Chapter-3 Transport layer

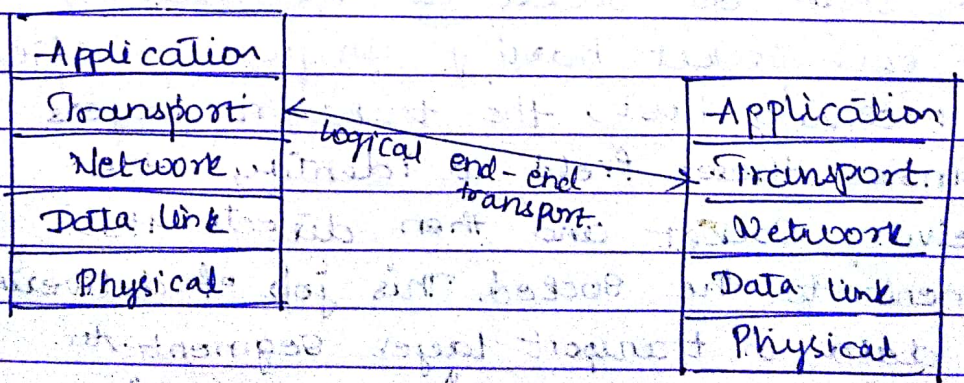
★ Transport layer services.

→ Provides logical communication between application process running on different hosts.

→ Application process uses the logical communication provided by the transport layer to send messages to each other, free from the worry of the details of the physical infrastructure used to carry the msg.

→ A working example:-

- On the sender side the transport layer converts the appl layer msg to transport layer packets (segments), this is done by breaking the message to smaller chunks and adding transport layer header.
- then the transport layer passes the segment to n/w layer,
- where the segment is encapsulated with in the n/w layer packet (datagram) and sent to destination
- At the reciever end the n/w layer extracts the transport layer segment from datagram and passes the segment to the transport layer
- The transport layer proces the segment there by making the data in the segment available for the recieving application.



* Relationship between Transport and Network layer

→ Transport layer protocol provides logical comm^{n} between processes running on hosts.
→ N/w layer protocol provides logical comm^{n} between hosts.

Text Book

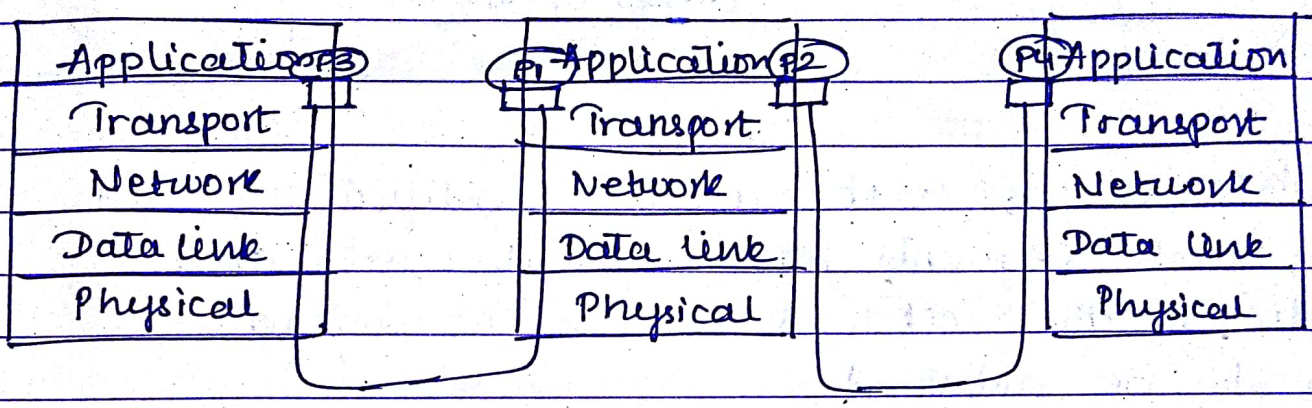
* Internet - transport layer protocols

→ TCP
→ UDP

* Multiplexing and Demultiplexing

— A process can have one or more 'sockets' through which data passes from n/w to the process and through which data passes from the process to the network
→ The transport layer ~~doesnot~~ in the receiving host does not deliver data directly to a process, but instead to a intermediate socket
→ Bcoz at any given time there can be more than one socket at the receiving hos with each socket having unique identifier
→ At receiving host, the transport layers examines these field to identify the receiving socket and then directs the segment to the socked. This job of delivering the data in transport layer segment to correct socket is called "demultiplexing"

