Part-1 (Open Book)

<*Max. Marks: 15*> <*Weight 15%*>

1. Consider a class of applications that warrants use of a *reliable Application Layer protocol* that should be able to provide desired application services atop an unreliable transport layer protocol. *It is desired that such a protocol be able to have the following features:*
    a. *'In-order' 'Message' / 'Application Protocol Data Unit (APDU)' delivery to the invoking application,*
    b. *Messages / APDUs larger in size (in term of number of octets) than the payload of a single 'Segment' (TPDU) which could be transported by the underlying transport level protocol,*
    c. *Lossless Message / APDU delivery,*
    d. *Ability to handle varying degrees of network delays caused due to short-term network traffic overload / error conditions,*

    Conceptualize and design a simple protocol (let's call it '*R-ALP*') briefly describing required *steps* such that at end of the process, you could suggest:
    - a *Message / APDU format* along *with due reasoning,*
    - a *Timing-and-Sequence Diagram* (or a corresponding *State Transition Diagram)* depicting the protocol behavior.

    Please also *state and justify* whether the resultant protocol '*R-ALP*' is correct and reasonably robust.

Msg. APDU Format | S.T.D. OR Timing & Seq. Dia. | Reasoning

4 + 4 + 4

4x3+3=15

Comment on Correctness + Robustness

3

CSF303

A-1

17/03/16

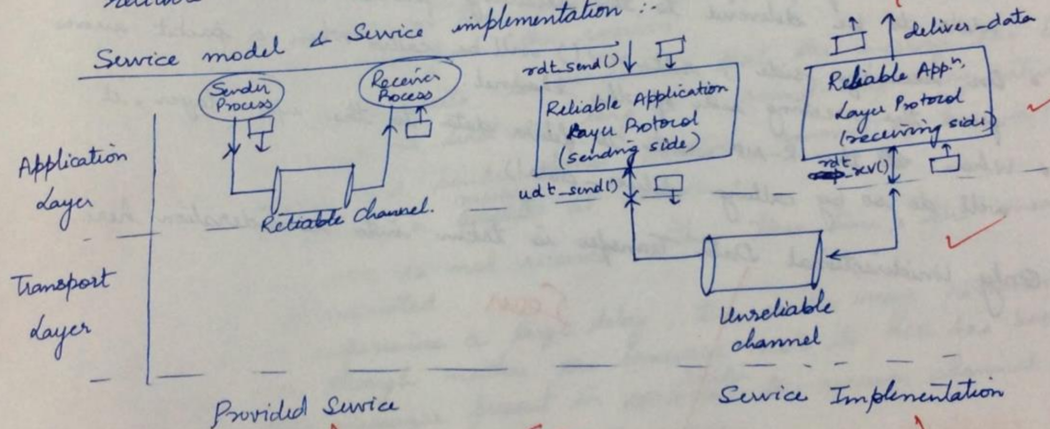
COMPUTER NETWORKS

PART-1

Q1. Reliable Application Layer Protocol (R-ALP)

- An Application Layer protocol defines how an application's processes, running on different end systems, pass messages to each other.
- So, we needs to create a protocol to make this message passage reliable.

Service model & Service implementation :-



Seen

- The service abstraction provided to application layer is that of reliable channel through which data can be transferred.
- With a reliable channel, no transferred data bit are corrupted or lost and all are delivered in the order in which they ~~came~~ were sent.
- We view the lower layers simply as an ~~a~~ unreliable point-to-point channel.

In the figure:-

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

(*) Only Unidirectional Data Transfer is taken into consideration here.

