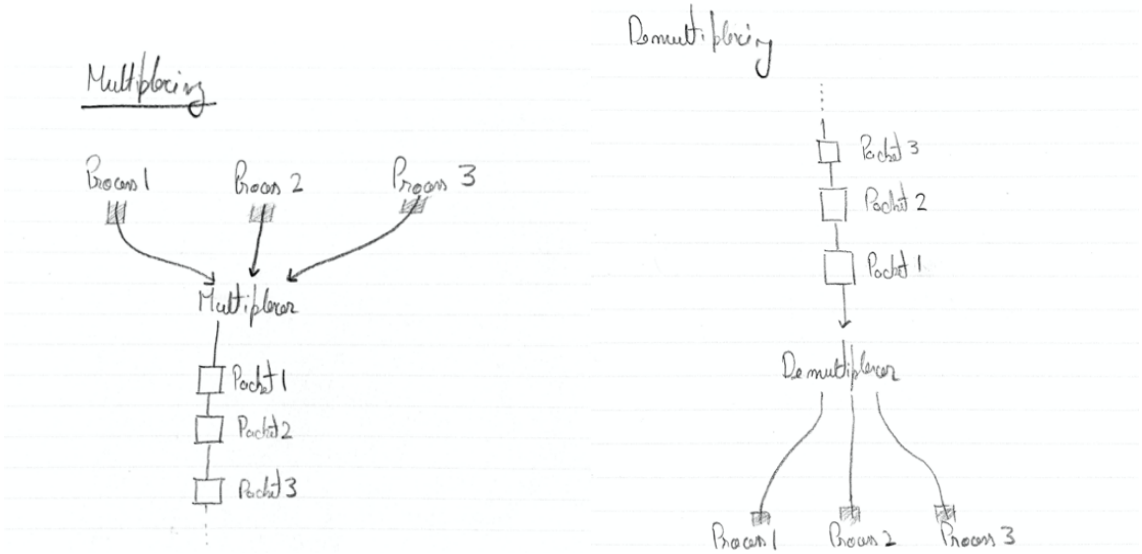
Ans1:

a)

Transport Layer	Network Layer
• Transport Layer, in a nutshell, provides communication protocols for two processes running on different hosts to communicate with each other. • Transport Layer is responsible for flow control, *i.e.* tasks like checking if the data received is error-free, processing and packaging data for delivery. • Transport Layer is where data delivery protocols work, as an example, this is where TCP, UDP work.	• While the Network layer provides communication between two or more hosts. • Network Layer is responsible for logical and physical addressing, *i.e.* Translating logical addresses to physical addresses and vice versa. • Network Layer is where addressing happens, *i.e.* IPv4 and IPv6 addresses are managed here. Other protocols like ICMP, RARP, ARP also work on the network layer

b)

Now we know the job of the transport layer is to provide communications for processes running on hosts, which means it gathers data from the processes socket and converts it into packets with their designated headers to be passed to the network layer. This process is known as multiplexing



Now, the reverse of the process mentioned above is known as demultiplexing, *i.e.* Talking packets from the network layer and then analyzing their headers and port numbers and then forwarding them to their designated sockets.

Ans2:

a) Given that the network layer will guarantee the delivery of the segment for us we only have to design a protocol that will be responsible for multiplexing, demultiplexing. The protocol has 4 bytes assigned for the header information.

While sending some data this header will store the port number of the destination process and the destination host address which will be provided by the application layer. Now that we're using 4 bytes for the header, our protocol can handle 1196 bytes of data at one time.

While receiving our protocol will examine the 4 bytes of header data in a received message to find the port number out. The protocol will then extract the data from the received message and forward the data to the recognized port number.

b) To add a return address to the destination process we can add another field to our header, *i.e.* Source port number. Using this we can identify on what port the application is using to send the data and can be used as a "return address" to the destination process.

Ans3:

a) A -> S
A = 467
S = 23

b) B -> S
B = 513
S = 23

c) S -> A
S = 23
A = 467

d) S -> B
S = 23
B = 513

e) Since A and B are different hosts, it is possible for the source port number in the segments from A to S to be the same as that from B to S because both of them will have different TCP addresses.

f) No, it's not possible. If A and B are the same host, that means two different sockets on the same host are communicating with S, two sockets on the same host cannot share the same port.