→ The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information to create segments, and passing the segments to the network layer is called "Multiplexing"

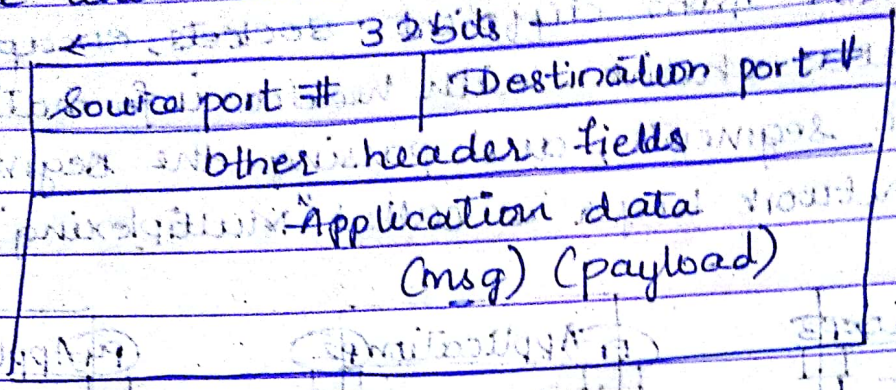


process

Socket

→ The transport layer in the middle host shown in the figure must gather ~~data from~~ outgoing data from these sockets, form transport layer segments and pass these segments down to the network layer.

Socket and destination port number in transport layer



The sockets have unique identification and the each segments have special feilds and indicate the socket to which the segment is to be delivered

This special feilds are the "source port number feild" and the "destination port number feild".

Each port number is a 16bit-number ranging from 0 to 65535

The port numbers ranging from 0-1023 are called "well known port numbers" and are restricted. ie they can be used by only application protocols such as HTTP, FTP

Connectionless Multiplexing and Demultiplexing:

clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)

↳ UDP packets are created in this way

- The transport layer assigns a port number in the range 1024 to 65535 that is currently not being used by any other UDP port in the host.
- If the developer is writing the code where he is implementing the server side of a "well-known protocol", then the developer would have to assign the corresponding "well-known port number".
- The client side of the application lets transport layer automatically assign the port number where as server side application assigns a specific port number.

→ Ex

- Suppose a process in Host A with UDP port 19157 wants to send a chunk of data to process at UDP port 46428 in host B.
- The transport layer in Host A create a transport layer segment with source and destination port number, data and other segments
- The transport layer then passes the resulting segment to network layer.
- The network layer encapsulates in the IP datagram and makes best effort for delivering.
- If the segment is recieved at the destination host, the host examines the port number and forwards the packet to the same port numbers
- Suppose if two segments have same port numbers

then the 2 segments will be directed to same destination port.

Connection - Oriented Multiplexing & Demultiplexing

- TCP socket is identified by four tuple ie source IP address, source port number, destination IP address, destination port number.
- Thus when a TCP segment arrives from the network to a host, the host uses all 4 values to direct the segment to the appropriate socket.

Working Ex:

- The TCP server application has a "welcoming socket", that waits for connection establishment request from TCP clients on port number 12000.
- TCP client creates socket and sends a connection establishment request segment with lines
- connection establishment request is TCP segment with destination port number
- When the host ~~of~~ running the server process receives the incoming connection request seg with destination port 12000 It locates the server process that is waiting to accept a connection on port number 12000 The server then creates new socket

- the transport layer at the server notes the following 4 values
  1) Source port number in segment
  2) IP address of the source host
  3) destination port number in the segment
  4) Its own IP address

Connectionless transport UDP

- Apart from multiplexing / demultiplexing function add's some light error checking it adds nothing to the IP.
- IF application developer uses UDP over TCP then they are directly talking to the IP
- UDP takes messages from the application process attaches some sourc and destination port number and some othe small fields and passes the resulting segment to n/w layer.
- The n/w layer encapsulates the segment to IP datagram and makes an best effort to deliver the datagram to the receiving host.
- There is no handshaking between sending and receiving transport layer entities. for this reason UDP is called "connectionless".
- DNS is an example that typically uses DNS.
- When DNS application in a host wants to make a query, it constructs a DNS query messages and passes the message to UDP
- Without performing any handshaking with the UDP entity running on the destination end system, the host side UDP adds header fields to the message and passes the resulting segment to the n/w layer

- The n/w layer encapsulates the data to datagram and sends it to a name server.
- The DNS application at the querying host then waits for a reply to its query. If it doesnot receive any reply, either it tries sending the query to another name server or ~~invokes the~~ informes the present application that it cant get a reply.

