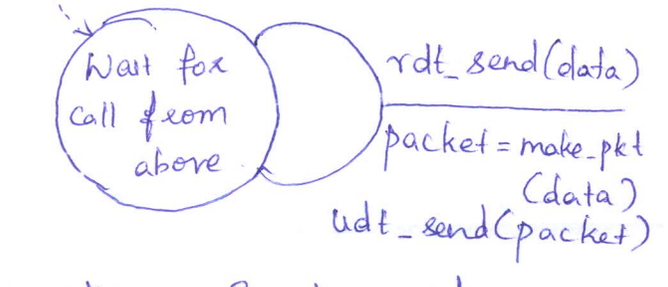
Reliable data transfer: Service Model & Service Implementation.

TCP provides - Reliable Data Transfer, but IP provides Unreliable Data Transfer. rdt - Reliable Data Transfer udt - Unreliable Data Transfer

Building a Reliable Data Transfer Protocol.

1. Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0

Finite-State-Machine (FSM) Definitions



a. rdt1.0: Sending side

b. rdt1.0: Receiving side.

→ Arrow Mark - transition of the protocol from one state to another.

→ The event causing the transition is shown above the horizontal line labeling the transition, & the actions taken when the event occurs are shown below the horizontal line.

→ Λ - no event / action.

→ All packet flow is from sender to receiver, wch with a perfectly reliable channel there is no need for the receiver to provide feedback to the sender. (no need for the receiver to ask the sender to slow down).

Reliable Data Transfer over a channel with Bit Errors: rdt2.0

Bits in the packets may be corrupted as a packet is transmitted, propagates or it is buffered.

Assumption: All transmitted packets are received in order (although their bits may be corrupted).

eg: In telephone conversation, the receiver may say 1. ok (+ve) ack). 2. Pls repeat (-ve ack).

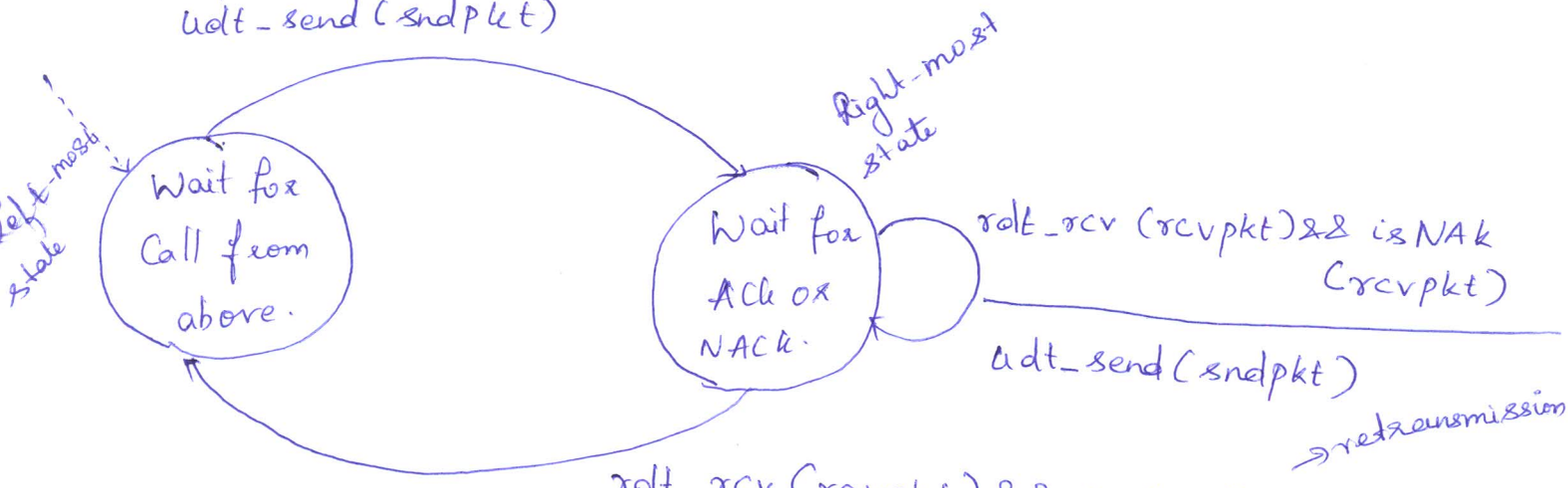
Reliable data transfer protocols based on such retransmission are known as Automatic Repeat reQuest (ARQ) protocols.

Additional capability required in ARQ to handle bit errors.

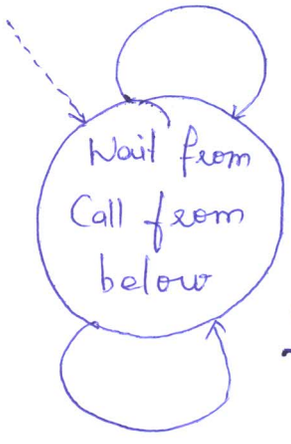
1. Error detection: checksum.
2. Receiver Feedback: 1 - Ack, 0 - [-ve Ack] NACK.
3. Retransmission: → by the sender.

rdt_send(data)

sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)



a) rdt 2.0: Sending side.



rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt = make_pkt(NAK)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && not corrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK)
udt_send(sndpkt).

b) rdt 2.0: Receiving side.

rdt 2.0 - A Protocol for a channel with bit errors.

→ When the sender is in the wait-for-ACK or NAK state, it cannot get more data from the upper layer. Because of this behavior, rdt 2.0 is known as stop-and-wait protocol.

→ Fatal Flaw in Protocol: ACK/NACK packets also may get corrupted.

→ Checksum bits need to be added to ACK/NACK packets in order to detect such errors. If it is corrupted the sender doesn't know whether the last piece of transmitted data has correctly received or not.

3 Possibilities for handling corrupted ACKs or NAKs.

1. If the speaker didn't understand "ok" or "Please repeat" reply from the receiver (i.e ACK / NAK), the speaker can ask, "What did you say" The receiver can repeat the reply. Else if again the speaker's "What did you say" is corrupted? → Problem.
2. Adding enough checksum bits to allow the sender not only to detect, but also to recover from, bit errors.
3. 3rd Approach is that the sender have to simply resend the current data packet when it receives a garbled ACK/NAK.

→ But it introduces duplicate packets into the channel. The receiver can't know apriori whether an arriving packet contains new data or is a retransmission.

→ Soln to this new pblm is to add a new field to the data packet & have the sender no' its data packets by putting a sequence no' into this field.

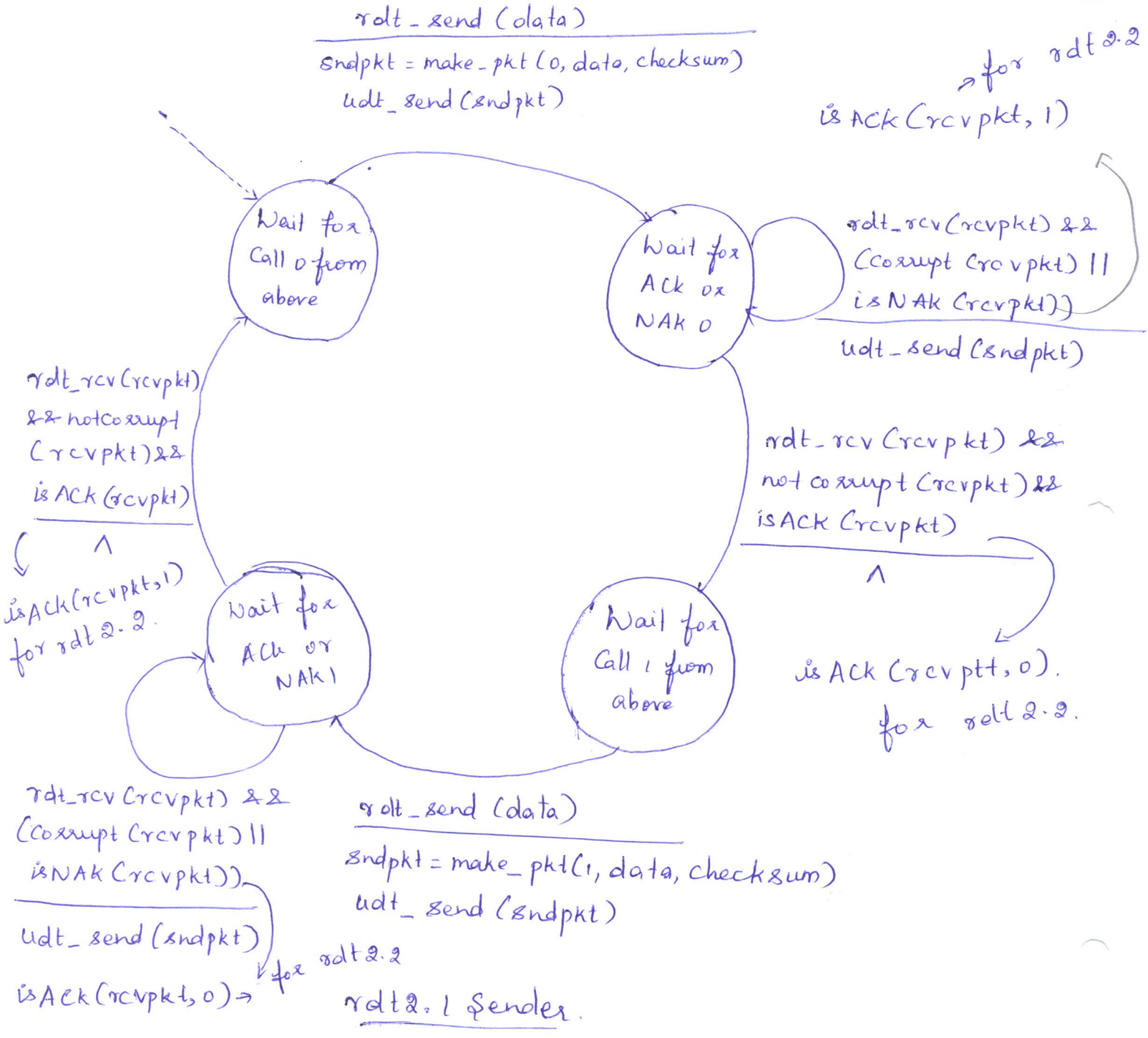
→ Then the receiver need to check only this sequence no' to known whether the received packet is a retransmission or not. 1-bit sequence no' is enough for stop & wait protocol.

→ Fixed version of rdt 2.0 is rdt 2.1.

→ Rdt 2.1 has more states for reflecting the sequence no' as 0 or 1.