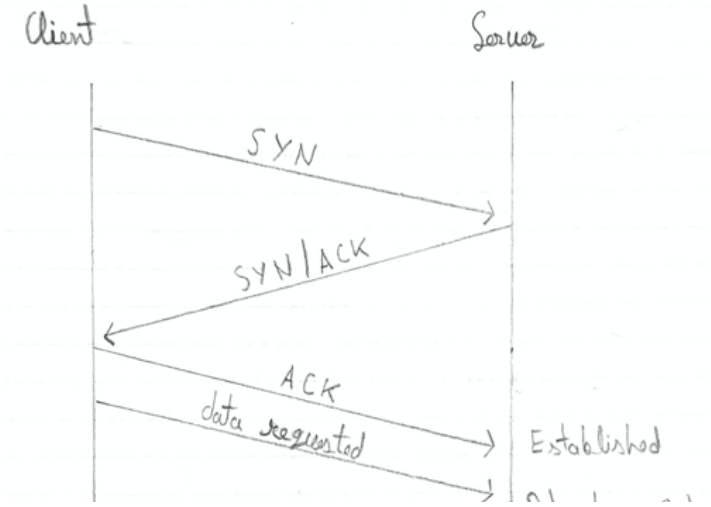
Ans4:

a) There are multiple reasons why it would be preferred to use TCP over UDP for video and audio streaming traffic. Part of the first reason is that often a lot of firewalls are set to block any UDP packet that isn't a DNS packet. This is done for security reasons, given that it's not easy to trace back UDP packets and that it's a stateless protocol, UDP can be used for DDoS and other similar attacks.
Another reason has got to do with the monitory side of things, services like Netflix do not want people snooping on their streams, potentially recording and decrypting the streams only to feed to the piracy frenzy. To deal with the mentioned and similar practices, streaming services implement DRMs, which can potentially stop or reduce such attempts at stealing data. UDP being a stateless nonpersistent protocol, it's really difficult to implement a consistent and secure DRM, which is why services like Netflix and Hulu choose to use persistent TCP connections for video delivery.

b) Yes, an application can enjoy reliable data transfer even when an application is running over UDP. The developer always has the choice to develop a custom solution on the application layer which could work over UDP but provides reliable data transfer. They might choose to use RUDP(Reliable UDP) as well, a protocol that builds on UPD, provides the benefits of using UDP and provide a basic reliability layer as well.

Ans5:

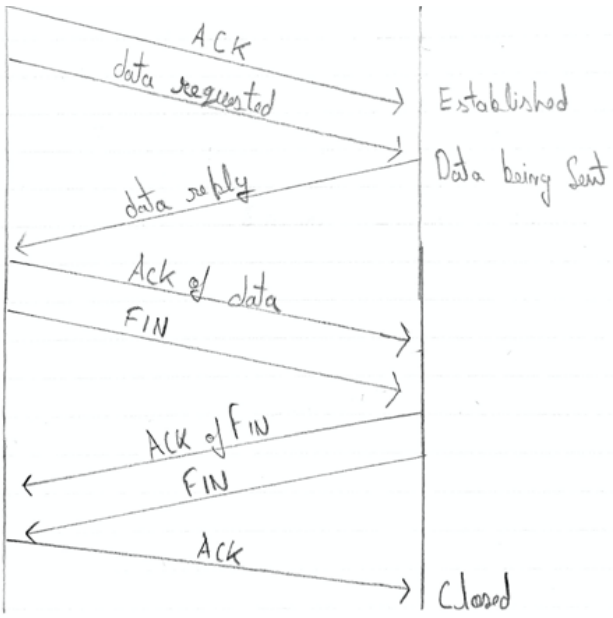
a) The TCP connection is established using a method known as 'three-way handshake'[1]. The Three-way handshake involves sending three messages to a host and back to negotiate to start a TCP connection between the two devices.



In a scenario where a client has to request a file from a server; The Client sends the SYN, a synchronization message to the server. An SYN message is a signal that a certain host is trying to form a TCP connection. After receiving the message if the server wants to form the connection it sends back an SYN/ACK message, which is essentially an acknowledgment message stating that the server has received the SYN and is ready to form a connection. After the client has received the SYN/ACK message, it replies with an ACK message, an acknowledgment to the SYN/ACK. A TCP connection is formed when the

server receives the ACK message from the client. After this, the client can go on to request the data.

b) Here we can see the missing part of the diagram shown above. Which shows how the connection is closed.