UDP segment structure

32bits

Source port #	dest^n. port #
Length	Checksum
Application data (message)	

- The UDP header has only 4 fields each of two bytes
- The port numbers allow the destination host to pass the application data to the correct process running on the destination end system.
- The length feild specifies the number of bytes in the UDP segment (header + data) this is required bcoz ~~the~~ data length in UDP may vary from segment to segment.
- The checksum is used by receiving host to check whether error has been introduced.
- The application data feild occupies the data feild of the UDP segment.

UDP Checksum

- provides for error detection
- determine whether bits within the UDP segment have been altered as it is moved from source to destination
- UDP at the sender side performs 1's compliment of the sum of all the 16 bit words in the segment.
- any overflow encountered during sum is wrapped arround
- This result is put in the checksum field of the UDP segment.

- Ex:

Consider following 3 16-bit words

1. 0110011001100000
2. 0101010101010101
3. 1000111100001100

Sum of first two 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 1 1 1 0 1 1 1 0 0 1 0 1 0 1
```

Adding third word

```
  1 0 1 1 1 0 1 1 1 0 0 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 0 1 0 0 0 0 1
↳                               1
  0 1 0 0 1 0 1 0 1 0 1 0 0 0 1 0
```

↳ 1's Comp

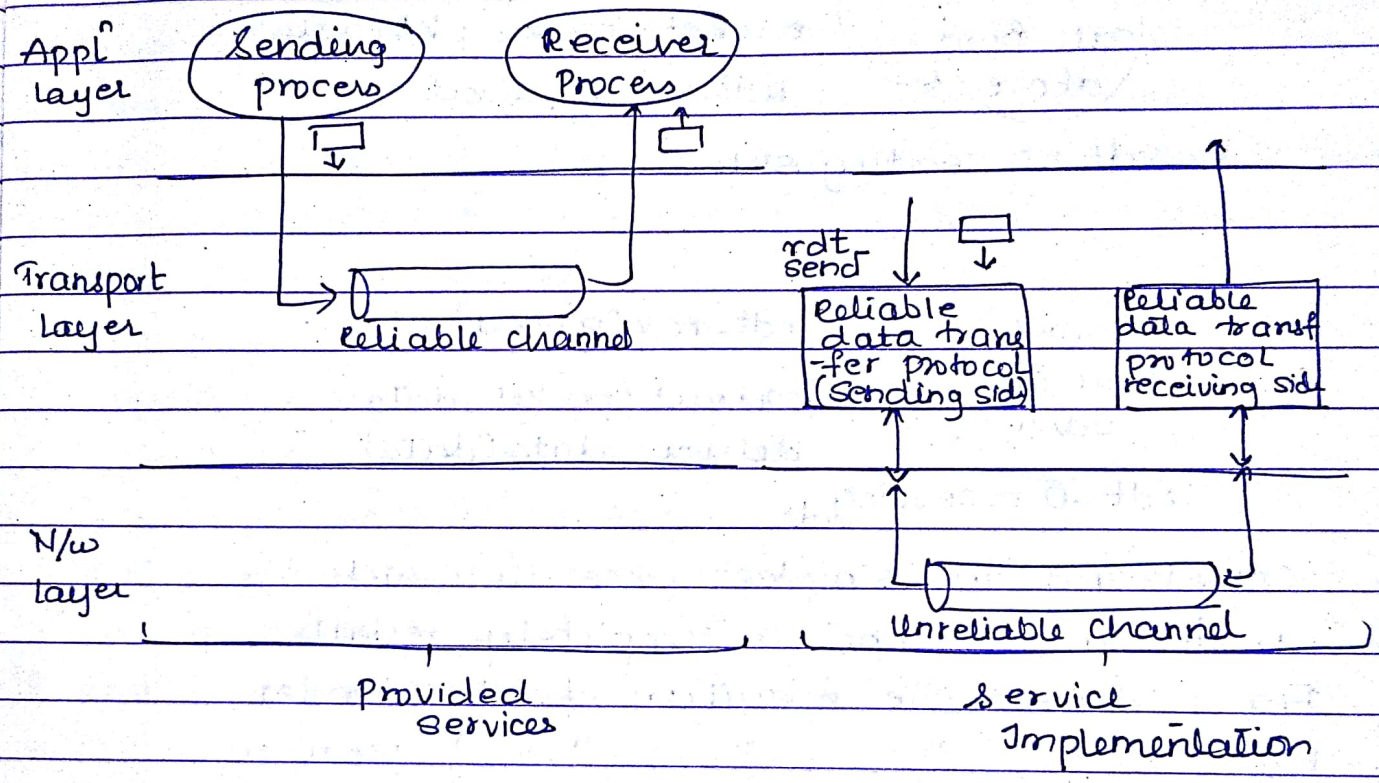
1011010101011101 ⟹ Checksum

Adding all the words along with checksum gives 1111111111111111 any 0 in this indicates error has been introduced in the packet.

* Why UDP provides a checksum...?
- Many link layer provides error checking
- But the reason is that there is no gurantee that all the links bet^n source and destination provide error checking.
- UDP must provide error detection at the transport layer on an end-end basis. if the end-end data transfer service is to provide error detection
- UDP provides error checking but it does not do anything to recover from an error.
- Some implementations of UDP simply discard the damaged segment,
- while some segment pass the damaged segment to the application with a warning

Principles of Reliable Data Transfer

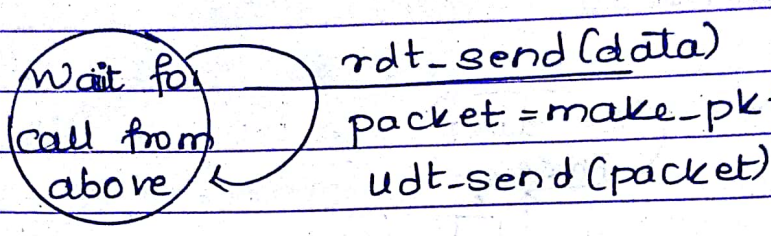
- In this section a reliable data transfer protocol is developed at sender and receiver side, considering increasingly complex models of the underlying channel.



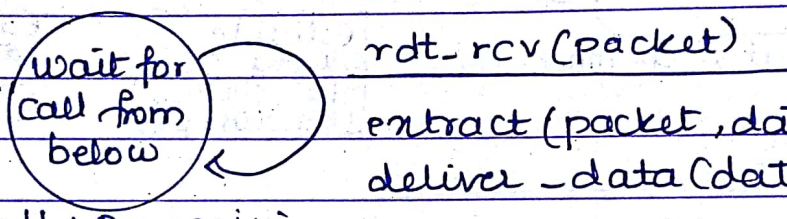
- The sending side of the data transfer protocol will be invoked from above by a call to rdt_send().
- It will pass the data to be delivered to the upper layer at the receiving side