Seen

We will evolve the protocol, step-by-step :-

R-ALP over a Channel with Bit Errors and Lossy Transmission :-

- Assuming underlying channel is one in which bits in the message may get corrupted.
- We need to deal with what to do when a packet loss occurs and how to detect it.
  - We put the burden of detecting? and recovering from lost packets on the sender.
  - Suppose, the sender transmits a data packet / message ~~the~~ and either that message or the receiver's ACK gets lost, no reply is forthcoming at the sender from the receiver.
  - The sender waits to be certain that the message has been lost. It waits at least as long as a round-trip delay between the receiver and sender plus whatever amount of time is needed to process a ~~packet~~ message at receiver.
  - If an ACK is not received within this time, the message is retransmitted.
- If a packet experiences a large delay, the sender may retransmit the message even though neither the message nor its ACK has been lost i.e. duplicate ~~data~~ messages present in ~~receiver~~ sender-to-receiver channel. It will be handled next along with bit errors.

Not the best way, this!

What about use of checksumming, multiple buffers, timer etc.?

How to handle pipelining in this case?

• Here, we will ~~take~~ use Automatic Repeat reQuest (ARQ) protocol when the message-dictation process uses both positive ACKs & negative ACKs to allow the receiver to let the sender know what has been received correctly and what has been received in error.

– In addition to ARQ, 3 additional protocols are required to handle bit errors:

• Error detection : to allow the receiver to detect when bit errors have occured.

• Receiver feedback: in the form of ACK & NAK.

• Retransmission :– Retransmission of ~~one~~ message received in error at receiver.

• There may be a possibility that ACK ~~or~~ NAK message could be corrupt. To handle this and to handle duplicate messages, a new field is added to the data message, by putting a sequence number.

You can't use both!

Which one would you want and why?

• We require a countdown timer for retransmission.

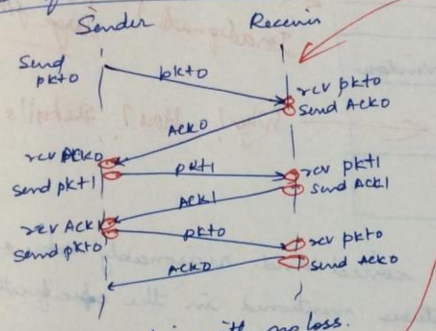
Seen

- So, ~~as~~ the stated protocol uses both NAK and ACK from the ~~se~~ receiver to sender & When an out-of-order message ~~packet~~ is received, the receiver sends a ACK for the message it has received. When a corrupted message is received, the receiver sends a NAK.

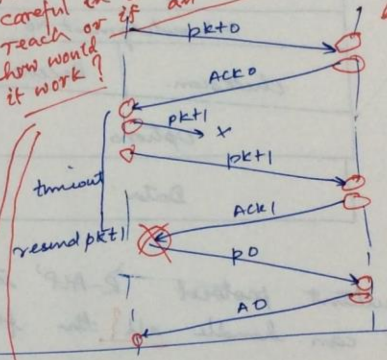
ACK & NAK BOTH?

Timing and Sequence diagram for the protocol:-

You must be more careful in correct depictions. If an APDU does NOT reach or if an ACK never originates at Rx end, how would it work?

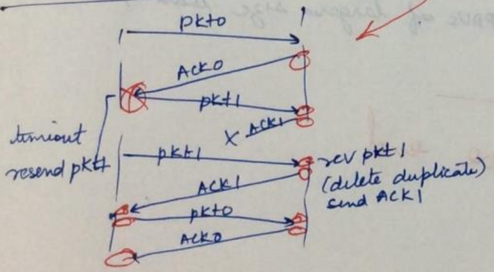


a) Operation with no loss.



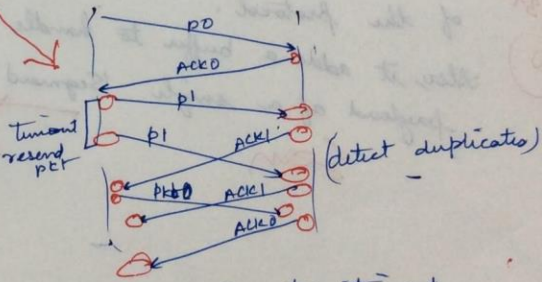
b) Lost packet

You've assumed one-way ACKs (only from Rx to Tx). WHY?



c) Lost ACK.

