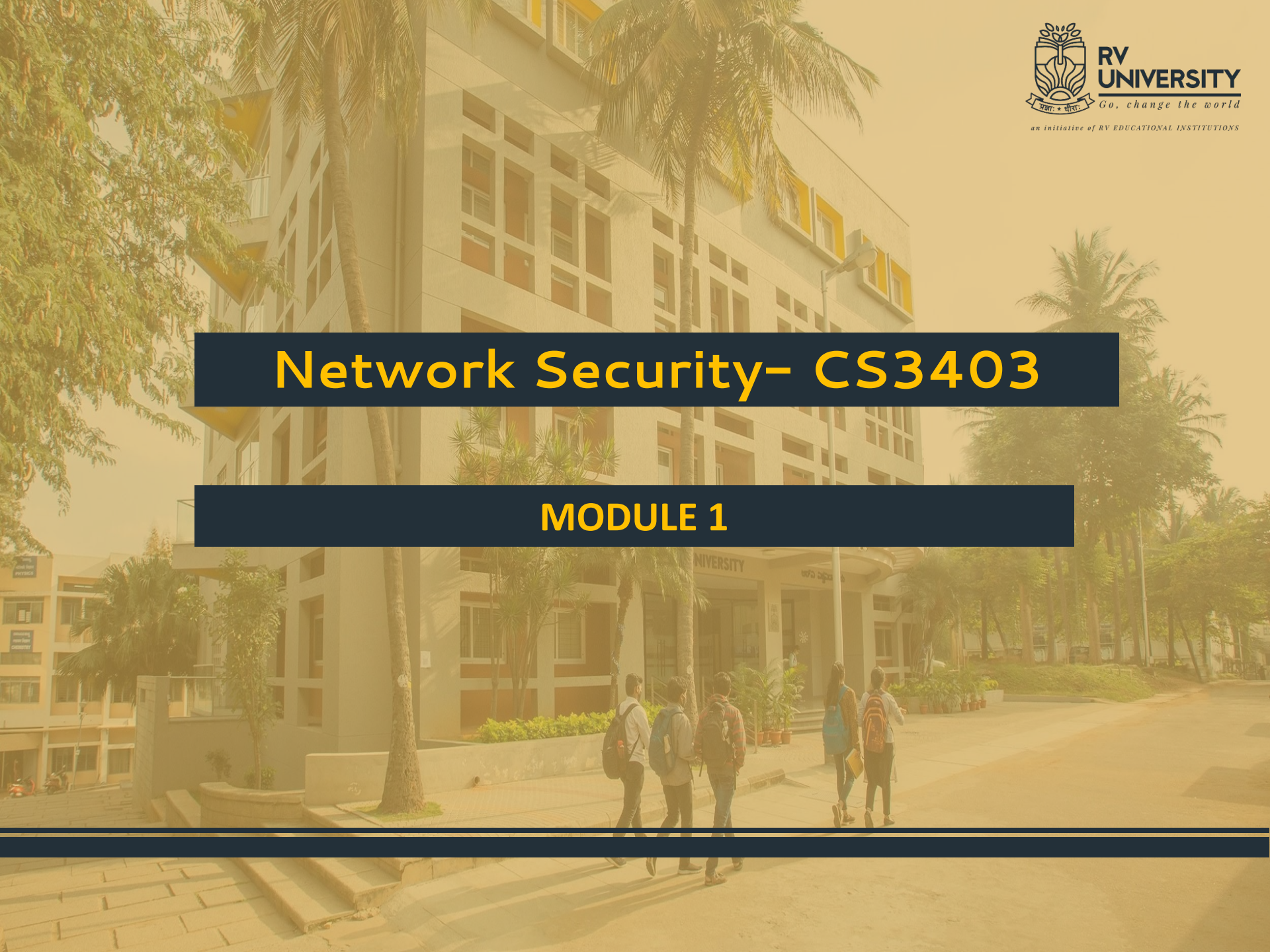


RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Network Security- CS3403

MODULE 1

Network Security- CS3403

MODULE 1

Network Security

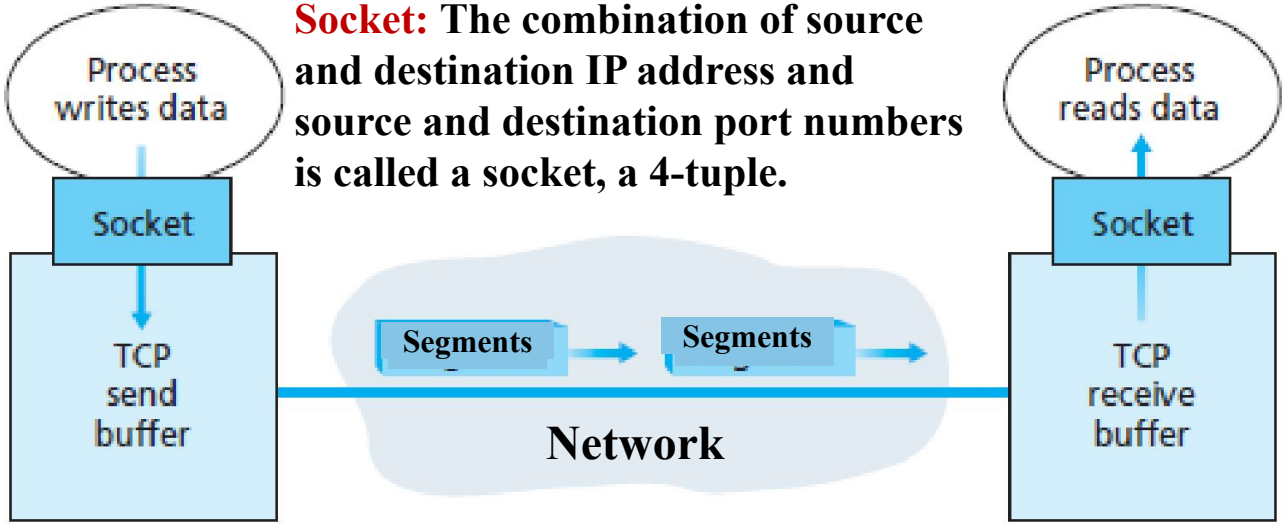
Introduction to TCP and Sockets

TCP
(Connection-Oriented)

TCP: Transmission Control Protocol

Layer 4

TCP Sockets



Socket: The combination of source and destination IP address and source and destination port numbers is called a socket, a 4-tuple.