- On the receiving side rdt_rcv() will be called when a packet arrives from the receiving side of the channel.
- When rdt protocol wants to deliver data to the upper layer it will do so by calling deliver_data()

Building a Reliable Data Transfer Protocol

Reliable Data Transfer over a Perfectly Reliable channel rdt 1.0



a. rdt 1.0 sending side



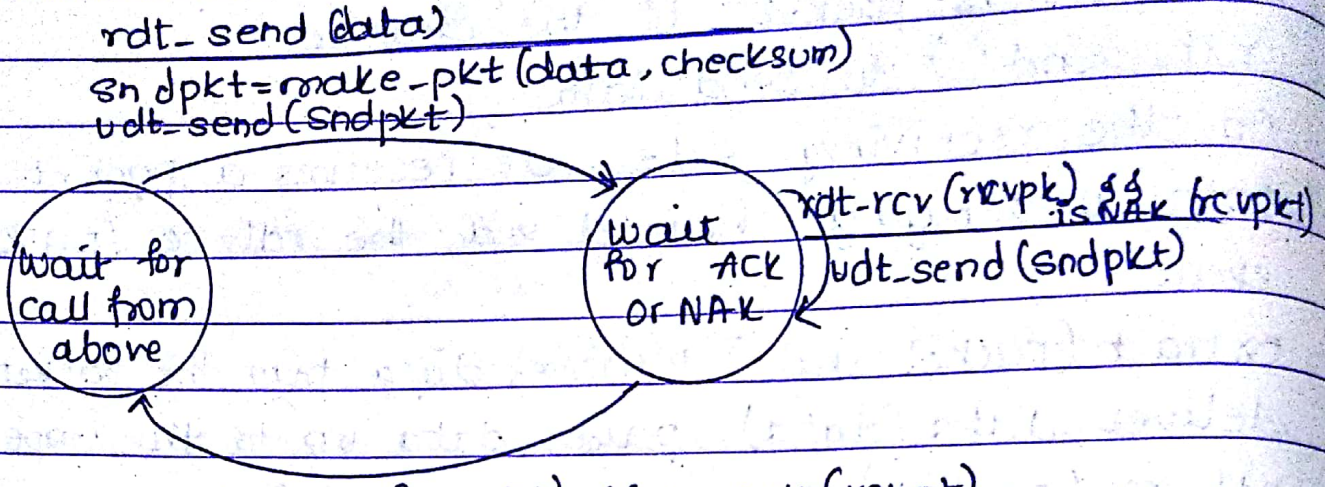
rdt 1.0 receiving side

- Considering the simplest case in which the underlying channel is completely reliable.
- Fig 1 defines the operation of the sender
- Fig 2 " " " " " receiver
- Its imp to note that there are seperate FSM for both sender and receiver, each has only one state
- The event causing the transition is shown above the horizontal line and the event causing is given below the horizontal line
- When there is no event or action taking place it is denoted by the symbol Λ below or above the horizontal respectively to denote the lack of action

- The sending side of rdt simply accepts data from the upper layer via rdt_send(data) event,
- creates packet containing the data make_pkt(data) and sends packet to the channel.
- rdt_send() to send data
- On the receiving side, rdt receives a packet from underlying channel via the rdt_rcv(packet) event,
- extract(packet, data) removes data from the packet