Seen



d) Premature timeout

• Message format :-

Source Port # ~~Message~~	Destination Port #
Sequence No.	
Acknowledgment No.	
Checksum.	Receive Window
Options	
Data.	

Inadequate !

How to handle large files?

Why? How? Details?

The resultant protocol 'R-ALP' is correct and reasonably robust as it can handle all ? the features mentioned in the specifications of the protocol.

Also, it adds a buffer to handle APDUs of larger size than the payload of a single 'Segment'.

3+2+3+1 = 10

Sen

the end

B-2

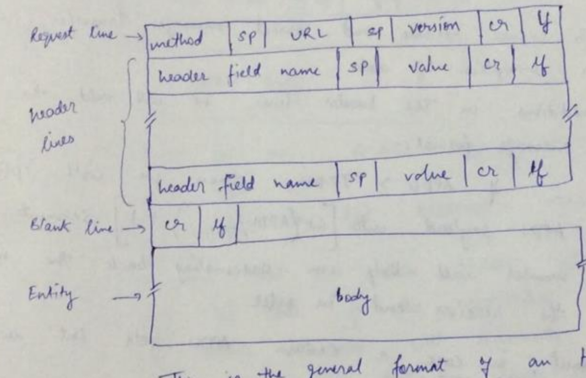
Then why have ANY Transport Layer at all?

1. → In this hypothetical R-ALP protocol, we are basically inculcating all the properties of a typical ~~TCP~~ Transport Layer protocol (say, TCP) into the Application Layer itself; as we have studied.

→ We know that Application Layer communicates through "processes". Each process is forwarded down to Transport Layer (and further below) through Sockets. In order to include the reliability and other features as mentioned in the problem statement into this layer, we will have to modify the header of each such process.

→ I will explain here with the help of an HTTP message as a starting case.

Irrelevant here!



cr: carriage return (\r)
lf: line feed (\n)
sp: space

This is the general format of an HTTP request message as we have studied

→ We will modify the header lines as follows:-

Seen

→ We will modify the header lines by adding the following processes and fields within it :-

(a) • Sequence Number of process :- for in-order delivery of messages

(c) • Acknowledgement Number :- To keep track of messages received correctly on the other side and without any losses.

(d) • Receiver Window Size :- To keep track of available buffer size on the receiver size and throttle the sending procedure accordingly to prevent overloading / congestion.

(e) • Checksum :- To ensure reliable and correct message transfer without losses or corruption of data.

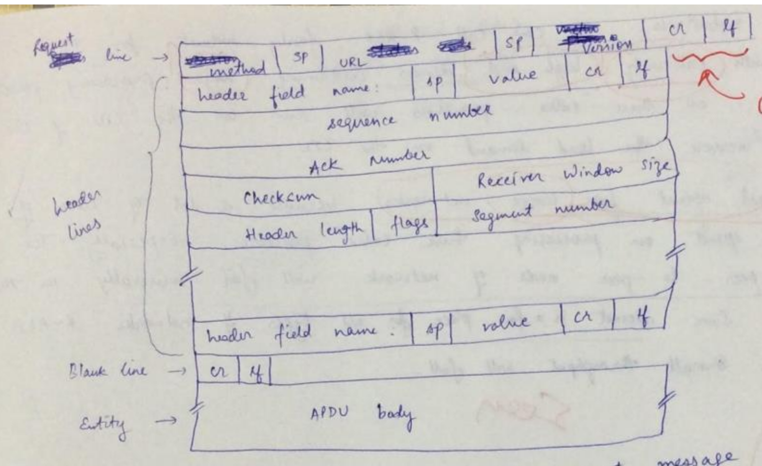
→ Apart from these four additions in the header lines, we will add the following processes within the message format :-

(b) • APDU Segment Number :- If APDU > TPDU, then we will split our APDU payload into $\left[\text{int}\left(APDU/_{TPDU}\right) + 1\right]$ segments. The segment number will help in assembling back the message at the receiver end in order.

() • Timer :- To retransmit, in case a certain APDU gets lost during transmission.