→ rdt 2.1 protocol uses +ve & -ve acknowledgments.
+ve ACK - out of order packets
-ve ACK - Corrupted packets.

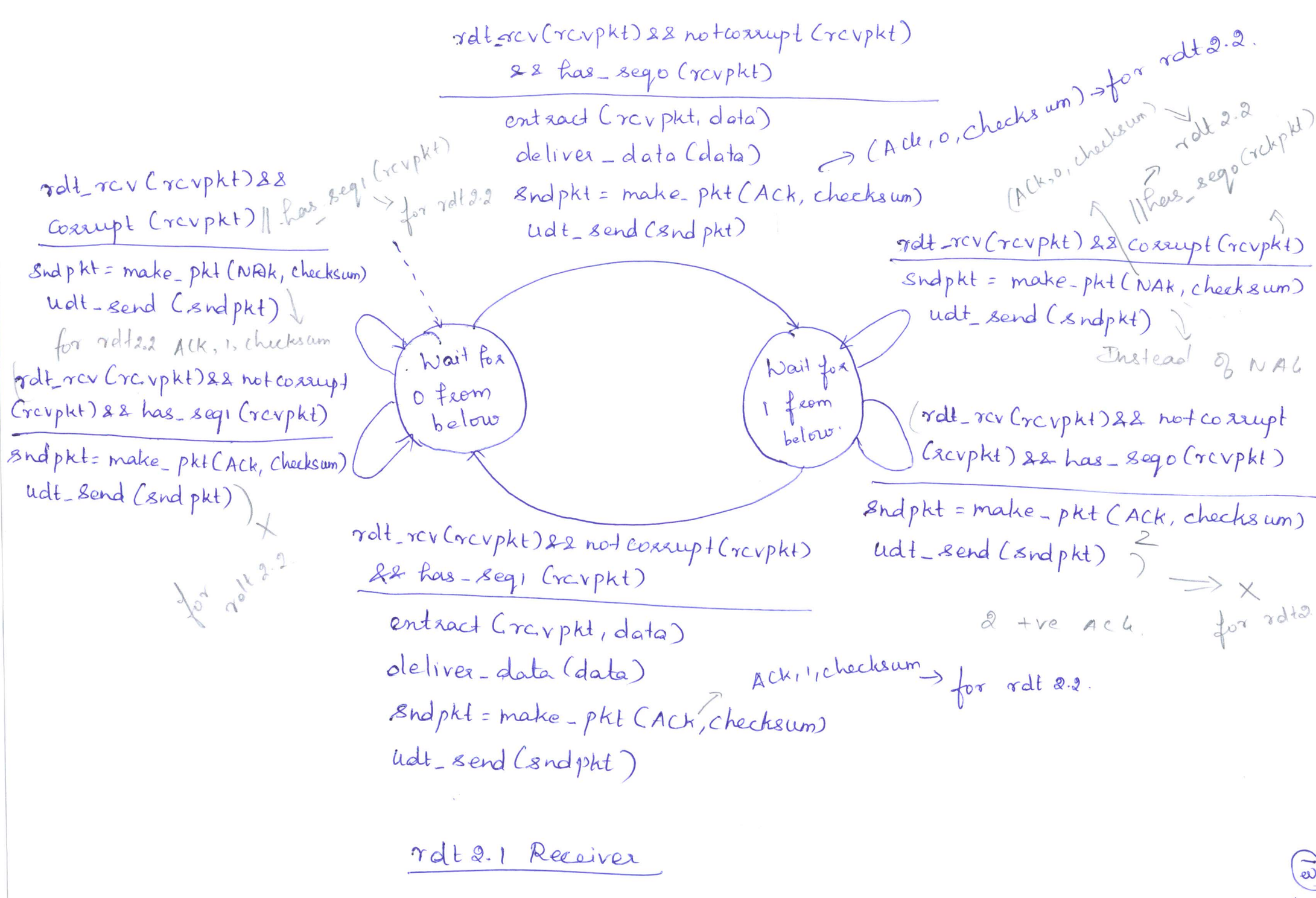
→ Instead of sending NAK, can send ACK for the last successfully received packet.



rdt2.1 Sender.

The sender getting duplicate ACk will identify that the receiver didn't receive the packet following the pkt that is being ACked twice.

Fnt rdt 2.2 protocol the ACk msg is given sequence numbers.



rdt 2.1 Receiver

Reliable Data Transfer over a Lossy channel with Bit Errors: rdt 3.0 (Alternate Bit Protocol).

→ Packet loss can be handled.

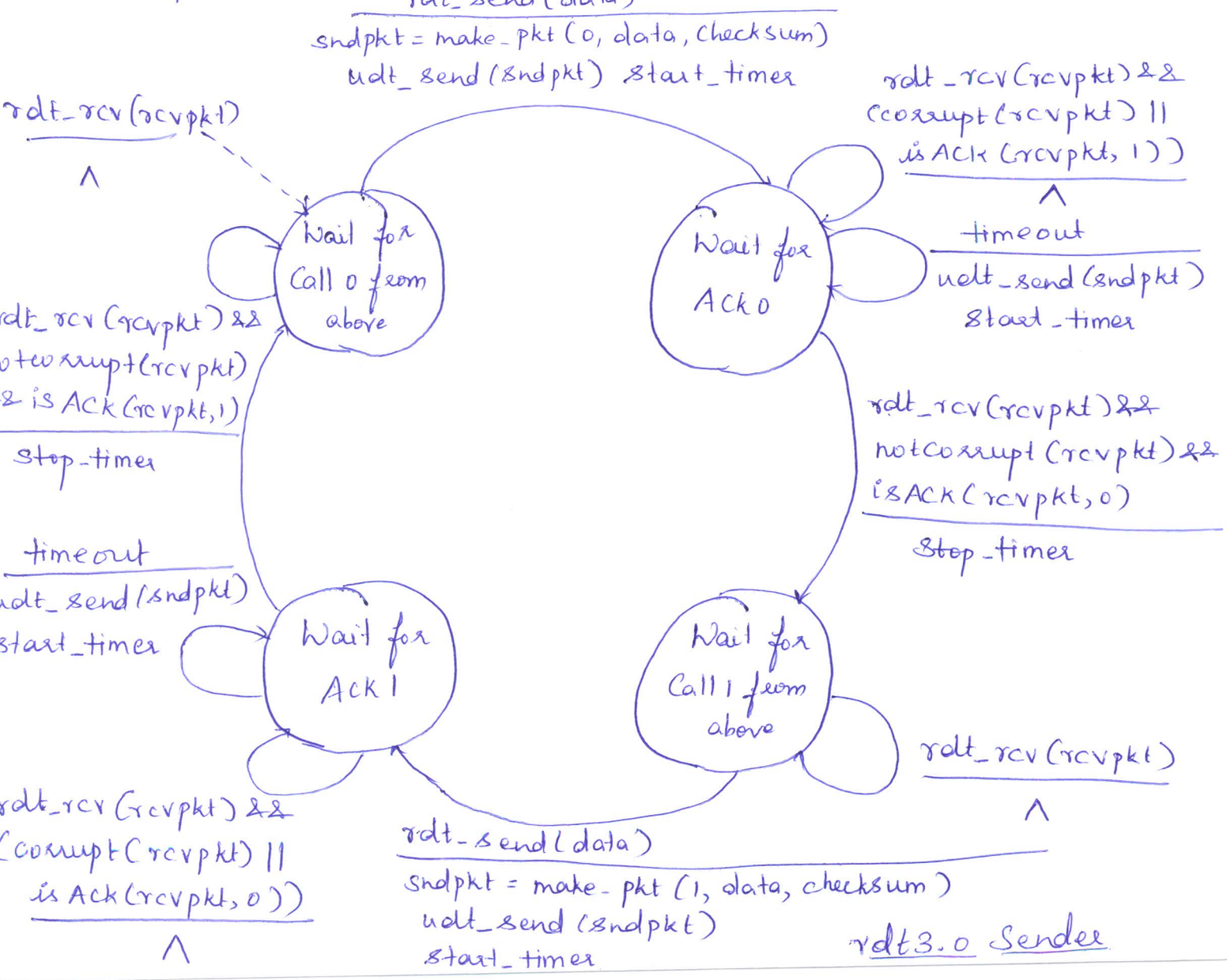
→ If the sender doesn't receive the ack, it doesn't know whether a data packet was lost, an Ack was lost, or if the packet / Ack got delayed. In order to overcome this Countdown timer is used.

→ The sender need to

i) Start the timer each time a packet is sent.

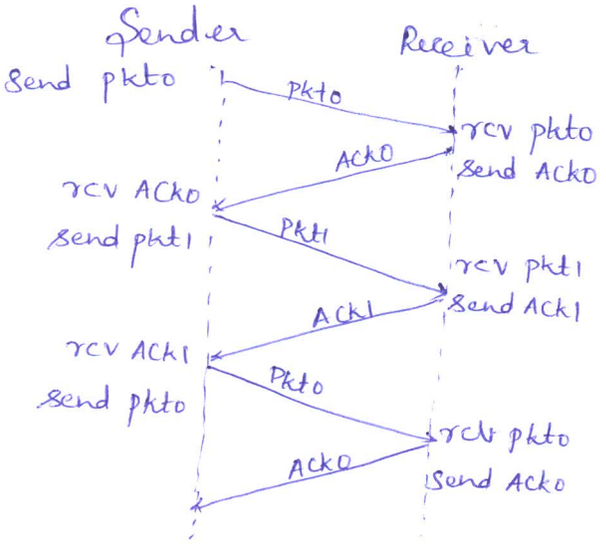
ii) Respond to a timer interrupt.