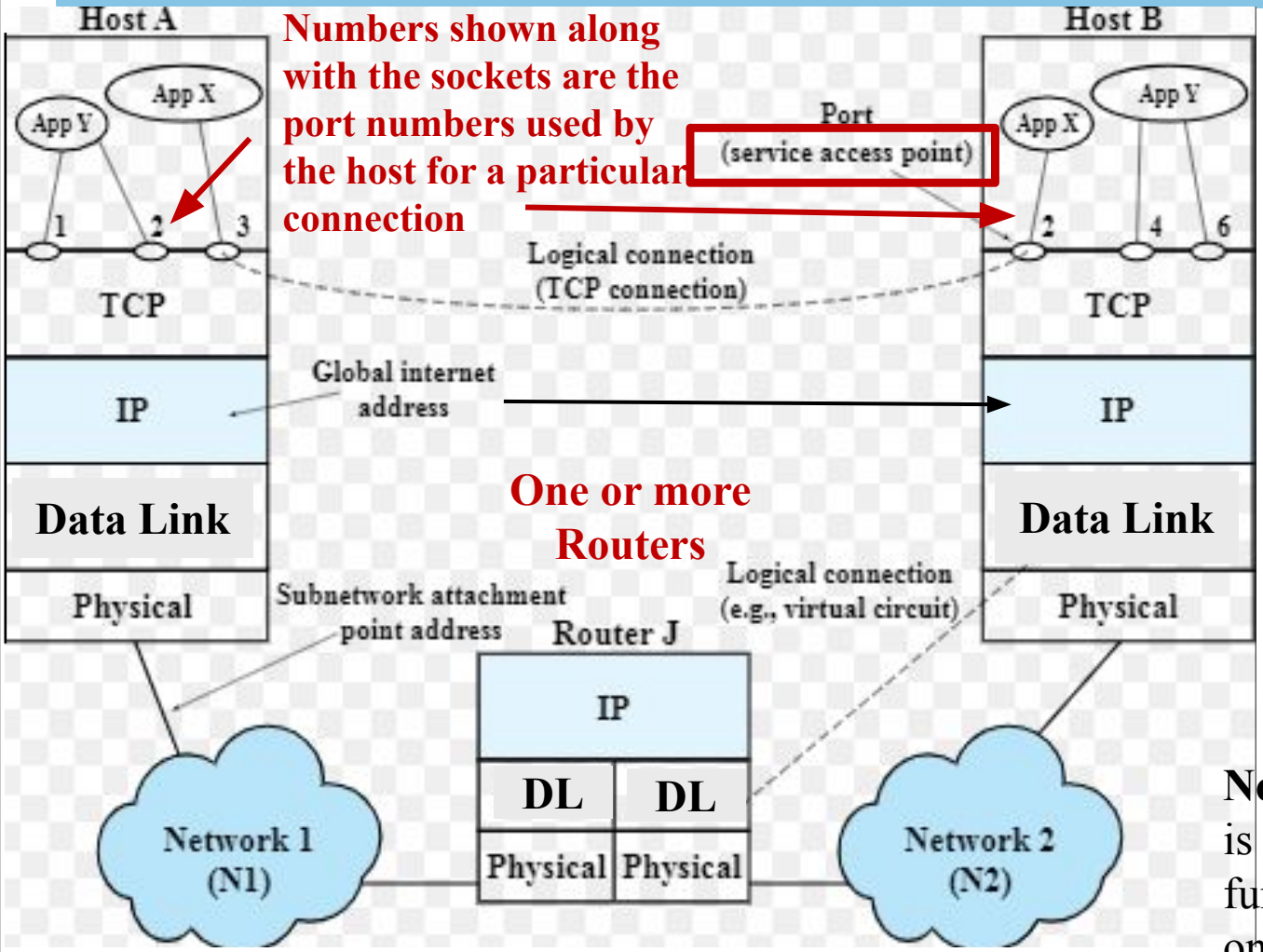
TCP connections are between two: **c**

a. Hosts
b. Subnets
c. Processes running on Hosts
d. None

Port numbers are some 16 bits values

- TCP is said to be **connection-oriented** because before one application process can begin to send data to another, the **two processes** must first "**handshake**" with each other
  - That is, they must send some **preliminary segments** to each other to establish the parameters for ensuing reliable data transfer, including the size of data in each segment
- As part of **TCP connection establishment**, both sides of the connection will initialize many TCP state variables

Multiple TCP connections on Host



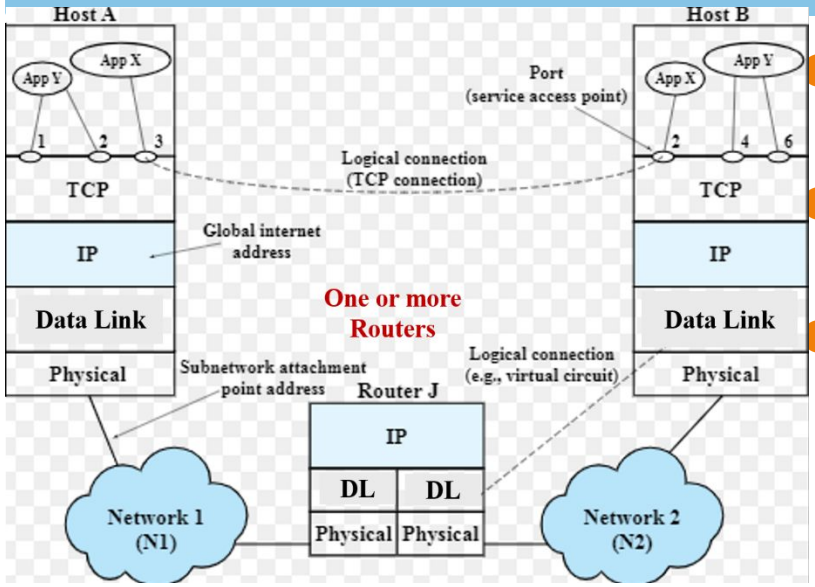
Note1: While a TCP connection between Host A and Host B already active, both hosts can establish multiple new TCP connections with many other hosts as well.

Note2: Remember that the TCP/IP protocol stack running on the hosts maintains the state of each active TCP connections that the hosts are currently having.

Note3: The TCP/IP stack is responsible for the functioning of all connections on a host.

SAP: **Service Access Point** is an identifying label for network endpoints used in Open System Interconnect (**OSI**) networking.