The new message format will look like :-

Seen



Outside 32/64/128/256-bit boundary? why?

Bit positions missing! Fragmentation handling?

A typical format for an R-ALP request message

Similar changes can be made in the response message

→ It is important to note that all these header changes will come with corresponding changes in the code of the application that is running R-ALP. All these checking and extra processes will eventually run on CPU of the hosts (both server and client) as it is an Application Layer protocol.

→ This protocol is correct in the sense that all the conditions required in the problem statement have been inculcated at the Application Layer in a 'reliable' format. The source and destination addresses will be provided by the unreliable Transport Layer protocol ; along with port numbers.

Seen

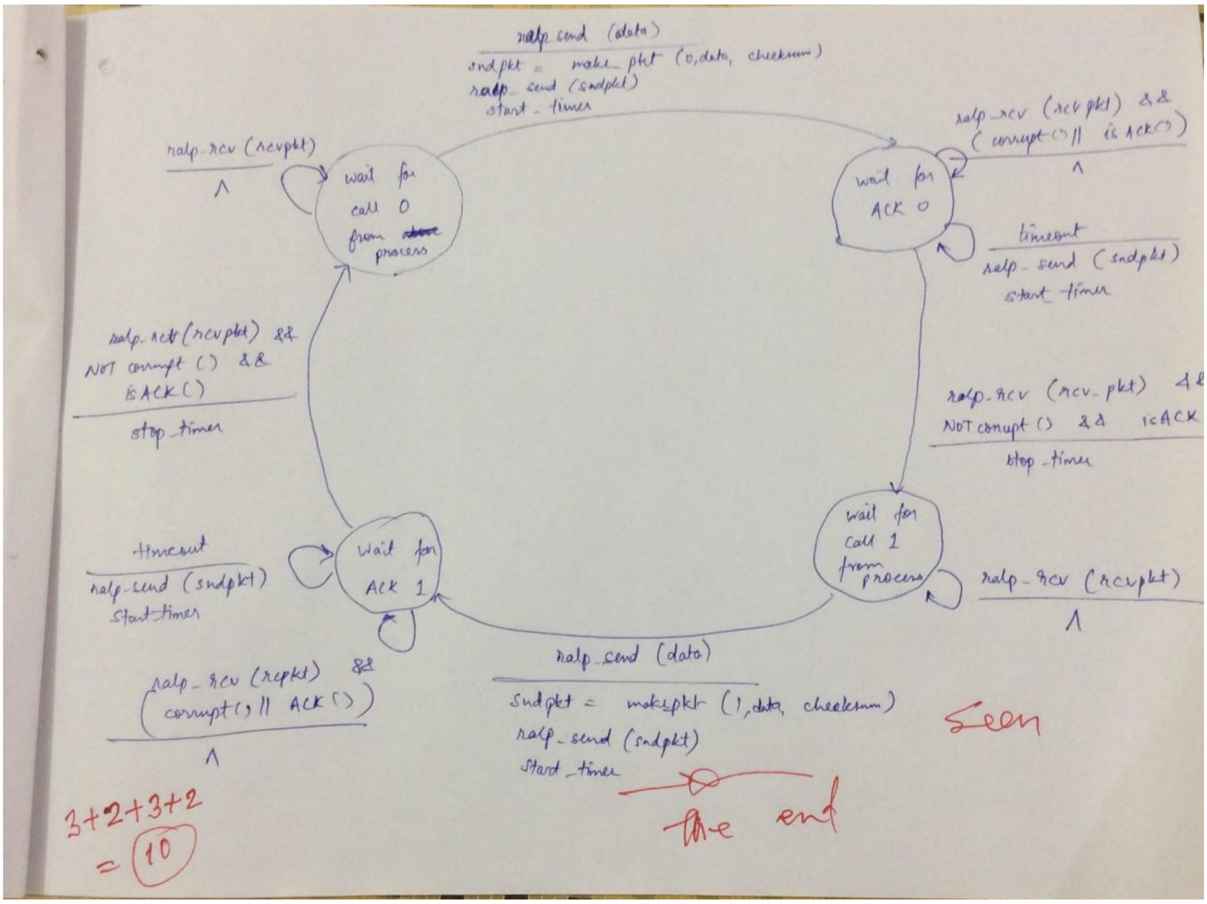
→ To justify the robustness, we can call R-ALP fairly robust for a small network with extremely ? high end ? devices containing huge ? processing power.