iii) Stop the timer

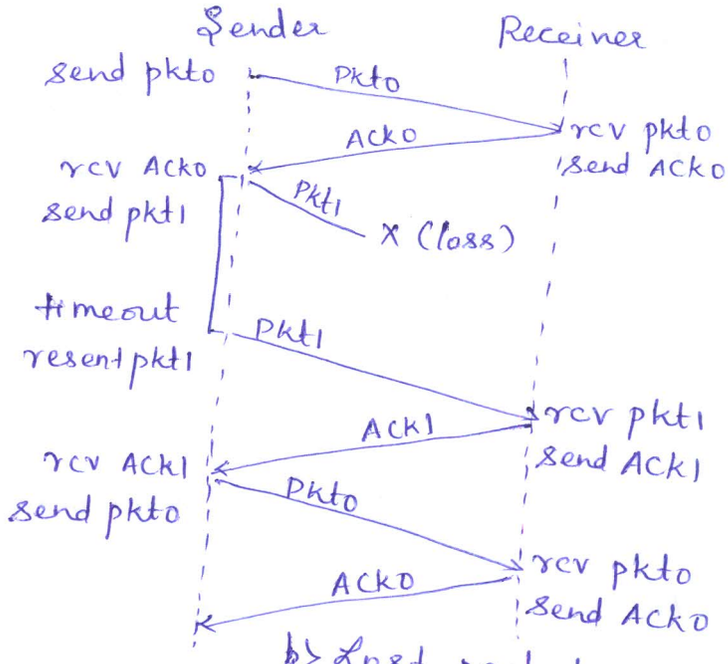


rdt3.0 Sender

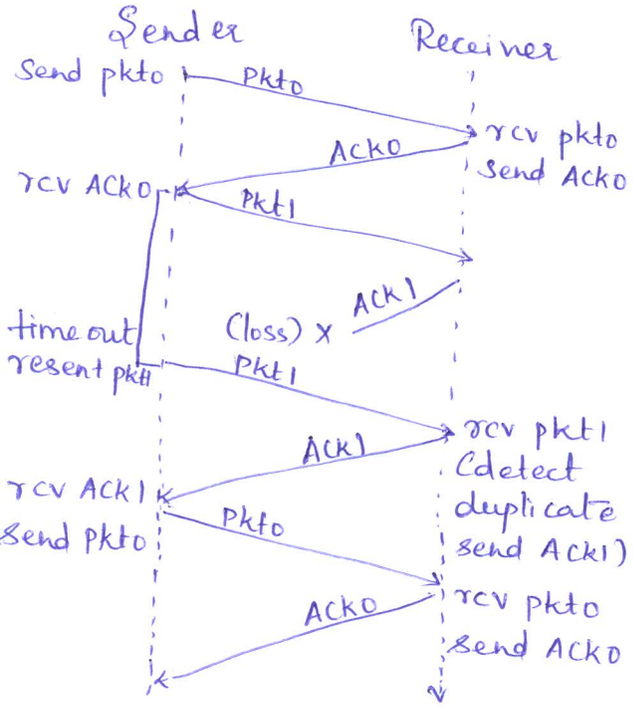
Pipelined Reliable Data Transfer Protocols



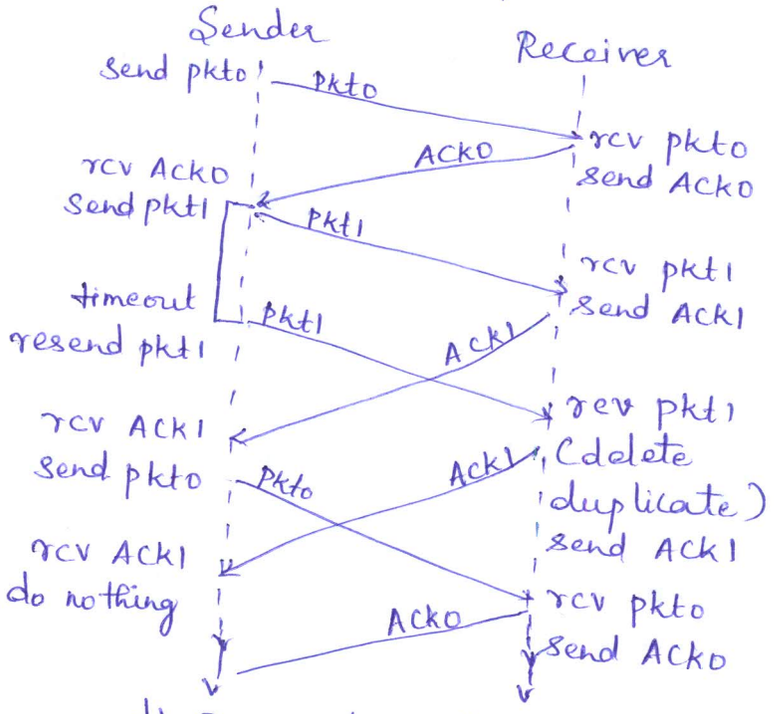
a) Operation with no loss



b) Lost packet



c) Lost Ack



d) Premature timeout

Operation of rdt3.0, the alternating-bit protocol

Pipelined Reliable Data Transfer Protocols

In Stop-and-wait manner, the transmission time is very high. Inorder to overcome this, the sender is allowed to send multiple packets (eg. 3) without waiting for acks.

Since the many in transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as pipelining.

Consequences:

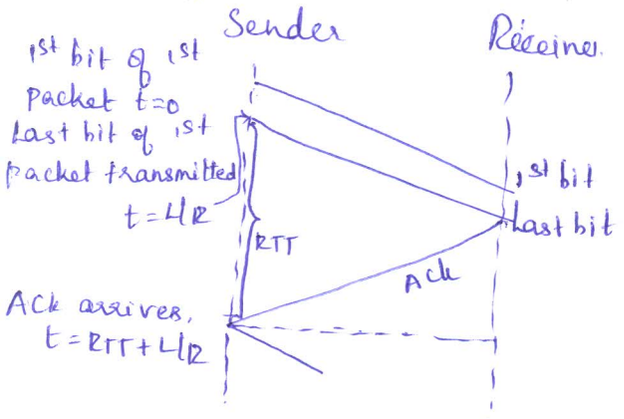
i) Range of sequence no' must be increased.

ii) Sender/Receiver sides have to buffer more than one packet.

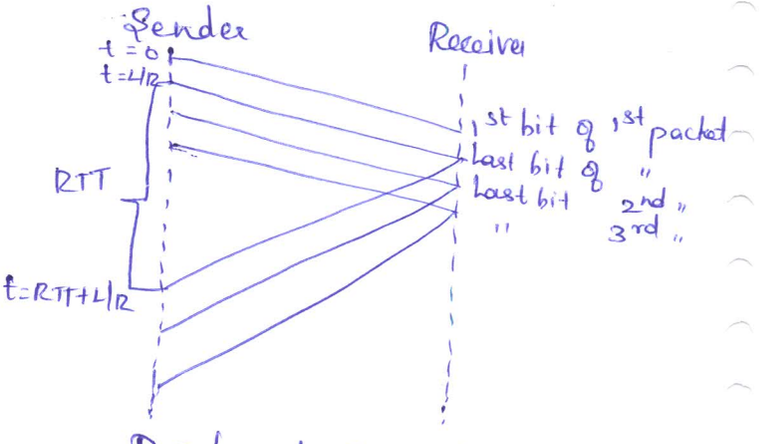
iii) How protocol responds to lost, corrupted, delayed packets

↳ Approaches for pipelined error recovery

a) Go-Back-N b) Selective repeat.



Stop-and-Wait Operation



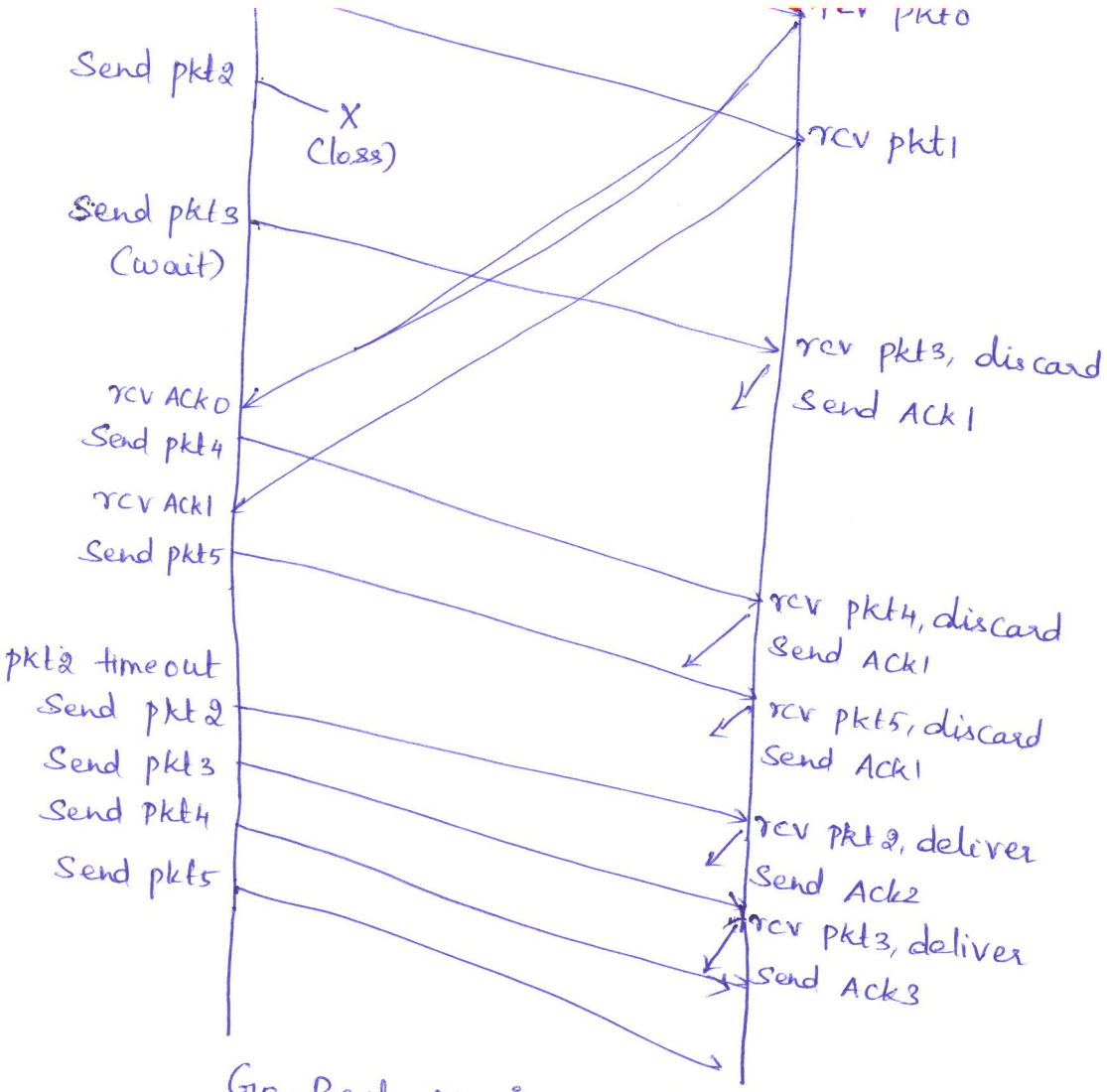
Pipelined Operation

Go-Back-N (GBN) Sliding-Window Protocol.