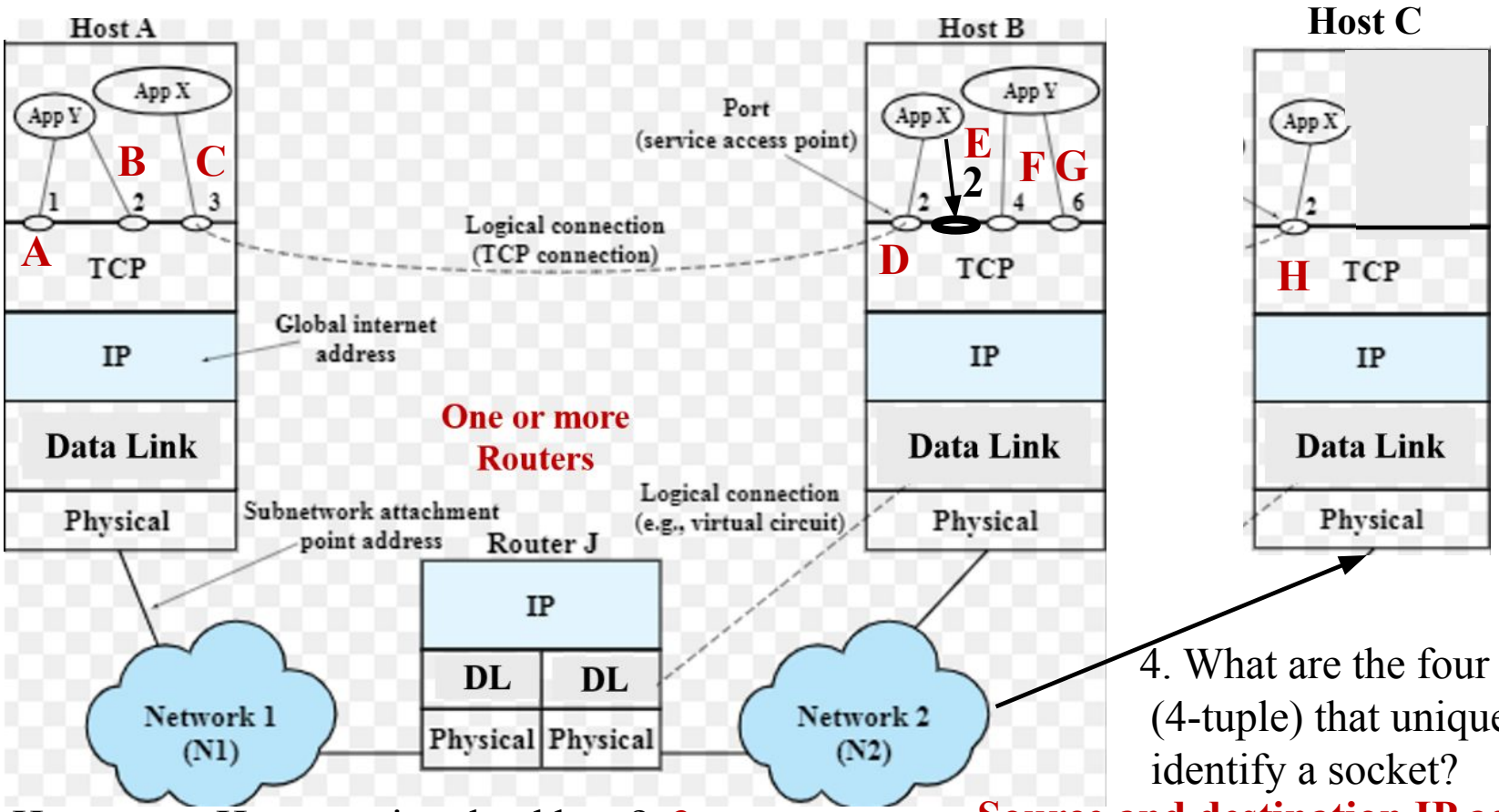
TCP connections: Explained



- Each host can have multiple TCP connections simultaneously
- The connections can be between the same set of hosts or different hosts
- TCP connections are **duplex**
  - i.e., the data transfer can happen in both directions, in parallel, independent of each other

- There are **exactly two end points** (hosts) communicating with each other on a TCP connection.
  - Broadcasting and multicasting aren't applicable to TCP
- The TCP datagram is called a segment, which includes TCP header + data
- Each TCP segment contains the **source** and **destination port number** to identify the sending and receiving applications
- These above two values along with the **source** and **destination IP addresses** in the IP header, **uniquely** identifies each **connection**

How does a Socket Identify a Unique Connection?