The reason being:- all these "extra processes will run on the CPU of the hosts. This will increase the load demand on the CPU.

The protocol is not robust for large networks because a lot of time of the hosts will be spent on processing these "extra" processes. Especially the server. Also, peer-to-peer mode of network will fail miserably in this format of R-ALP. Since internet is a fair place for all types of networks, R-ALP is not recommended. Overall throughput will fall.

Is it NOT a Wrong reason! If hosts are busy, does network size matter?

Seen



rdt_send (data)
sndpkt = make_pkt (0, data, checksum)
rdt_send (sndpkt)
start_timer
rdt_rcv (rcvpkt)
Λ
wait for call 0 from process
wait for ACK 0
rdt_rcv (rcvpkt) &&
(corrupt() || is ACK())
Λ
timeout
rdt_send (sndpkt)
start_timer
rdt_rcv (rcvpkt) &&
NOT corrupt () &&
isACK ()
stop_timer
rdt_rcv (rcv_pkt) &&
NOT corrupt () && isACK
stop_timer
wait for call 1 from process
Wait for ACK 1
timeout
rdt_send (sndpkt)
Start_timer
rdt_rcv (rcvpkt)
Λ
rdt_send (data)
sndpkt = make_pkt (1, data, checksum)
rdt_send (sndpkt)
Start_timer
rdt_rcv (rcvpkt) &&
(corrupt() || ACK())
Λ
Seen
the end
3+2+3+2
= 10

C-2

I Since transport layer protocol is unreliable, Application Layer protocol should be reliable.

To ~~do~~ have 'Inorder' message → receiver side needs to maintain only in-order bytes & discard the out-of-order bytes. But such a design would require a lot of retransmission. Eg → if 1,2 4 ~~3~~5 are received on the receiver side, & if in-order bytes are to be maintained then 4,5 ~~needs~~ will be discarded even though they are received correctly. ~~3~~ 4,5 will be retransmitted in this case. Better design would be that out-of order bytes to be keept & waiting for missing bytes to fill in the gaps. Clearly, this choice is more efficient in terms of network bandwidth.

~~Also~~ so ~~to~~ we can have a hybrid ~~typ~~ design in which there is cummulative acknowledgement (only bytes upto first missing byte are acknowledged) & out-of order segments to be kept & not discarded.

Seen

Check-sum should be there so that bit-errors can be detected.

Since network traffic / error condition needs to be handled, congestion control & ~~control~~ flow control mechanism should be there.

To ~~Send~~ avoid overflowing at receiver's side, sender should maintain the variable called receive window which will give the sender an idea of much free buffer space available at receiver.

In Congestion Control, we can have three states, slow start, congestion avoidance & fast recovery

Slow start → Initially we do not know how much traffic is there in the network, so we should have size of current window to be equal to 1. Then as we move along, after ~~1st~~ transmitted segment is acknowledged, we can increase it by 1. ∴, value of current will double every Round Trip time.

Seen

Check-sum should be there so that bit-errors can be detected.

Since network traffic / error condition needs to be handled, congestion control & ~~control~~ flow control mechanism should be there.

To ~~Send~~ avoid overflowing at receiver's side, sender should maintain the variable called receive window which will give the sender an idea of much free buffer space available at receiver.

In Congestion Control, we can have three states, slow start, congestion avoidance & fast recovery

Slow start → Initially we do not know how much traffic is there in the network, so we should have size of current window to be equal to 1. Then as we move along, after ~~1st~~ transmitted segment is acknowledged, we can increase it by 1. ∴, value of current will double every Round Trip time.