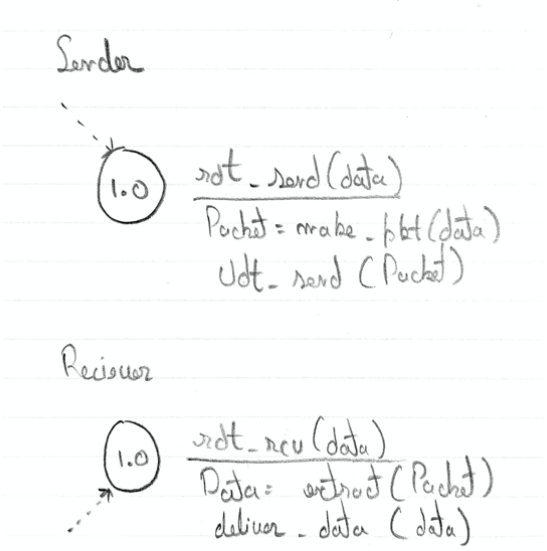


After establishing the connection, the client can request data from the server, the server then replies which the data requested. If the client receives the requested data correctly it replies with an acknowledgment, stating it has received the data. And if the client doesn't require more data it immediately sends a FIN(an abbreviation for final) message, signaling that it wants to close this connection. After receiving the FIN message, the server replies with an acknowledgment of the FIN message is received and a FIN message from itself to the client. After receiving the FIN, the client sends an acknowledgment to the server and the connection is closed.

Ans6:

Rdt 1.0

RDT 1.0 is the simplest of all the versions without implementations of concepts like dealing with bit errors. 1.0 simply accepts the data from the upper layers using rdt_send(data) and receives data from lower layers using rdt_rcv(data).

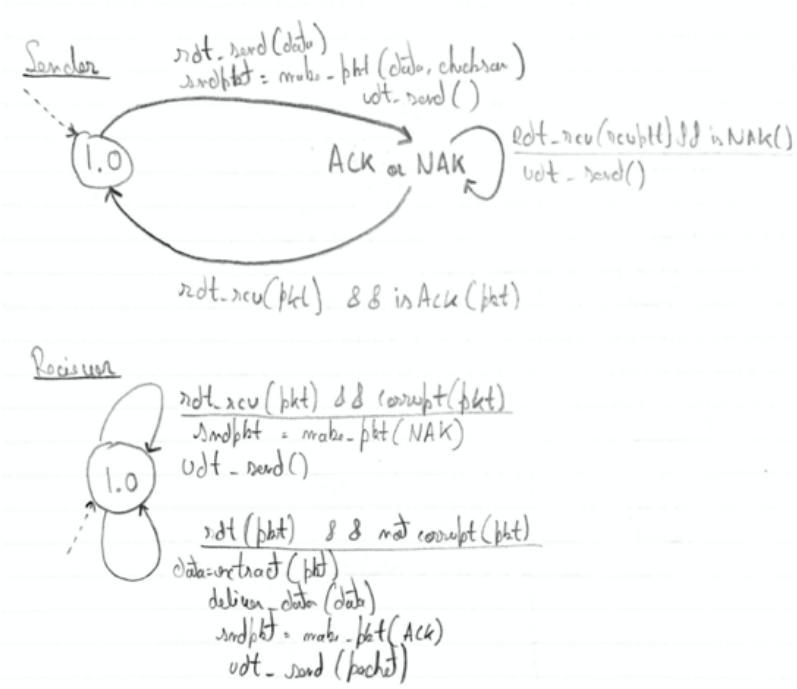


While sending, when rdt_send is called, the protocol makes packets using packet = make_packet(data) and then sends those packets using udt_send(packet).

And while receiving the data is received by rdt_rcv(data), it's extracted using extract(data) and is delivered to the process using deliver_data(data).