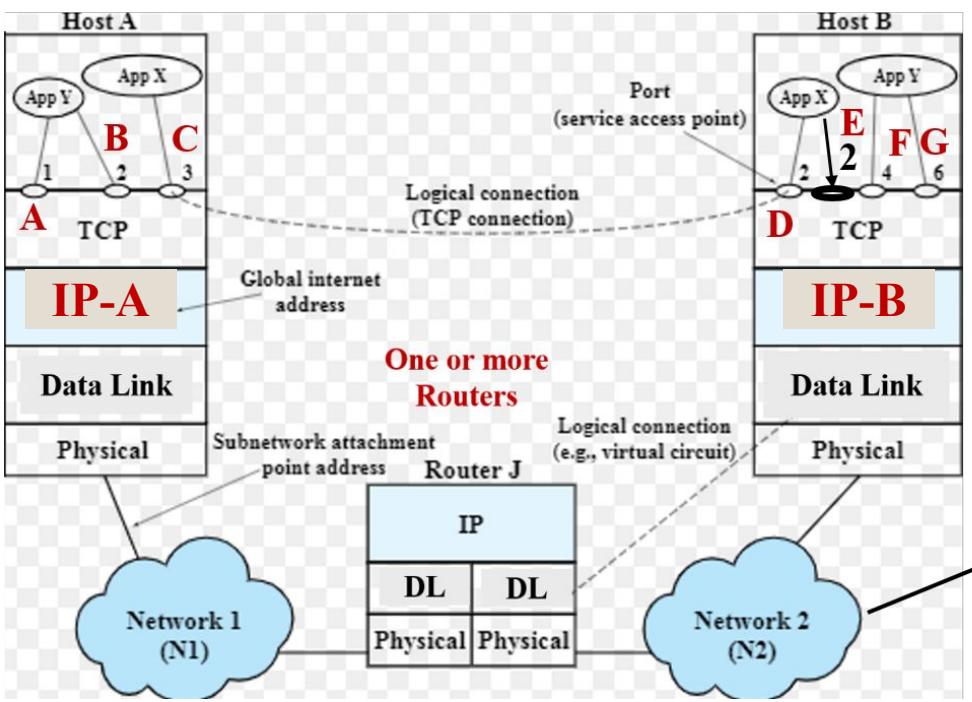
1. How many Hosts are involved here? **3**
2. How many sockets are shown here? **8**
3. Let us name the sockets from **A to H**

4. What are the four values (4-tuple) that uniquely identify a socket?

Source and destination IP addresses, source and destination Port numbers

Check if these sockets can be uniquely identified?

What are the values of the Sockets here?



Socket is a 4-tuple:

Source IP address, destination IP addresses, source Port Number and destination Port number

As you are aware, each host will have an unique IP address each

Let them be: IP-A, IP-B, IP-C

Let the **socket pairs** which are **forming a TCP connection** are: **(A-F), (B-G), (C, D), (E-H)**

Now, give the 4-tuple values of each Socket: (SrcIP, DestIP, SrcPort, DestPort)

On Host A

Socket A: (IP-A, IP-B, 1, 4)
Socket B: (IP-A, IP-B, 2, 6)
Socket C: (IP-A, IP-B, 3, 2)

On Host B

Socket D: (IP-B, IP-A, 2, 3)
Socket E: (IP-B, IP-C, 2, 2)
Socket F: (IP-B, IP-A, 4, 1)
Socket G: (IP-B, IP-A, 6, 2)