In a Go-Back-N (GBN) protocol, the sender is allowed to transmit multiple packets without waiting for an ack, but is constrained to have no more than some maximum allowable no', N, of unacknowledged packets in the pipeline.

Cumulative acknowledgment is given. If the packets are received out-of-order, then that packets are discarded.

↳ AckN → All packets upto N have received successfully.

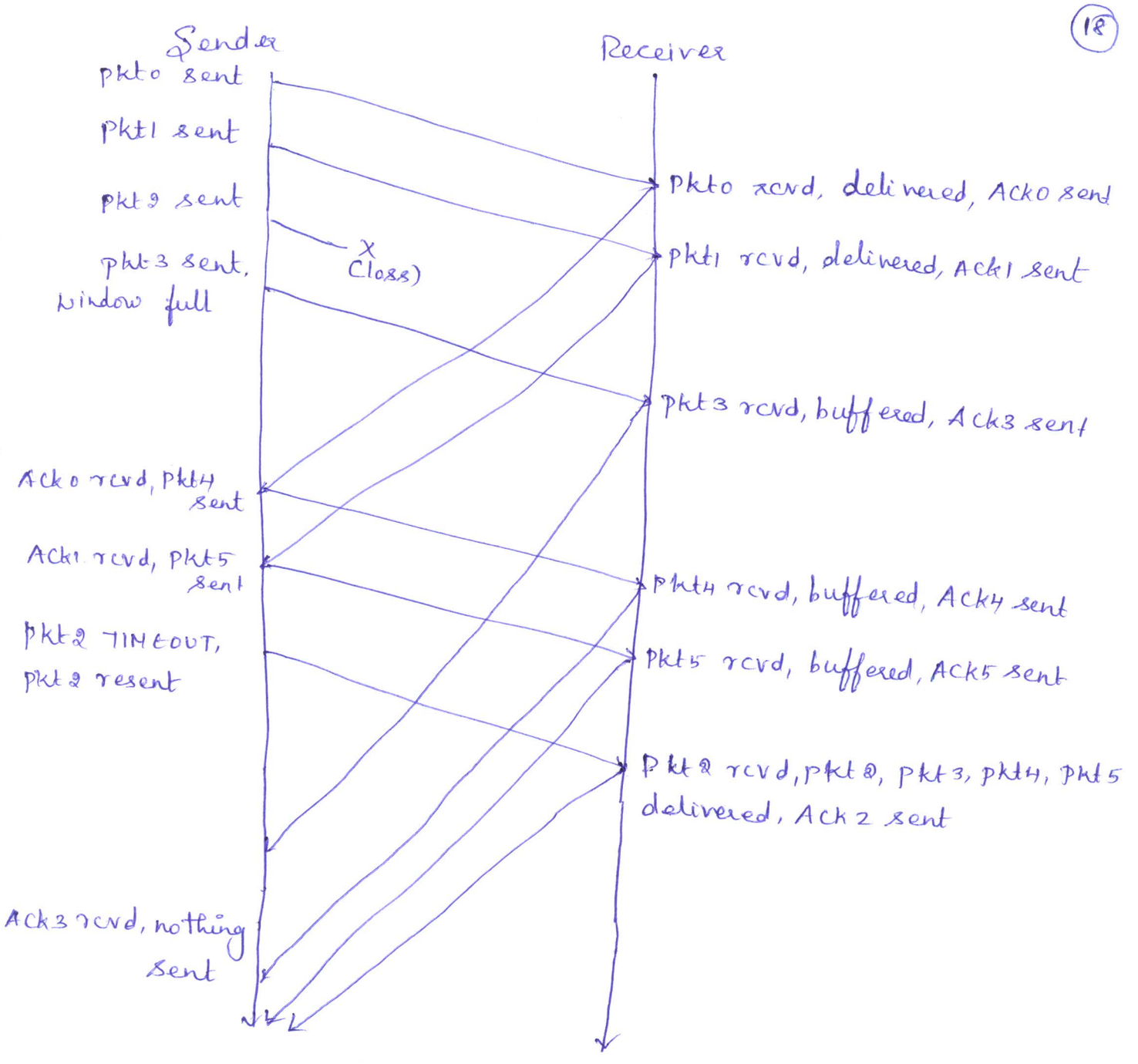


Go-Back-N in operation.

Selective Repeat (SR)

A single packet error can cause GBN to retransmit a large no' of packets, many unnecessarily. The SR receiver will acknowledge a correctly received packet whether or not in order.

Out-of-order packets are buffered until any missing packets are received, at which point a batch of packets can be delivered in order to the upper layer.



SR Operation

Connection-Oriented Transport: TCP

TCP relies on principles of reliable data transfer to provide including error detection, retransmission, cumulative acks, timers & header fields for sequence & ack no's.

The TCP Connection.

TCP is said to be connection-oriented because before one app' process can begin to send data to another, the 2 processes must 1st "handshake" with each other.

TCP connection provides full-duplex service.

TCP connection is point-to-point (single server & single receiver), Multicasting is not possible in TCP.

client → Socket s = new Socket ("localhost", 6666);

Server → ServerSocket ss = new ServerSocket (6666);

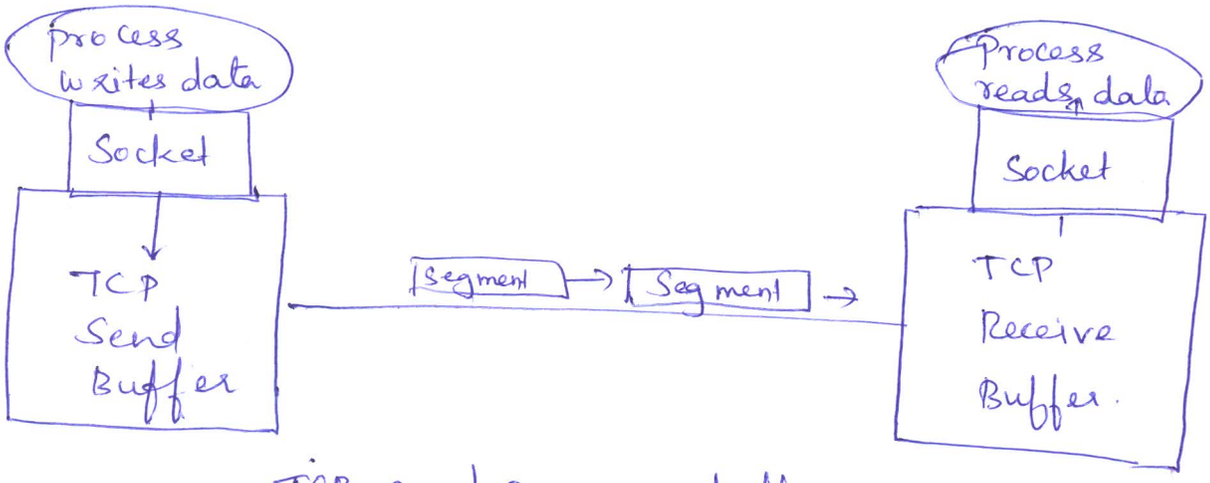
← Socket s = ss.accept(); ⇒ Handshaking.

Once a TCP connection is established, the 2 app' processes can send data to each other.

From time to time, TCP will grab chunks of data from the send buffer & pass the data to the nlw layer.

Maximum Segment size (MSS) → limit to transfer is typically set by determining the Maximum transmission Unit (MTU) of the link-layer frame.

TCP Segment = Each chunk of client data
+
TCP Header.

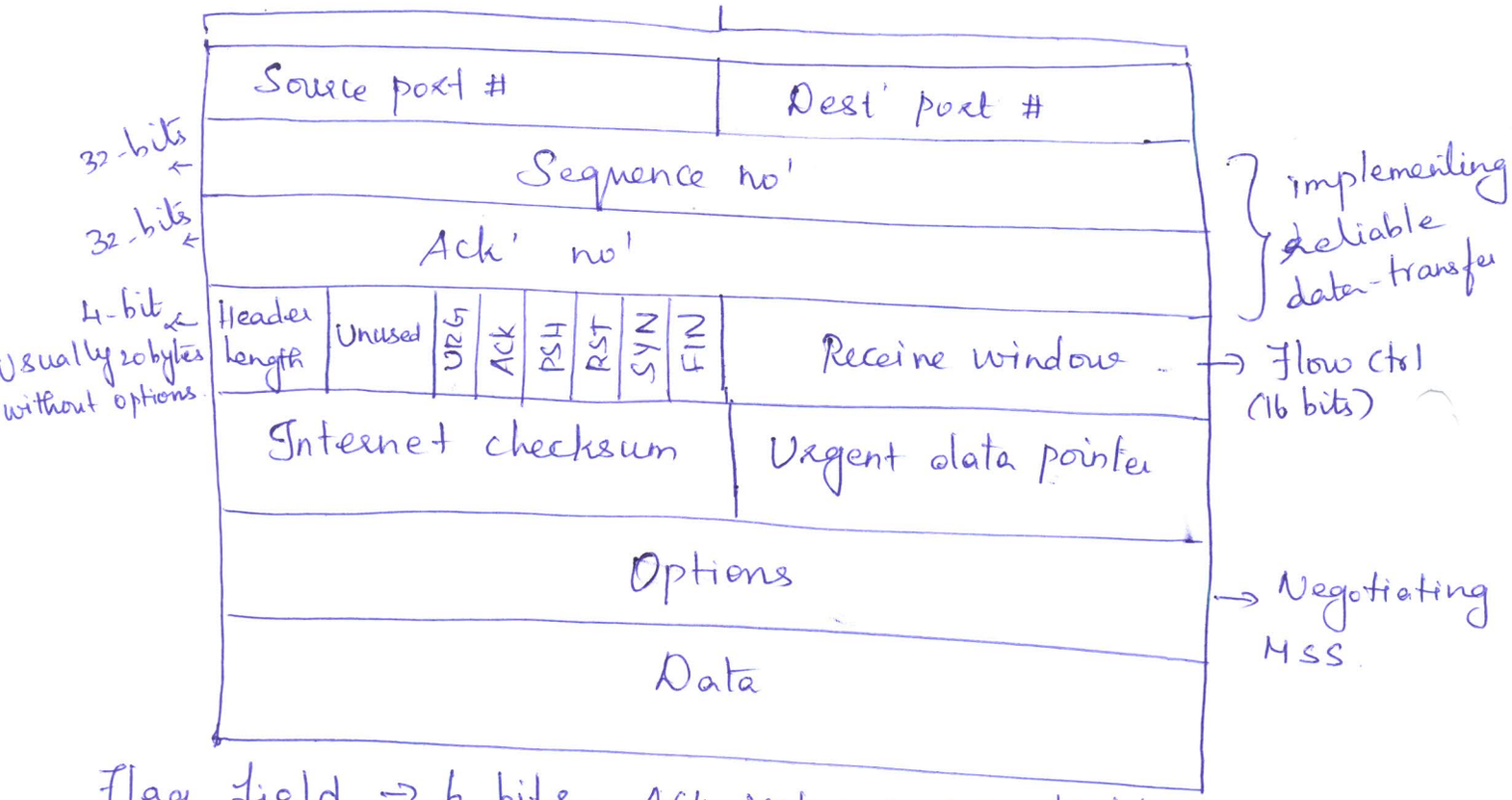


TCP send & receive buffers

The segment are passed to nlw layer, where they are encapsulated by IP headers

TCP Segment Structure [20 bytes]