Rdt2.0

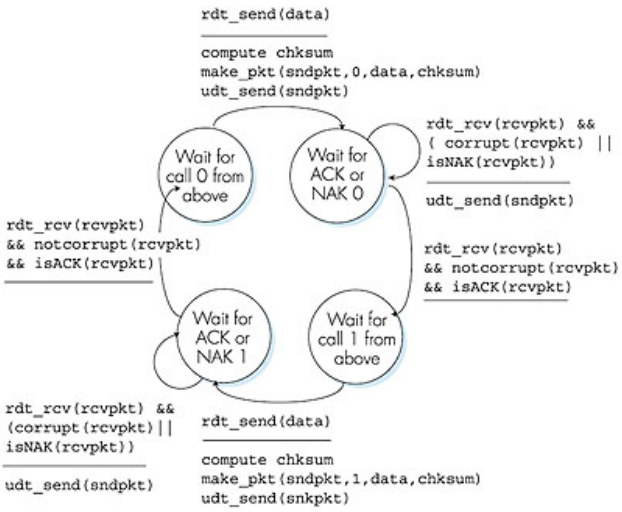
RDT 2.0 builds on the 1.0 by adding 3 essential features; Error detection, receiver feedback, retransmission.



Similar to the RDT 1.0 rdt_send is called and packets are generated using make_pkt but this time a checksum is included in the arguments which will be sent with the packet. The packets are now sent using udt_send. Now the sender will check for an acknowledgment, it will receive a packet using rdt_rcv() and check if it's a NAK or a ACK using isNAK() and isACK(). If it's an isNAK() the senders resend the packet. On the receiver's side data is received and is checked if it's corrupted or not using corrupt(pkt) and notCorrupt(pkt). If it's not corrupted, the data is it's extracted and delivered to the process like done previously and an ACK packet is generated using make_pkt(ACK) and sent. If the data is corrupted a NAK packet is generated using, make_pkt(NAK) and sent.

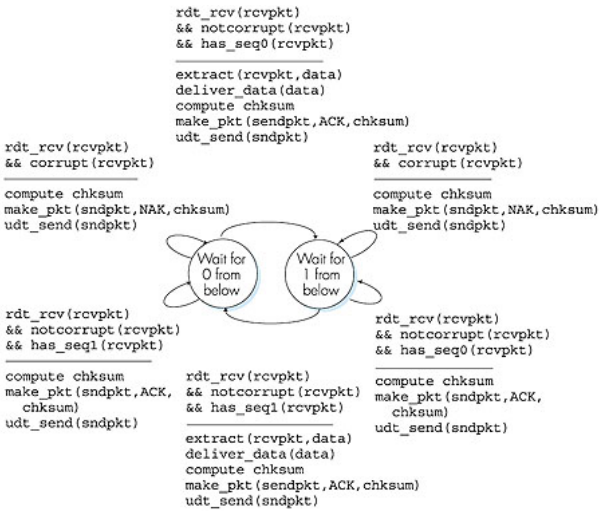
Rdt2.1

RDT 2.1 builds on the flaws 2.0 has. 2.0 would often result in receiving multiple packets because of corrupted acknowledgments packets. 2.1 attempts to solve this by adding a packet counter to the packet header. The make_pkt() function would have 3 arguments now: make_pkt(packet number, data, checksum). And the sender wouldn't resend the packets until it received a NAK with the packet



2.1 Sender Source : temple.edu
https://astro.temple.edu/~stafford/cis320f05/lecture/chap3/deluxe-content_files/03-11.jpeg

number attached to it. For example while sending a bunch of packets, the packet 4 gets corrupted, the receiver can simply send back a NAK 4 after which the sender will resend the packet 4 again, previously there was a chance that by the time the sender receives the NAK he'd have sent the packet 5 hence thinking the packet 5 got corrupted and resend packet 5. 2.1 works on solving situations like mentioned.

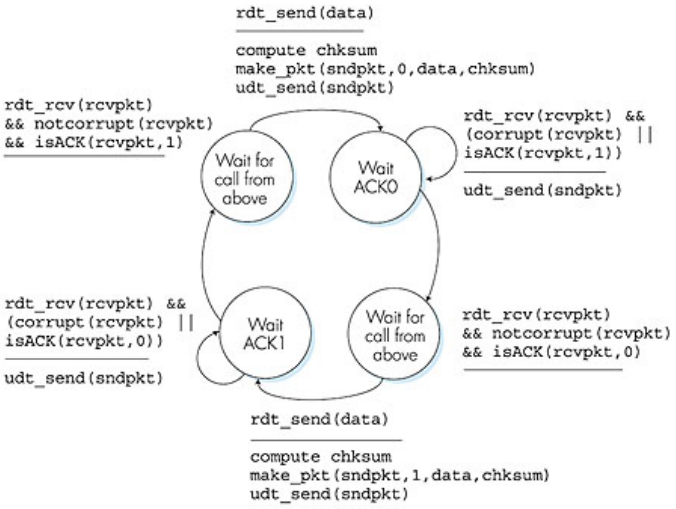


2.1 Receiver Source : temple.edu

https://astro.temple.edu/~stafford/cis320f05/lecture/chap3/deluxe-content_files/03-12.jpeg