Seen

Why bring in all TCP features?

But doubling the size of current window(cwnd) every RTT, is not a good idea, so we can set a threshold beyond which won't double the size rather increase the size of current window by 1 every RTT. This will be called congestion avoidance.

Why at AL?

~~If we lose~~
~~We can encounter~~
It is possible that we ~~want~~ get duplicate acknowledgment from receiver if the transmitted segment get lost. In such a case after three duplicate acknowledgment we can retransmit the missing segment?. This is fast recovery & we can use it as it will save our time rather than waiting for ~~a~~ timeout to occur.

You are designing at AL (not at TL)!

After retransmitting missing segment, we can enter into fast recovery ~~which~~ where ~~will decrease~~ the ~~window~~ threshold will decrease size by 2. & current window will also be decreased & it remain in fast recovery state ~~as~~ until there is a timeout or new Acknowledgmt is received.

Seen

APDU should contain the followin

(1) Payload (Data to be ~~to~~ carried)

(2) 32-bit Sequence number & 32-bit acknowledgement number as they help in implementing a reliable data & service.

(3) Receive window → for flow control as explained ~~above~~ earlier

(4) ~~ACK~~ ~~Sou~~ Source & Destination port numbers for multiplexing / demultiplexing from / to other layer.

(5) ACK ~~field~~ bit → To indicate value carried in acknowledgement field is valid.

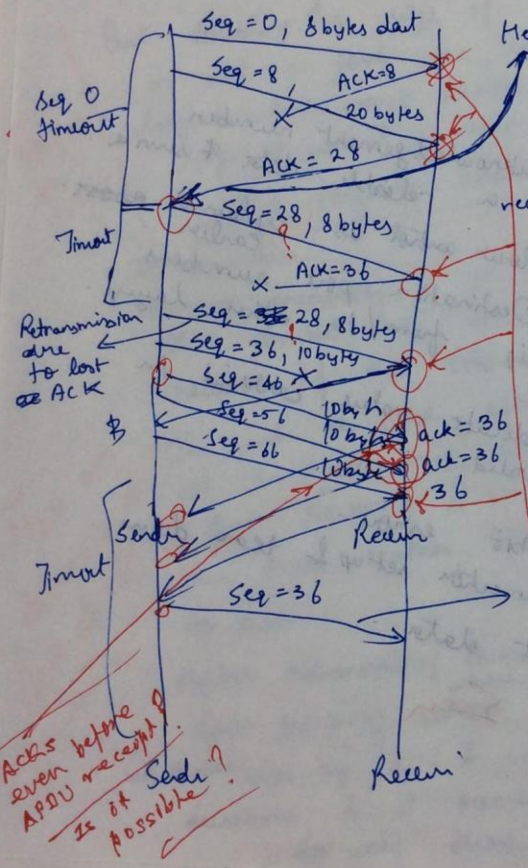
(6) Checksum → to check for bit errors.

(7) SYN & FIN bit for connection set up & tear down.

(8) URG bit to mark urgent data.

Seen

Why took so many ~~mechanisms~~ of TCP mechanisms here?



Here cummulate ACK avoid retransmission of first ~~segment~~ segment.