UDP - 8 bytes



Flag field → 6 bits ACK - value in ACK field is valid. (segment successfully received)

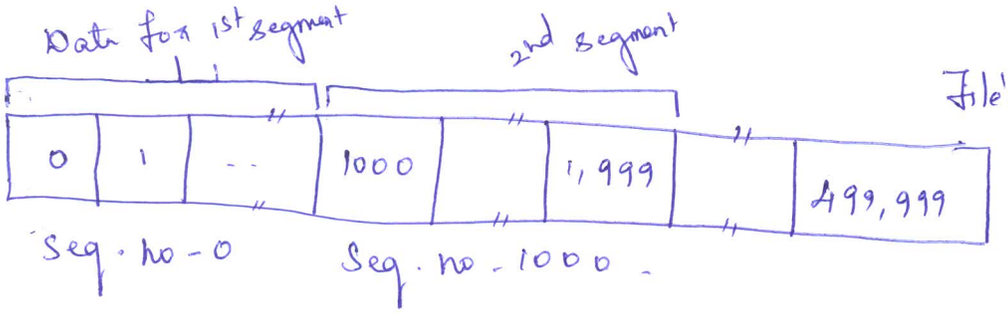
RST, SYN, FIN → connection setup & tear down. (Reset ← synchronize → finish)

PSH → receiver should pass the data to the upper layer immediately.

URG → Sending side upper layer has marked as urgent.
↓
location is provided by Urgent data ptr.

Sequence No's & Ack' No's

The Seq no' for a segment is therefore the byte-stream no' of the 1st byte in the segment.



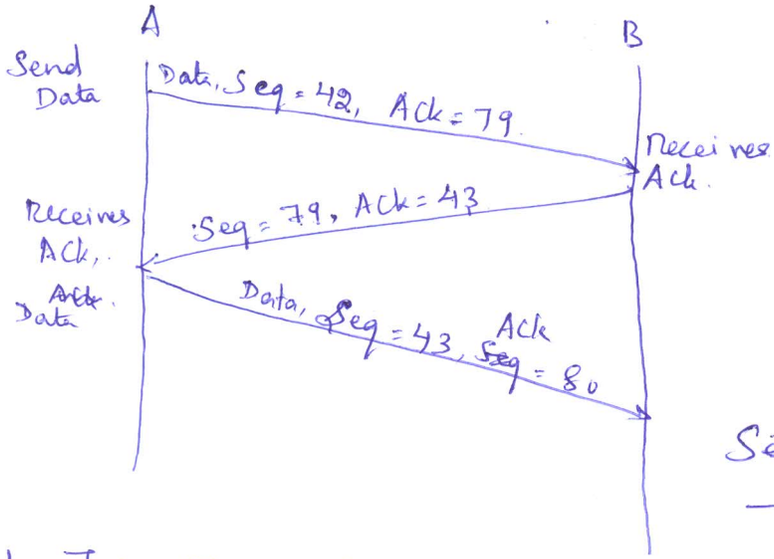
File's Total Bytes - 50,000
MSS - 1,000 Bytes

Dividing file data into TCP segments

The ack no' that Host A puts in its segment is the sequence no' of the next byte Host A is expecting from Host B.

TCP ack' bytes up to the 1st missing byte in the stream, TCP is said to provide cumulative ack'.

Out-of-packets - the receiver will wait for the missing bytes.



Seq & ACk No's

Round-Trip Time Estimation & Timeout.

The time-out should be larger than the connection's round-trip time (RTT), i.e the time from when a segment is sent until it is acknowledged.

Estimating the RTT

Sample RTT - amount of time blw when the segment is sent & when an ack' for the segment is received.

Sample RTT values will fluctuate from segment to segment.

Estimated RTT - Avg of sample RTT values.

$$\text{EstimatedRTT} = (1-\alpha) \cdot \text{EstimatedRTT}_{\text{previous value}} + \alpha \cdot \text{SampleRTT}$$

$$= 0.875 \cdot \text{EstimatedRTT} + 0.125 \cdot \text{SampleRTT} \quad [\alpha = 0.125 (1/8)]$$

This avg' is called an exponential weighted moving avg' [EWMA]

DevRTT - Diff' blw SampleRTT & EstimatedRTT.

$$\text{DevRTT} = (1-\beta) \cdot \text{DevRTT} \cdot \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

$\beta = 0.25$ ↓ deviates

Setting & Managing the Retransmission Timeout Interval

Timeout interval >= Estimated RTT, but not too much larger. (because TCP can't retransmit the segment) leading to large retransmission delays.

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

Reliable Data Transfer

3 major events related to transmission & retransmission in the TCP sender.

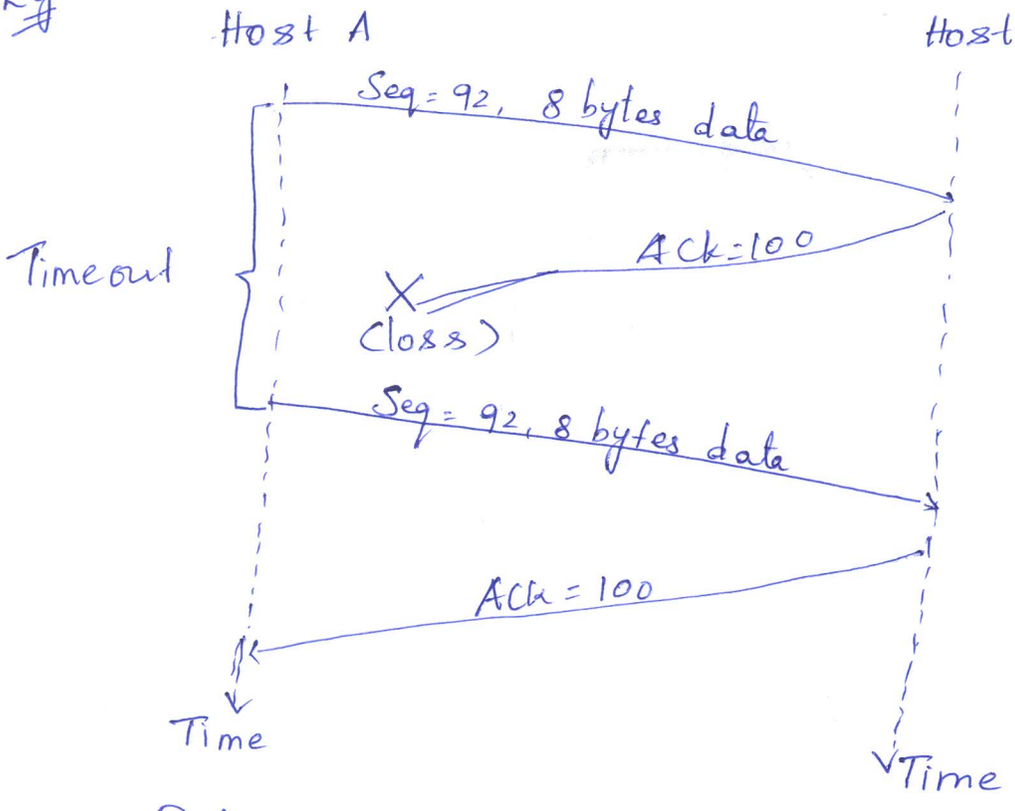
1. Data received from app' above. (seq.no).
2. Timer timeout.
3. ACK receipt.

NextSeqNum = InitialSeqNumber, SendBase = InitialSeqNumber
(next byte to sent) (unacknowledged byte)