On Host C

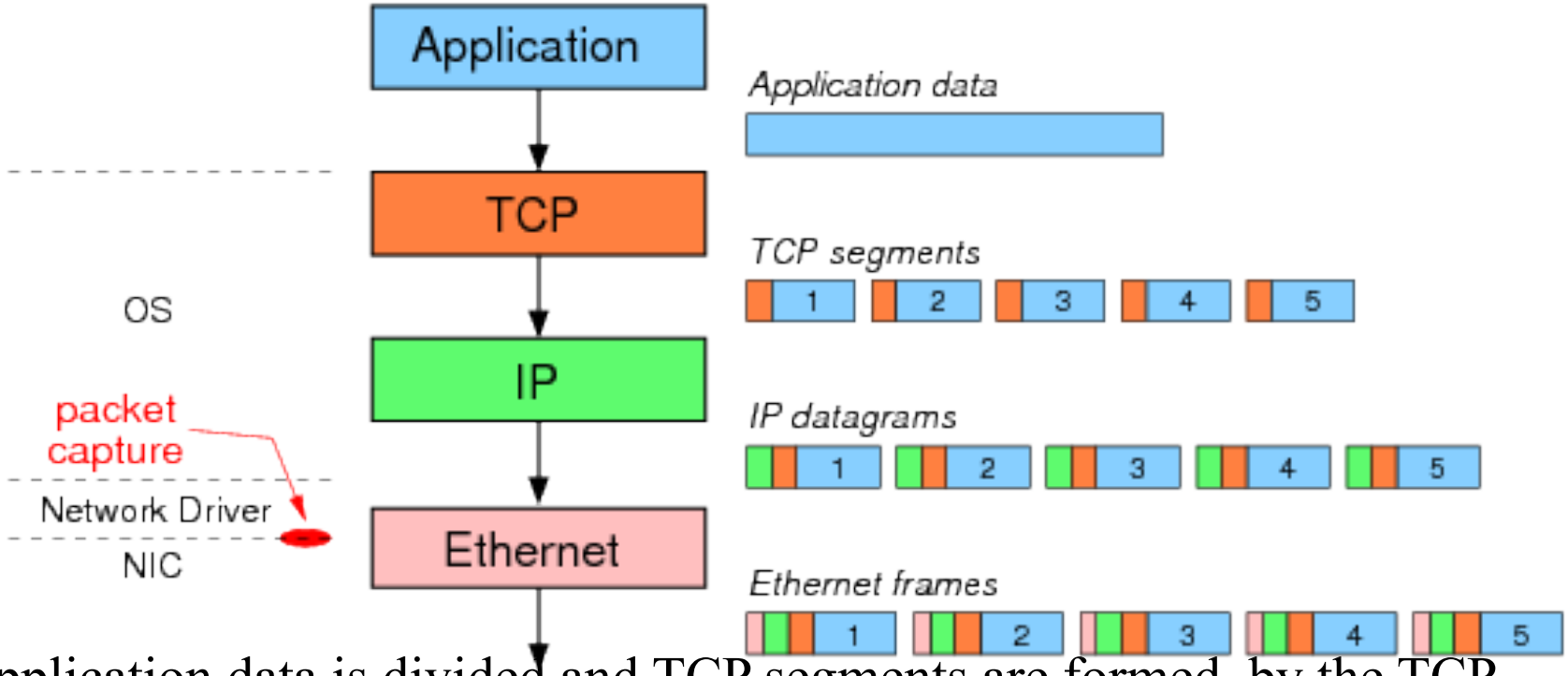
Socket H: (IP-C, IP-B, 2, 2)

Each connection is uniquely Identified with 4-tuple.

TCP Connection between two hosts

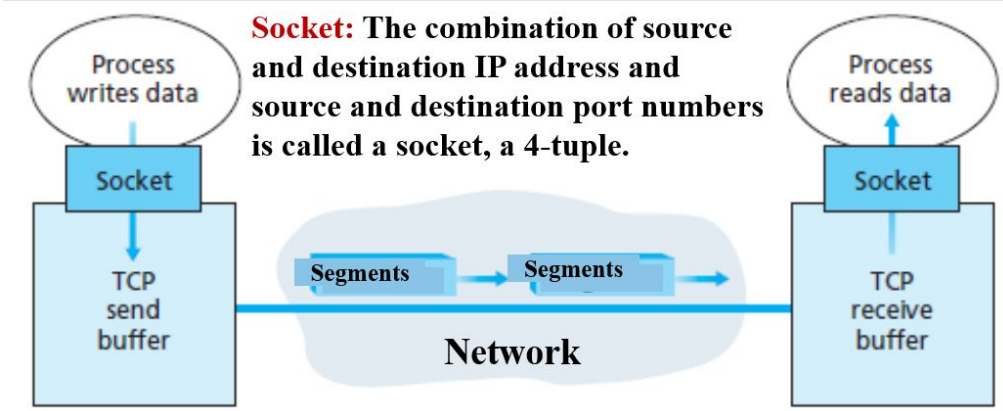
- TCP connection is always between only two end-points (two hosts)
- The socket values of each end is the same (order is different)
  - Thus, a socket identifies a connection uniquely
- The sockets within a host will also have a unique value because there cannot be more than one connection between the same set of ports on the same set of hosts
- A host can create multiple connections using the same source port as long as at least one of the other parameters (destination IP or destination port) is different.