- deliver_data(data) passes data up to the upper layer.
- rdt_rcv(packet) event would result from a procedure call from a lower layer
- where as rdt_send(data) is bcoz of upper layer application.

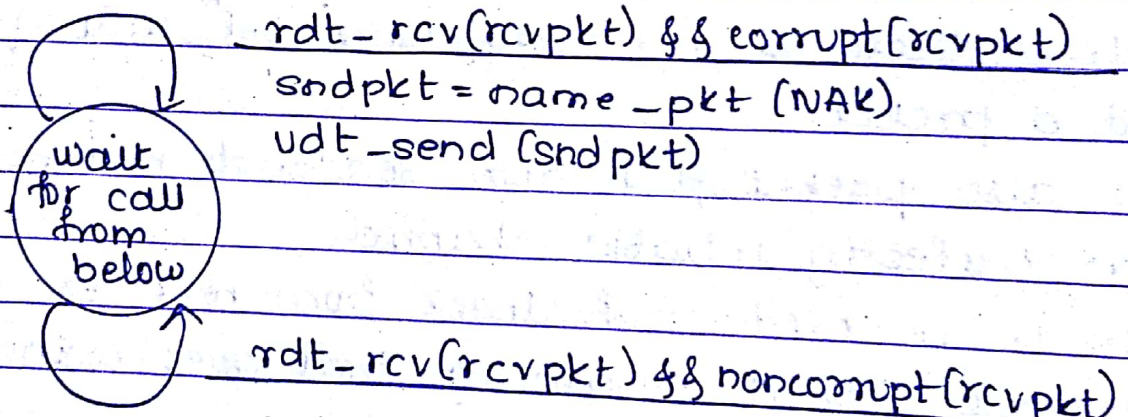
- This is a simplest protocol.
- Here there was no difference bet unit of data and a packet.
- and also packet flow from sender to reciever with perfectly reliable channel.
- There is no need of feedback from receiver to intimate about anything that goes wrong

Reliable Data Transfer over a Channel with Bit Errors rdt2.0



A

rdt2.0 Sending side



- Figure shows the FSM representation of rdt 2.0 a data transfer protocol employing error detection positive acknowledgments and negative acknowledgmen

- The sender side has 2 states
- In the leftmost state, the send side protocol is waiting for data to be passed from upper layes when rdt_send(data) event occurs the sender will create packets along with checksum and packet is sent via udt_send(sndpkt) operation.

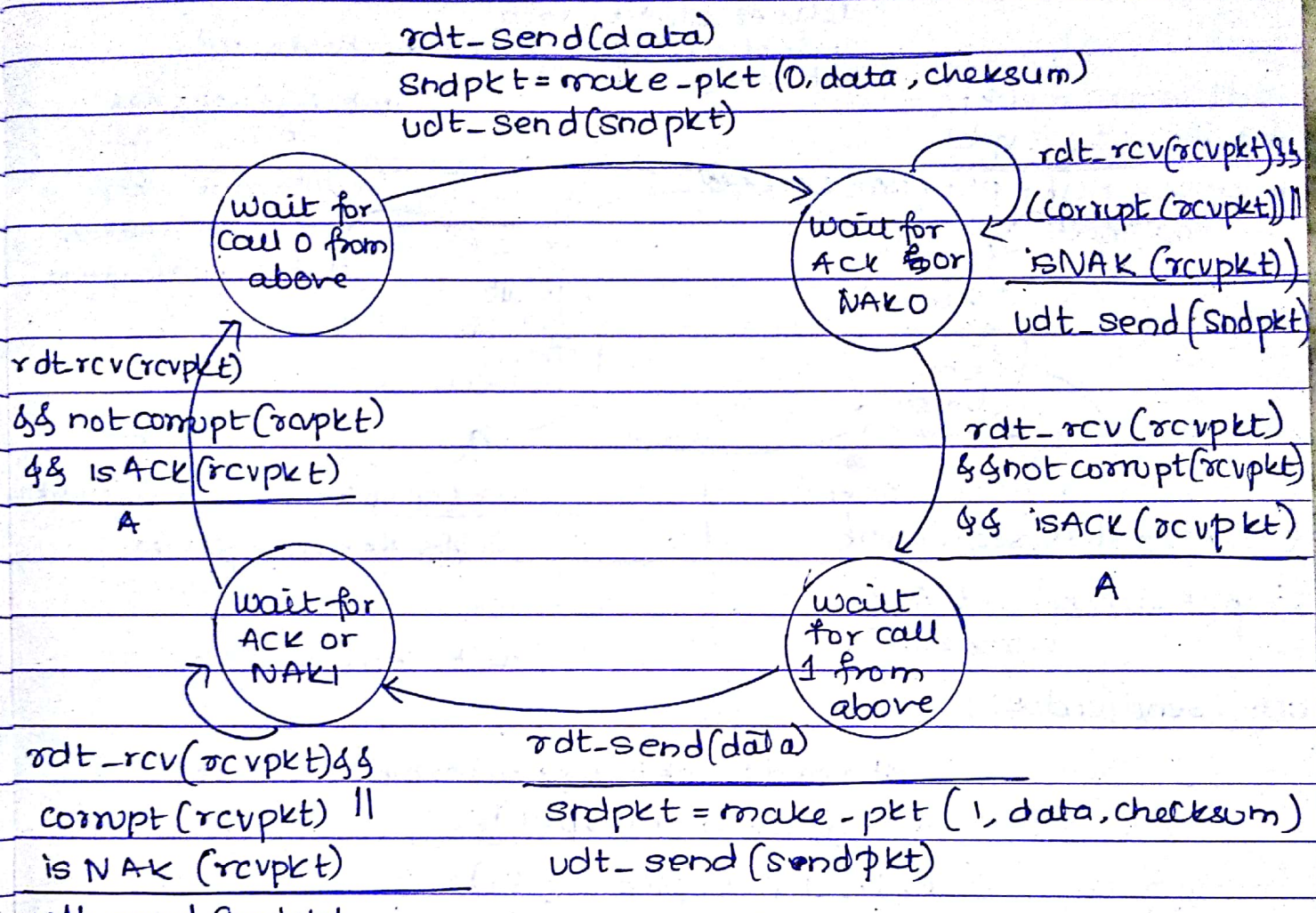
- In the right most side, the sender protocol is waiting for ACK or a NAK packet from the receiver if ACK packet is received [rdt_rcv(rcvpkt) && isACK(rcvpkt)] the sender knows that the transmitte packets is received ~~succes~~ correctly.