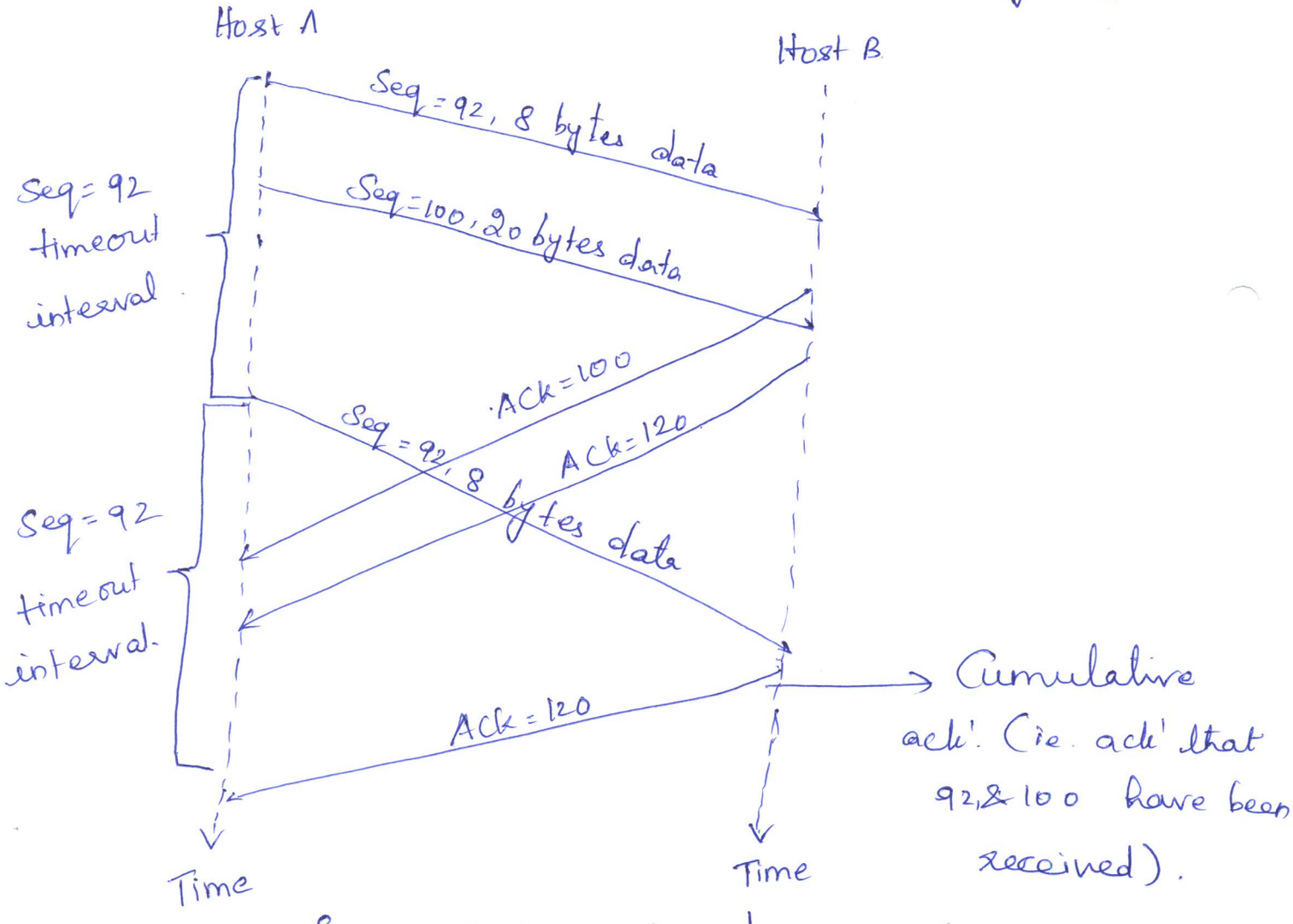
```
loop (forever)
{
 switch(event)
 event: data received from app' above
 Create TCP segment with seq no' NextSeqNum
 if (timer currently not running)
  start timer
 pass segment to IP
  NextSeqNum = NextSeqNum + length(data)
  break;
 event: timer timeout
 retransmit not-yet-acknowledged segment each with
 smallest sequence no.
 start timer
 break;
 event: Ack received, with ACK field value of y
 if (y > SendBase) → seq no' of oldest unacknowledged
 {                                    byte
  SendBase = y
 if (there are currently any not-yet-acknowledged
                                    segments)
  start timer
 }
 break;
}
```

Simplified TCP Sender

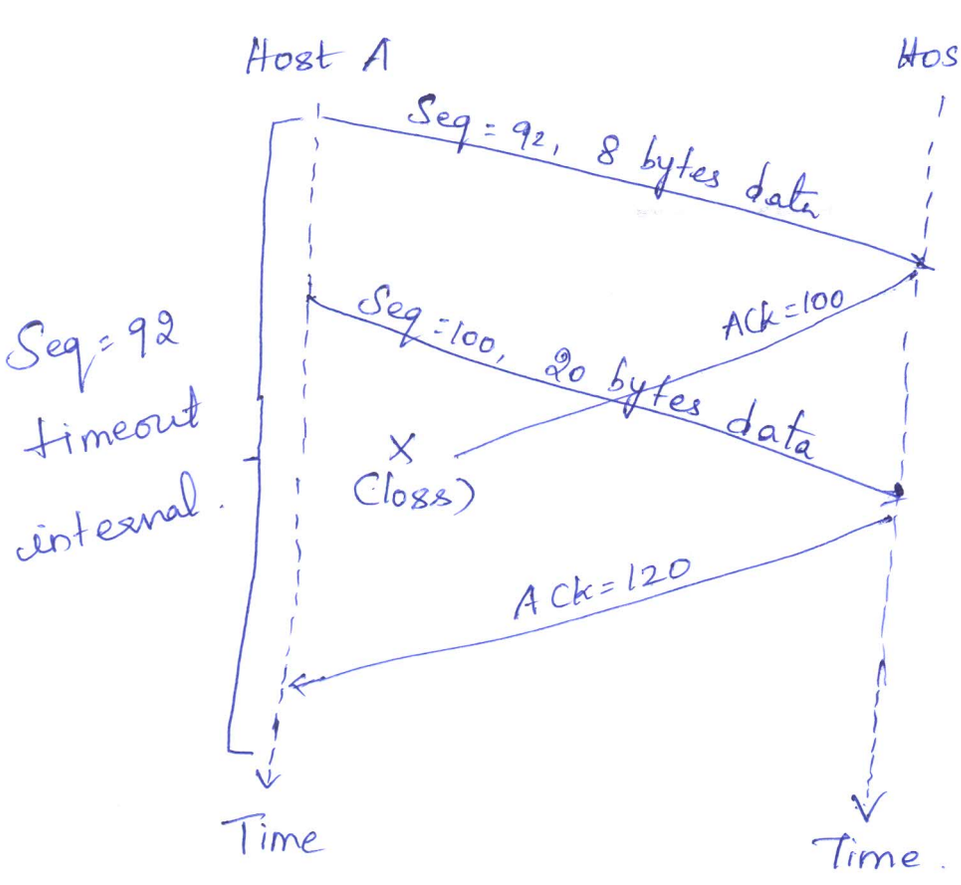
Eg:



Retransmission due to a lost acknowledgment



Segment 100 not retransmitted



A Cumulative acknowledgment avoid retransmission of the 1st segment.

Doubling the Timeout Interval.

After retransmiting, TCP sets the next timeout interval to twice the previous value, rather than deriving it from the last EstimatedRTT & DevRTT.

eg: Timeout Interval for oldest not yet acknowledged segment is = .75, after retransmission then time is 1.5 sec, then if again timer expires, new expiration time is 3.0 sec.

Thus the intervals grow exponentially after each retransmission. This also provides limited form of congestion ctrl by avoiding unnecessary retransmissions.

Fast Retransmit

If the timeout interval is too long then it delays the sender's lost packet retransmission.

The sender can often detect packet loss before the timeout event occurs by Duplicate ACK.

A duplicate ACK is an ACK that reacknowledges a segment for which the sender has already received an earlier ack.

TCP ACK Generation.

Event	TCP Receiver Action
1. Arrival of in-order segment with expected seq no. All data upto expected seq no already acknowledged.	Delayed ACK. Wait upto 500 msec for arrival of another in-order segment. If next in-order segment doesn't arrive in this interval, send an ACK.
2. Arrival of in-order segment with expected sequence no. One other in-order segment waiting for Ack transmission	Immediately send single cumulative ACK, ACKing both in-order segments
3. Arrival of out-of-order segment with higher-than-expected sequence no. Gap detected.	Immediately send duplicate ACK, indicating seq. no of next expected byte.
4. Arrival of segment that partially or completely fills in gap in received data.	Immediately send ACK, provided that segment starts at the lower end of gap.

TCP doesn't use -ve ack', instead it simply reacknowledges the last in-order byte of data it has received.

If the sender gets 3 duplicate ACKs (ie. the segment has been lost 3 times), then the sender performs fast retransmit, ie retransmits the missing segment before the segment's timer expires.

Replacement to ACK received event

```
event: ACK received, with ACK field value of y
    if (y > SendBase) {
    SendBase = y
    if (there are currently any not yet acknowledged segments)
    start timer }
    else { /* a duplicate ACK for already ACKed segment*/
    increment no' of duplicate ACKs received for y
    if (no' of duplicate ACKs received for y == 3)
    /* TCP fast retransmit */
    resend segment with sequence no' y
    }
    break;
```

Go-Back-N / Selective Repeat.

Selective ack - TCP receiver to ack' out-of-order segments selectively rather than just cumulatively acknowledging the last correctly received, in order segment.

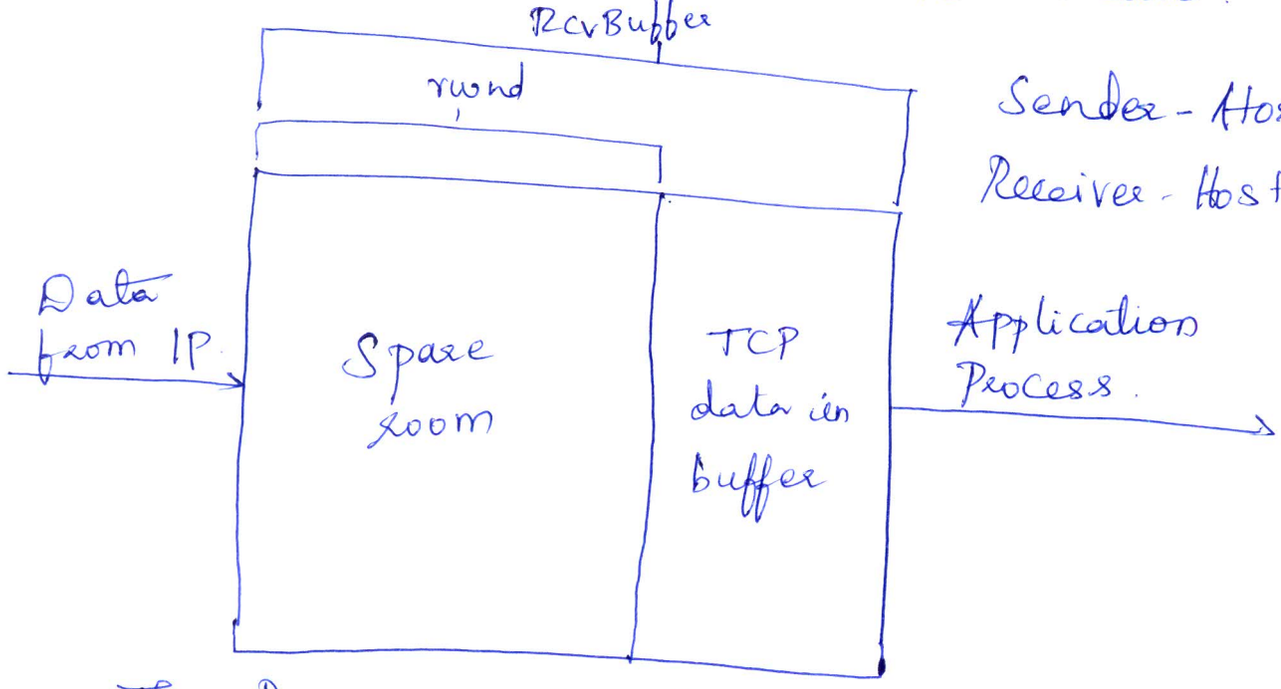
Selective retransmission can be combined with this.

Flow Control

TCP provides a flow-ctrl service to its app' to eliminate the possibility of the sender overflowing the receiver's buffer.