TCP Segments going over the Network



- Application data is divided and TCP segments are formed, by the TCP layer.
  - The amount of application data on each segment is based on what has been decided by the hosts during the connection establishment
- Which is then made into IP datagrams and sent as Frames over the physical network (here Ethernet frames)

TCP Connections



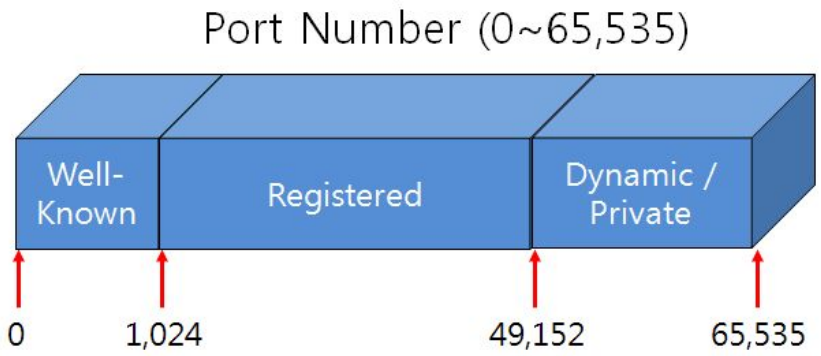
TDM: Time Division Multiplexing
FDM: Frequency Division Multiplexing
Note: In TDM or FDM, there is a physical circuit established between two nodes.

In **Virtual circuit**, the intermediate nodes maintains the information about a connection established between two nodes. E.g.: **VCI**: Virtual Circuit Identifier

- The TCP "connection" is not an end-to-end TDM or FDM circuit as in a circuit switched network. Nor is it a virtual circuit, as the **connection state** resides **entirely** in the **two end systems**.
- Because the **TCP protocol runs only in the end systems** and not in the intermediate network elements (routers and link-layer switches), the intermediate network elements do not maintain TCP connection state
- In fact, the intermediate routers are completely oblivious (unaware of) to TCP connections; they just see IP datagrams, not TCP connections or segments.
  - Segments are put inside an IP datagram; routers see only the IP header and not its payload

TCP/UDP- Ports

TCP/UDP Port Allocations – Well-Known Ports



How many **bits** are reserved for **port numbers** on the **TCP header**? **ANS: 16 bits**

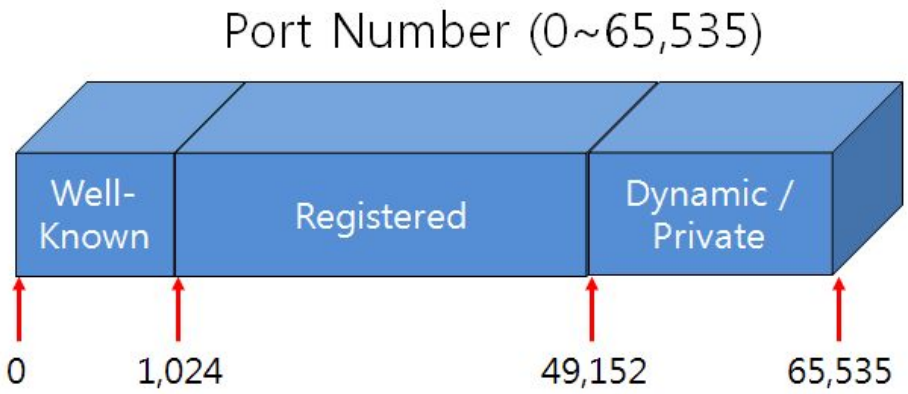
Well-Known ports These ports are reserved for widely used protocols and services, ensuring consistent communication across different systems and networks.

The **port** numbers direct packets to the appropriate applications running in the servers.

- The **port numbers** in the range from **0 to 1023** are the **Well-Known ports**.
- They are used by system processes that provide widely used types of network services (FTP, HTTP, DNS, SMTP, DHCP)
  - On Unix-like operating systems, a process must execute with **super-user privileges** to be able to bind a network socket to an IP address using one of the well-known ports
- The **dynamic port numbers** (also known as the private port numbers) from **49,152 to 65,535** are the port numbers that are available for **temporary use** by any application for communicating with any other application
- These private ports cannot be registered with **IANA**.