IF NAK is received the protocol retransmits the last packet and waits for ACK or NAK to be returned by the receiver for the retransmitted packet.

- When sender is in wait-for-ACK or NAK it cannot receive data from upper layer ie rdt_send() event cannot occur
- The sender will not send next piece of data until it is satisfied that the the recieved has recieved corre the current packect :: of this behavior rdt 2.0 is also known as "stop and wait" protocol

- The receiver side FSM of rdt 2.0 has a single state.
On packet arrival, the receiver replies with either an ACK or a NAK depending on whether the packet is corrupted or not.
- The notation rdt_rcv (rcvpkt) && corrupt (rcvpkt) corresponds to received a packet and found to have error in it

Unfortunately Protocol rdt 2.0 has a fatal flaw. It is possible that the ACK and NAK packet could get corrupted.

* rdt 2.1

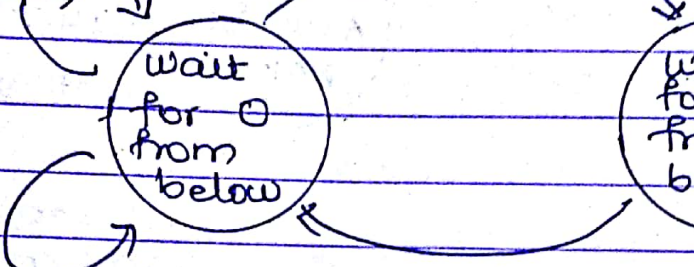


Sender. (rdt 2.1)

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)



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(rcvpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)

rdt2.1 & receiver.

The rdt2.1 has many states in the FSM because the protocol must reflect whether the packet currently being sent or expected should have a sequence number of 0 or 1

0 → packet is being sent or expected are mirror images of those

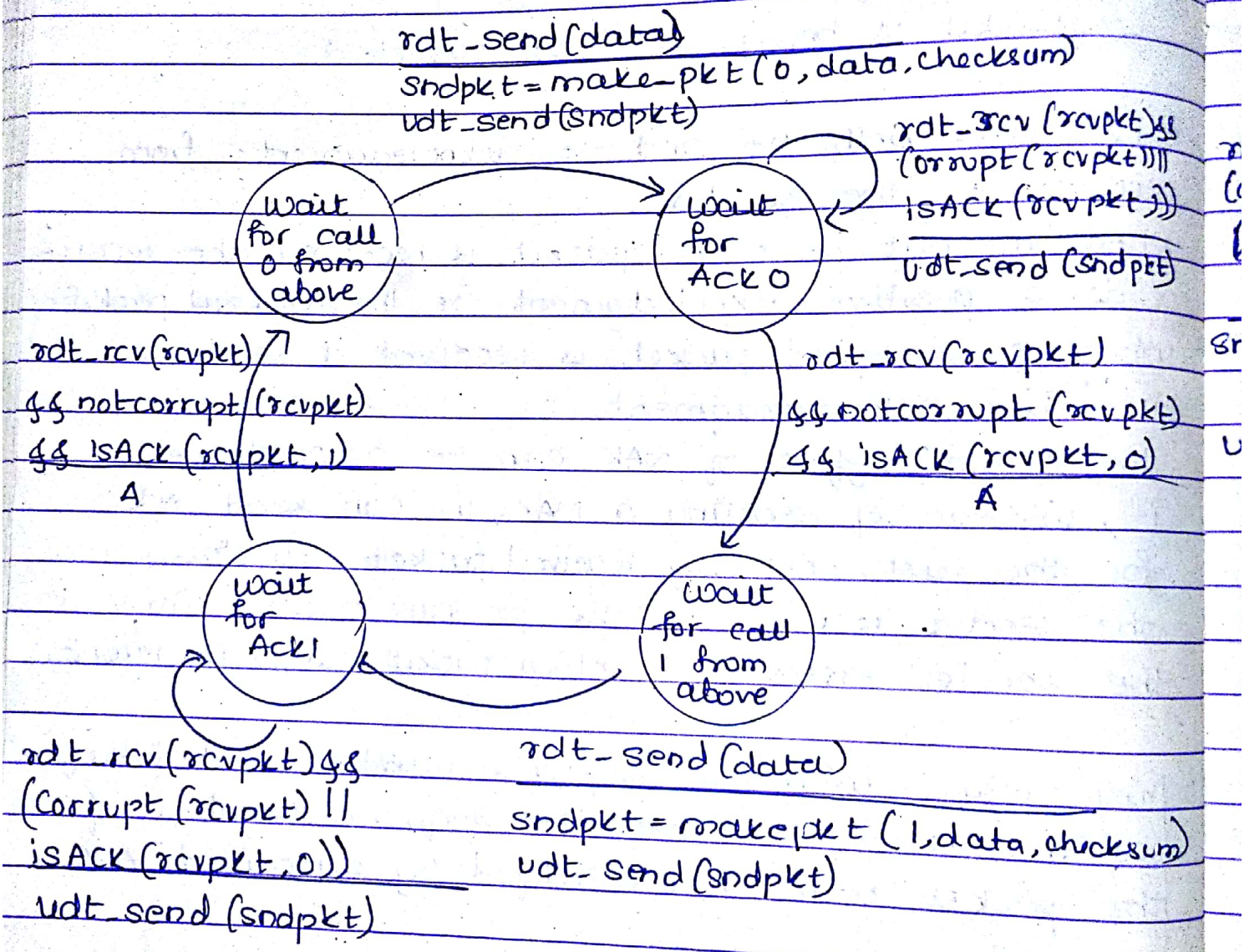