Flow ctrl is a speed-matching service - matching the rate at which the sender is sending against the rate at which the receiving app' is reading.

TCP TCP provides flow ctrl by having the sender maintain a variable called the receive window.



The Receive Window (rwnd) & the receive Buffer (RcvBuffer)

Variables

LastByteRead: the no' of the last byte in the data stream read from the buffer by the app' process in B.

LastByteRcvd: the no' of the last byte in the data stream that has arrived from the nlw & has been placed in the receive buffer at B.

Because TCP is not permitted to overflow the allocated buffer,

LastByteRcvd - LastByteRead ≤ RcvBuffer.

rwnd is set to the amount of spare room in the buffer.

rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]

Host B tells Host A how much spare room it has in the connection buffer by placing its current value of rwnd in the receive window field of every segment it sends to A. A makes sure that

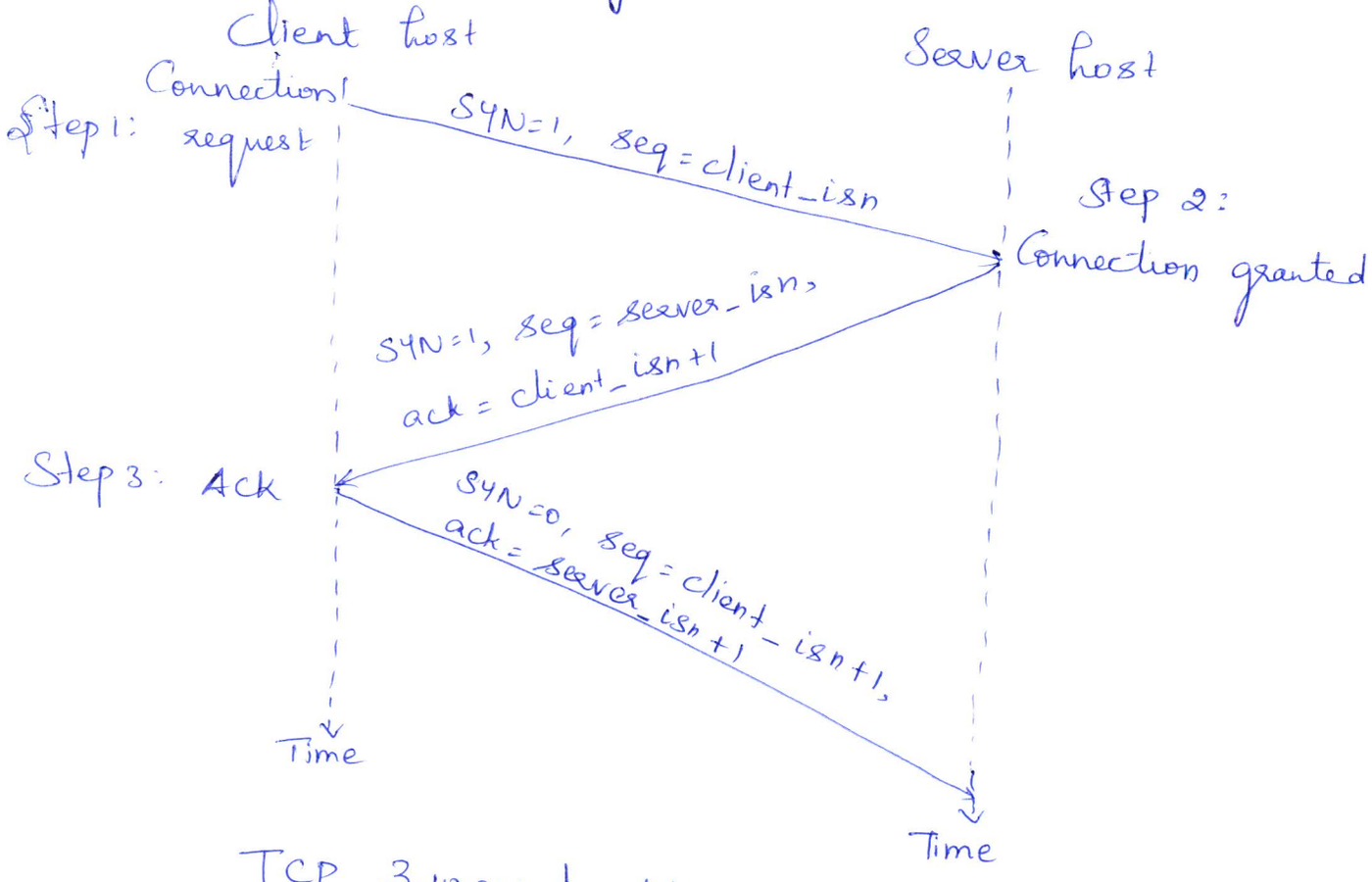
LastByteSent - LastByteAcked ≤ rwnd.

Pblm:

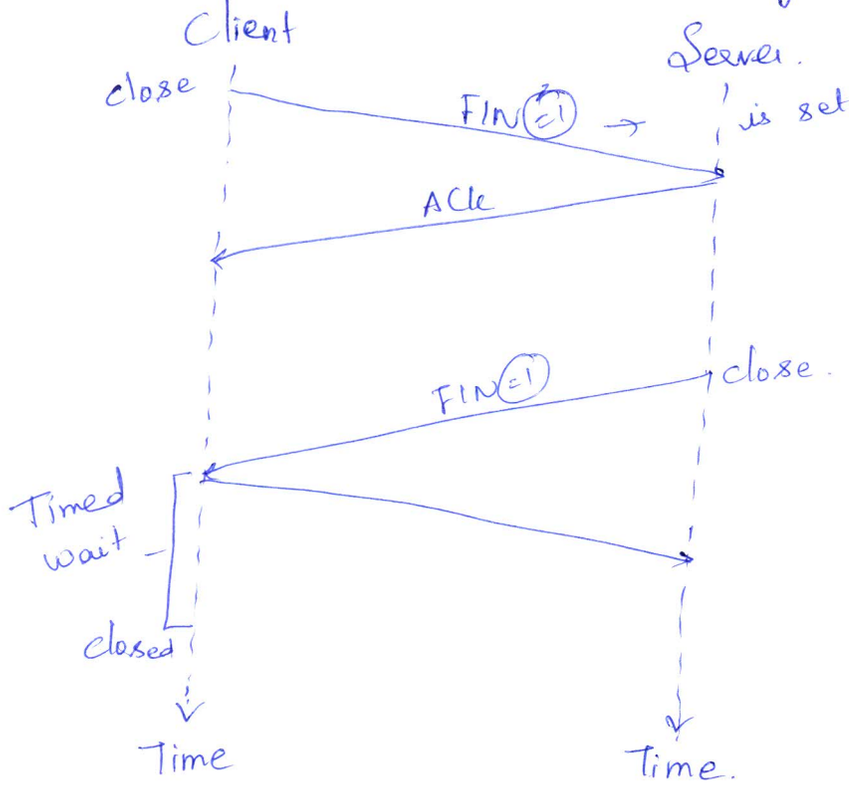
If B's buffer) rwnd has space, but B has nothing (data, ack) to send to A, then A is blocked & don't send any data.

To resolve this issue, A needs to send one data byte to check whether receive window is zero.

TCP Connection Management.



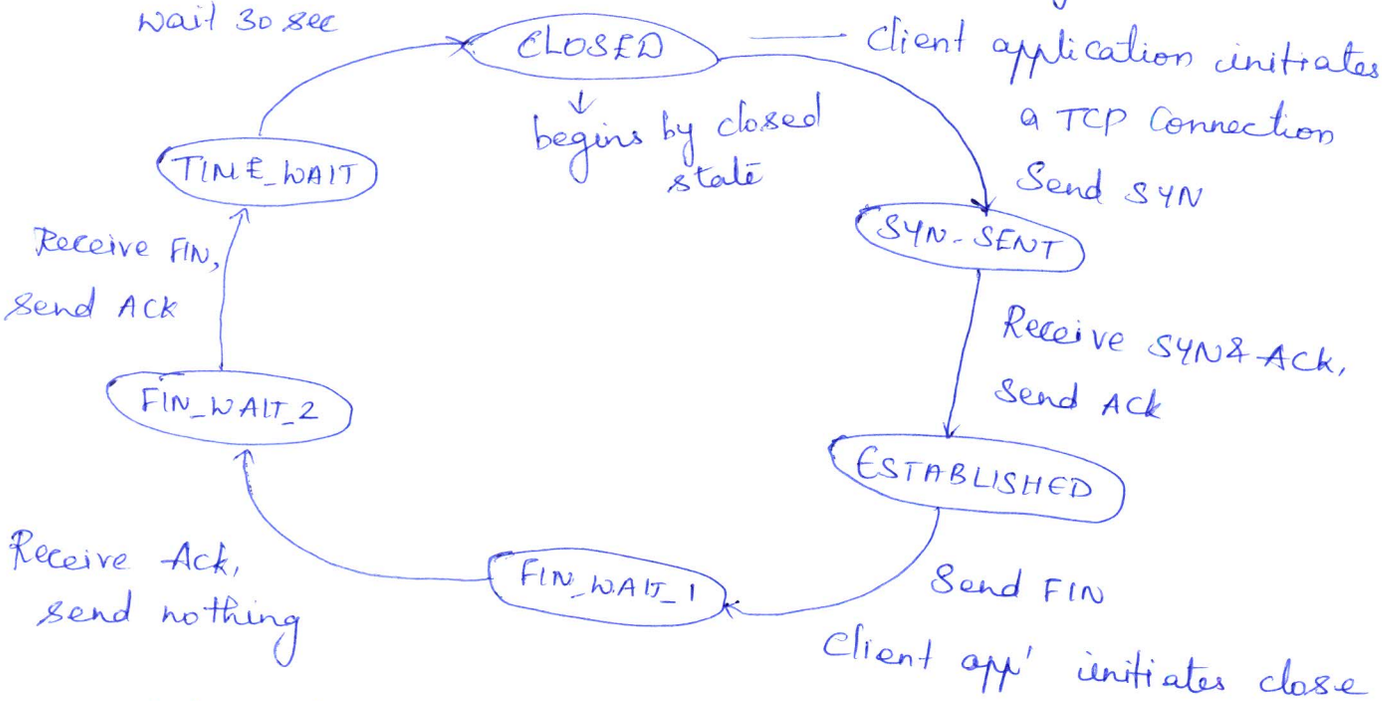
TCP 3-way handshake: segment exchange.