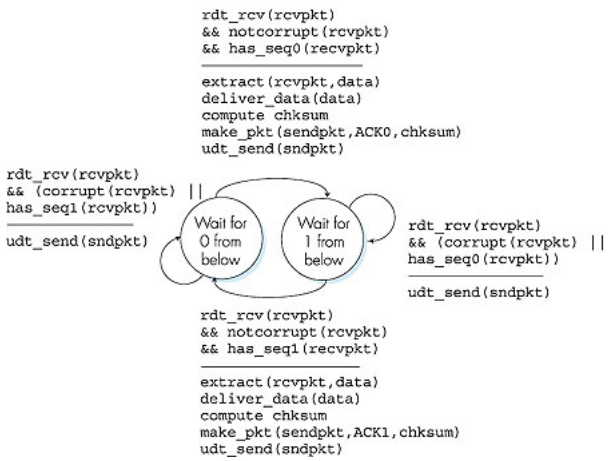
Rdt2.2

RDT 2.2 is a NAK free protocol, in 2.1 it was still le ACK for the same package, as the ACK wasn't numbe received if a NAK got corrupted while reaching :kage was the sender. 2.2 mitigates this by getting rid of NAKs and implementing ACKs which would be numbered according to their packets. Now before sending the next packet 1 after it just sent the 0 the sender would wait for the ACK 0 to be received. The sender would now check the sequence number on the previously received acknowledgment and send the packet next in sequence.



2.2 Sender Source : temple.edu

https://astro.temple.edu/~stafford/cis320f05/lecture/chap3/deluxe-content_files/03-13.jpeg

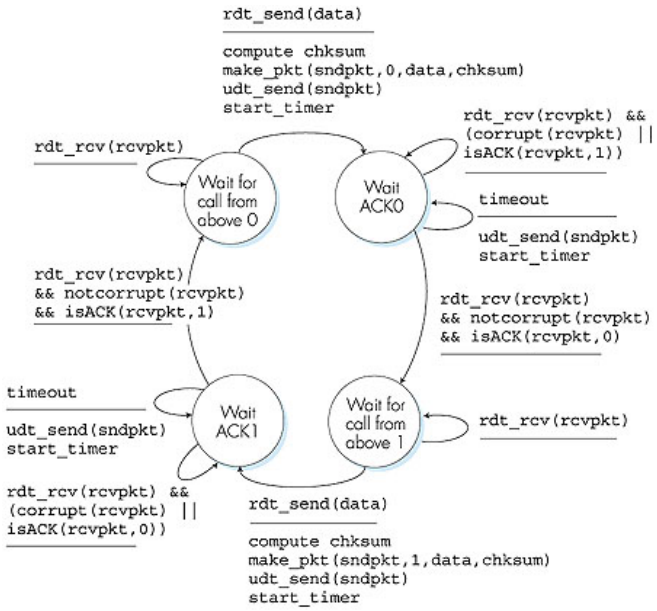


2.2 Receiver Source : temple.edu

https://astro.temple.edu/~stafford/cis320f05/lecture/chap3/deluxe-content_files/03-14.jpeg

Rdt3.0

In RDT 2.2, the sender still does not know if a packet was lost or is delayed. In all cases the action is the same, that is to resend the packet. This is where RDT 3.0 comes in, the previously mentioned versions looked for ACKs or NAKs to continue sending the next packet but in cases of packet delay or loss, that doesn't work at all. RDT3.0 introduces wait time on the sender's end to deal with the cases of loss. While waiting to receive the ACK for a particular sequence number, if a set timeout occurs, the sender will resend the packet just to avoid the possibility of packet loss. [2]



3.0 Sender Source : temple.edu

https://astro.temple.edu/~stafford/cis320f05/lecture/chap3/deluxe-content_files/03-15.jpeg

References

[1] Inetdaemon, May 2018. TCP 3-way handshake.

[2] James Kurose, Keith Rose. A Top-Down Approach, pg: 241 – 255.