1 → packet is being sent or expected.

- rdt2.1 uses both +ve and -ve acknoledgments from receiver to the sender.
- When an out of order packet is received the receiver sends a positive acknoledgment for the received packet
- When a corrupted packet is received it sends a negative acknowledgment.
- The same effect of NAK can be accomplished if, instead of sending a NAK, we can send ACK for the last correctly received packet. By this the sender receives 2 Ack's for same packet Thus the sender knows which packet has to be resent.

- This protocol includes sequence number and therefore now the sender must check the sequence number of the packet being acknowledged by a received ACK message.

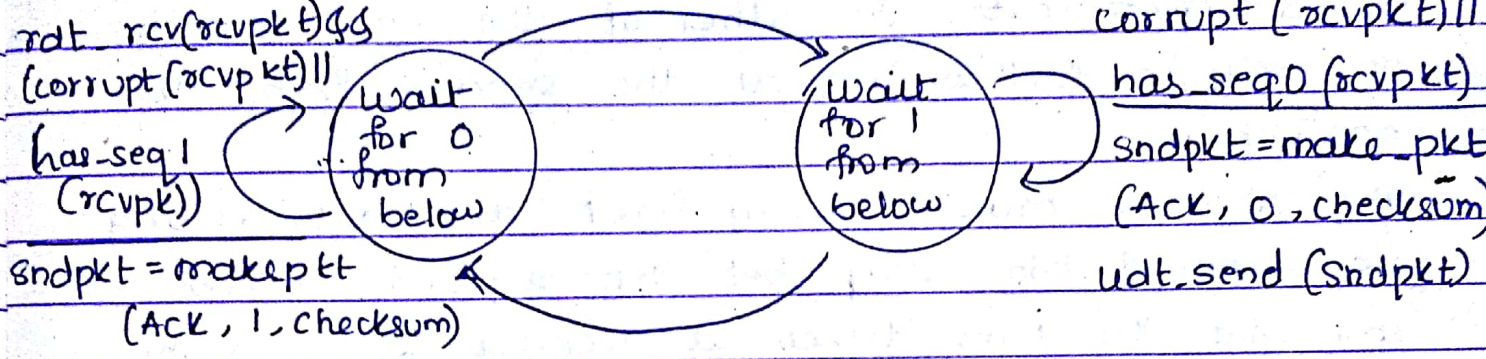
Reliable Data Transfers over a lossy Channel with Bit Errors rdt3.0

* rdt 2.2



rdt2.2 Sender.

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, 0, checksum)
udt_send(sndpkt)



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, 1, checksum)
udt_send(sndpkt)

rdt 2.2 receiver

- One subtle change between rtd2.1 and rtd2.2 is that the receiver must now include sequence number of the packet being acknowledged by a an ACK msg and the sender must now check the sequence number of the packet being acknowledged by a received ACK message in the sender FSM.