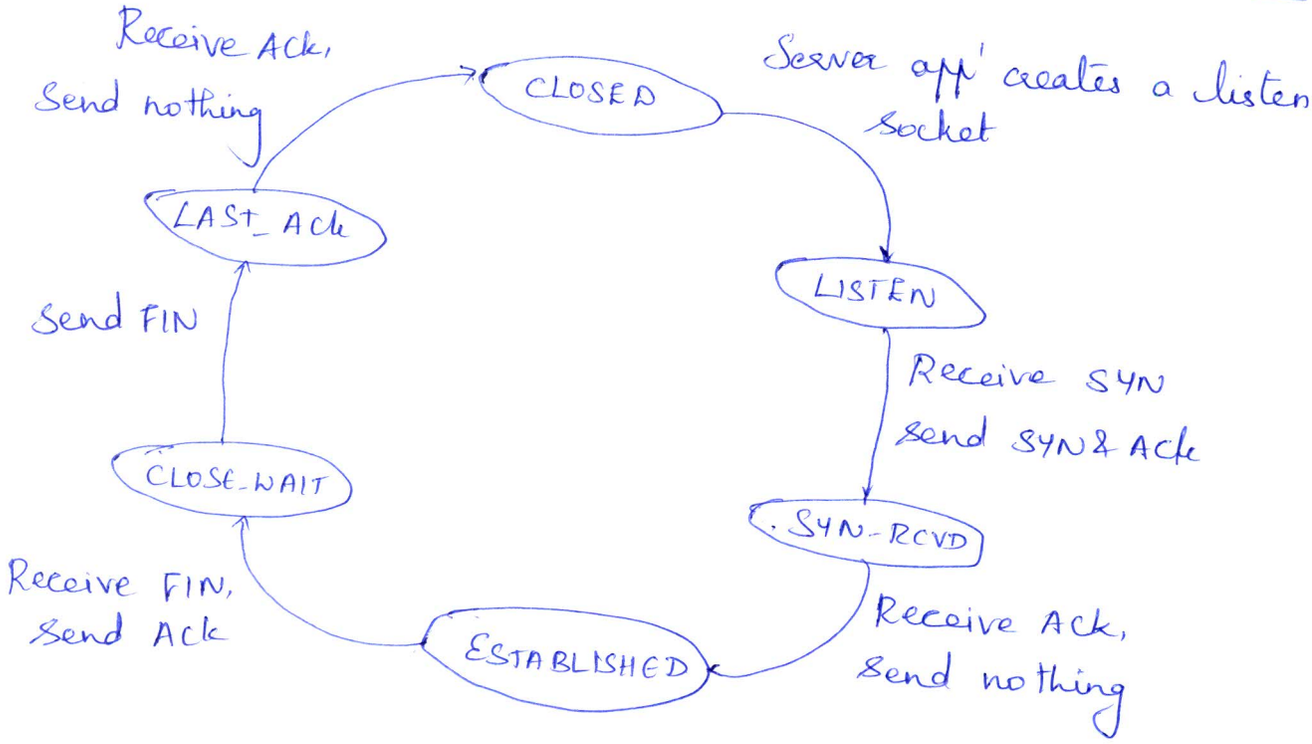
If the server host didn't accept the connection, then it will send RST (reset) segment to the source. In case of UDP, ICMP (Internet Control Msg Protocol) is send.

Closing a TCP Connection

During the life of a TCP connection, the TCP protocol running in each host make transitions through various TCP states.



A typical sequence of TCP states visited by a client TCP



A typical sequence of TCP states visited by a Server-side TCP

Principles of Congestion control

A TCP sender can also be throttled due to congestion within the IP nlw, this form of sender ctrl is referred to as congestion ctrl.

The Causes & the costs of Congestion

Scenario 1: 2 Senders, a Router with Infinite Buffers

Avg sending rate - λ_{in} bytes/sec.

Link Capacity (in Router) - R.

For a sending rate blw 0 & R/2. the throughput at the receiver = the sender's sending rate.

Sending rate is above R/2, receiver throughput = R/2.

If sending rate T above R/2. the avg delay become infinite

Scenario 2: 2 Sender & a Router with Finite Buffers

Unneeded retransmissions by the sender in the face of large delays may cause a router to use its link bandwidth to forward unneeded copies of a packet.

Scenario 3: 4 Senders, Routers with Finite Buffers & Multihop paths

Another cost of dropping a packet due to congestion - when a packet is dropped along a path, the transmission capacity that was used at each of the upstream links to forward that packet to the point at which it is dropped ends up having been wasted.

Approaches to congestion ctrl

i) End-to-end Congestion ctrl: Nlw layer doesn't provide any support to the transport layer for congestion ctrl. TCP segment loss (timeout or a triple duplicate ack) is taken as an indication of nlw congestion & TCP decreses its window size accordingly.

ii) Nlw-assisted Congestion ctrl: Nlw layer components (routers) provide explicit feedback to the sender regarding the congestion state in the nlw. eg: ABR

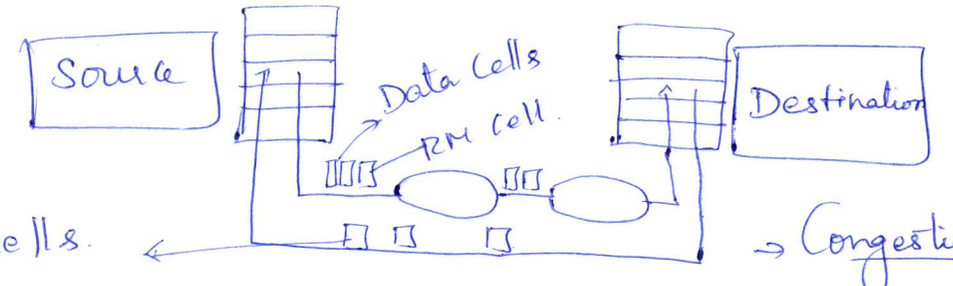
Congestion inf' is typically feedback from the nlw to the sender in one of 2 way.

1. Direct feedback → By choke packet, from nlw router to the sender.
2. When a router updates a field in a packet to indicate congestion

ATM Available Bit Rate (ABR) Congestion ctrl.

ATM- Asyncheronous Transfer Mode.

ATM takes a virtual-circuit (VC) oriented approach towards packet switching. This allows a switch to track the behavior of individual senders & to take source-specific congestion ctrl actions.



→ Congestion ctrl Framework

ABR has been designed as an elastic data transfer service, when the nlw is underloaded, it will take spare available bandwidth, when the nlw is congested, it will throttle to minimum transmission rate.

ABR : packets - cells. Routers - Switches.

RM (Resource Mgmt) cells are used to convey congestion-related info among the hosts & switches.

RM can be send from receivers or switches.

Mechanisms for Signaling congestion-related info to the receiver.

1. EFCI (Explicit forward Congestion Indication) bit:
   Switch will set EFCI bit to 1, in a data cell to indicate Congestion to the receiver. Then receiver sets RM CI bit in RM to 1.

2. CI & NI bits. RM cell contains CI & NI bits
   CI - Congestion Indication - Severe Congestion
   NI - No Increase - Mild Congestion

3. ER Setting (Explicit Rate) field.
   RM cell contains ER field - minimum supportable rate.

An ATM ABR source adjusts the rate at which it can send cells as a fun of the CI, NI & ER values in a returned RM cell.

TCP Congestion ctrl

The TCP congestion-ctrl mechanism operating at the sender keeps track of an additional variable, the congestion-window.

The Congestion window (cwnd), imposes a constraint in the rate at which a TCP sender can send traffic into the n/w.

The amount of unacknowledged data at a sender may not exceed the minimum of cwnd & rwnd, i.e.

LastByteSent - LastByteAcked $\leq$ min{cwnd, rwnd}

Sender's send rate cwnd/RTT bytes/sec. By adjusting the value of cwnd, the sender can therefore adjust the rate at which it sends data into its connection.

TCP uses ack' to trigger its increase in congestion window size, TCP is said to be self-clocking.

Principles

→ A lost segment implies congestion & hence, the TCP sender's rate should be decreased when a segment is lost.

→ An acknowledged segment indicates that the n/w is delivering the sender's segments to the receiver, & hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment.

→ Bandwidth probing

The TCP sender increases its transmission rate to probe for the rate that at which congestion onset begins, backs off from that rate & then to begins probing again to see if the congestion onset rate has changed.

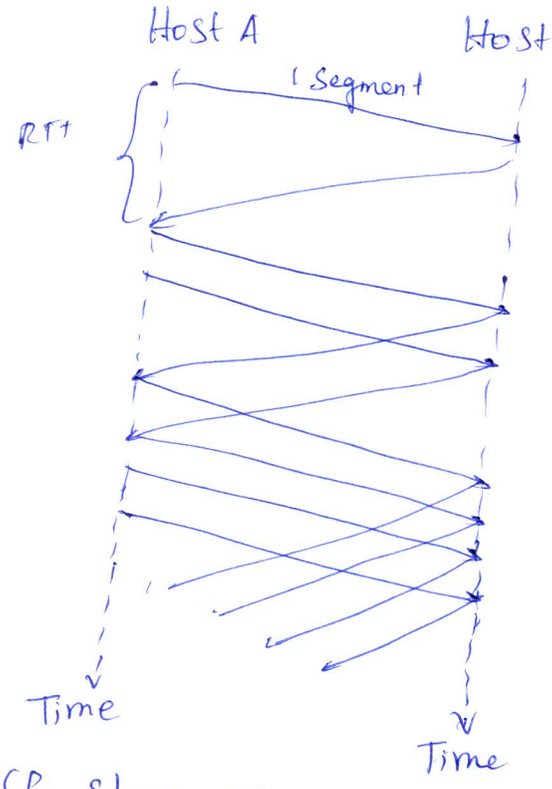
TCP Congestion-Ctrl Algorithm

Components: 1) Slow start 2) Congestion Avoidance 3) Fast recovery

Slow start

The congestion window size is set to 1. (1 segment is sent). If ack' received cwnd = 2, then 4, 8, ... exponentially it will be increased until → threshold reaches & increase linearly until Congestion occurs. If congestion is detected then again cwnd = 1 & threshold is set to cwnd/2.

If cwnd = threshold then Congestion avoidance mode.
If 3 duplicate ACKs are detected, TCP performs fast retransmit & enters the fast recovery mode.



TCP Slow Start.

Congestion Avoidance:

Once the threshold is reached the cwnd increases linearly. When triple duplicate ACKs were received then threshold = cwnd/2.

Fast Recovery

After retransmission of the lost segments the cwnd will grow exponentially. eg: TCP Tohoe, TCP Reno.