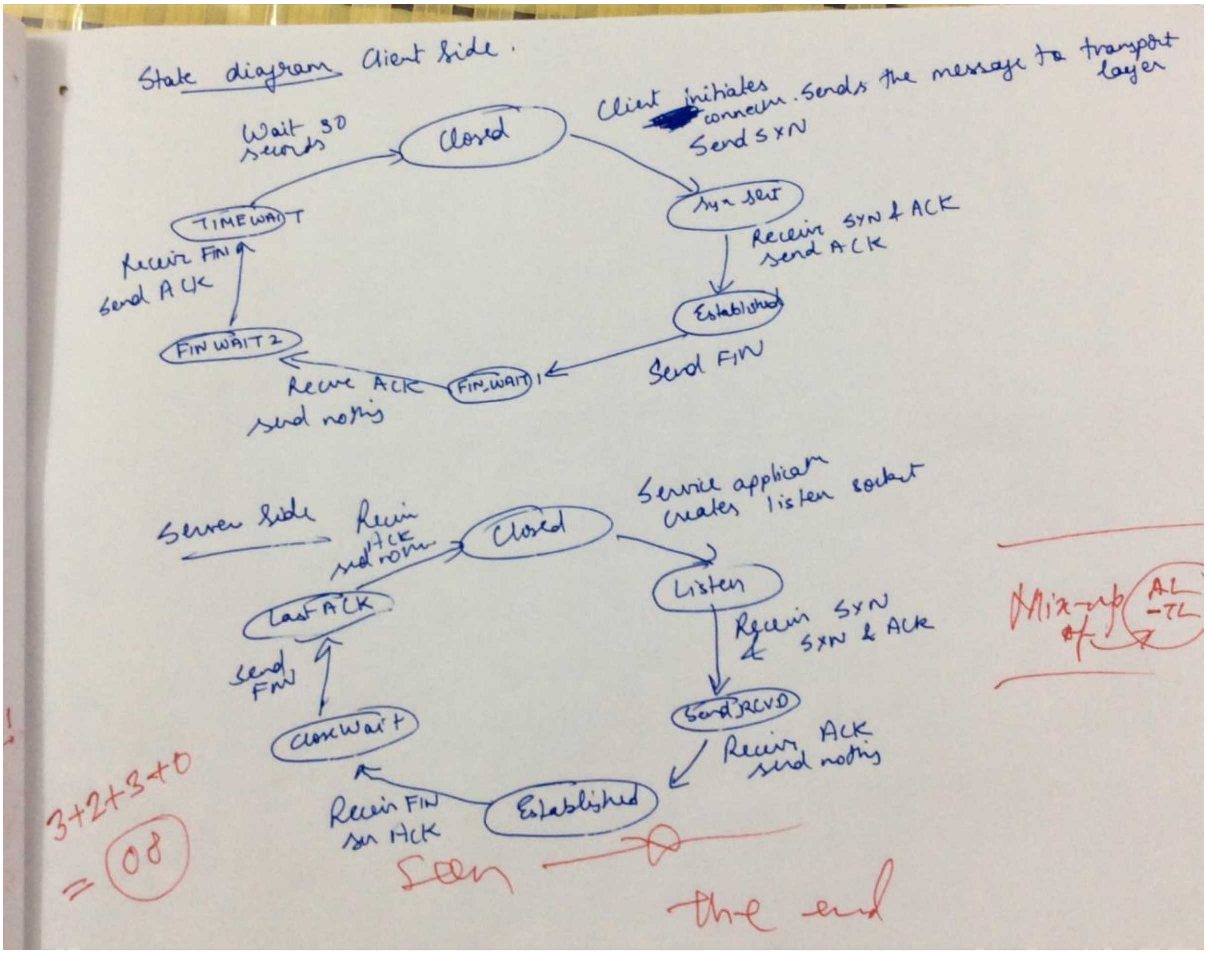
Here acknowledgement number that ~~sender~~ receiver puts is sequence number of next byte it is expecting from sender.

Timer is used to timeout/retransmit.

Sequence number store the byte stream number of first byte in the segment.

Retransmit missing segment before timer

You have shown the impossible here! Zero queuing, processing and transmission delays! WHY?


State diagram Client side.
Client initiates connection. Sends the message to transport layer
Send SYN
Wait 30 seconds
Closed
Syn sent
Receive SYN & ACK
send ACK
Established
Send FIN
FIN_WAIT 1
Receive ACK
send nothing
FIN WAIT 2
Receive FIN
Send ACK
TIMEWAIT
Server side
Receive ACK
send none
Closed
Service application creates listen socket
Listen
Receive SYN
& SYN & ACK
Send RCVD
Receive ACK
send nothing
Established
Receive FIN
send ACK
CloseWait
Send FIN
Last ACK
Mix-up of AL-TL
3+2+3+0
= 08
Seen
the end