IANA: Internet Assigned Numbers Authority

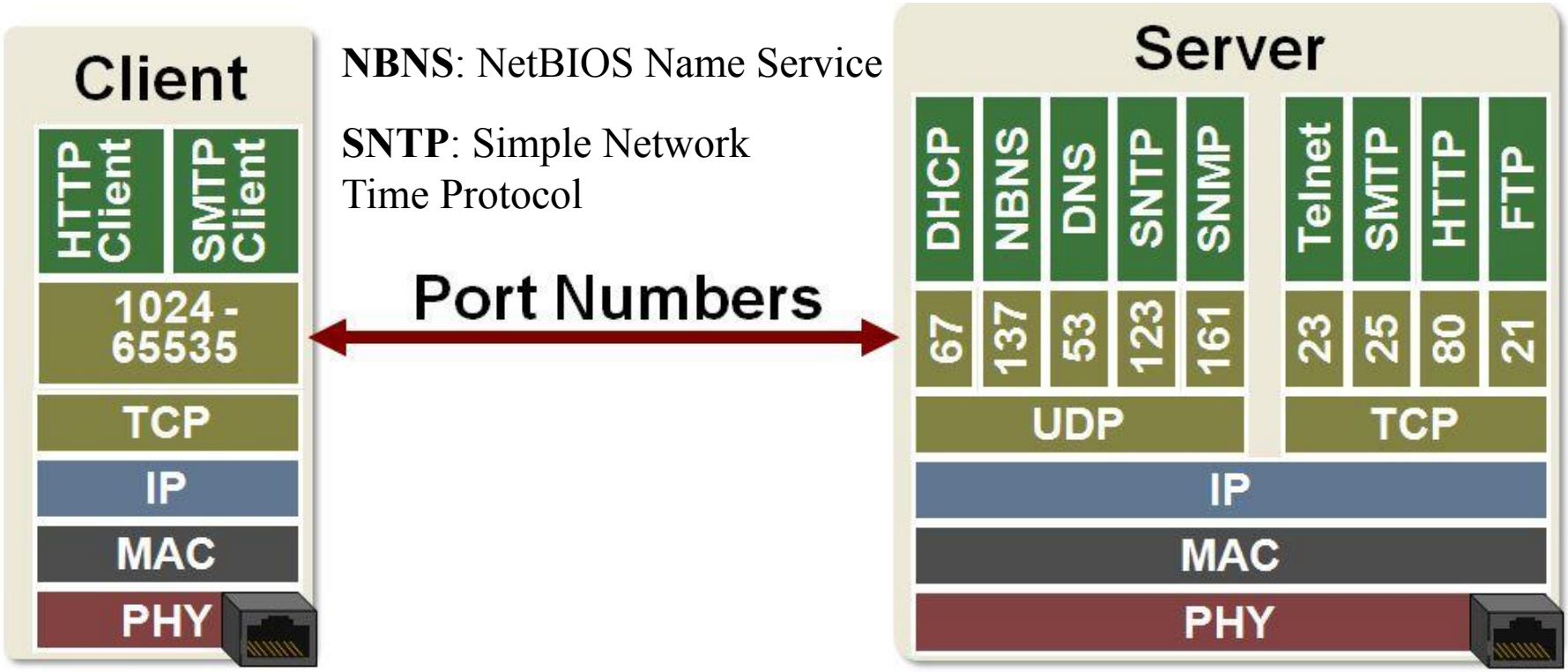
TCP/UDP Port Allocations: Registered Ports



- The range of port numbers from **1024 to 49151** are the **registered ports**.
- They are assigned by **IANA** for specific service, based on specific network applications offered by any organization.
- Commonly used by proprietary applications, third-party software, and specialized services that are not covered by Well-Known Ports
  - **Example**: Microsoft SQL Server (1433), NFS (2049), MySQL (3306), SIP (5060), etc.
- Administrative privileges are not required, user applications can use it
- On most systems, registered ports can be used by ordinary users as well

NFS: Network File System (file sharing), SIP: Session Initiation Protocol (for VoIP applications)

Ports used by Servers and Clients

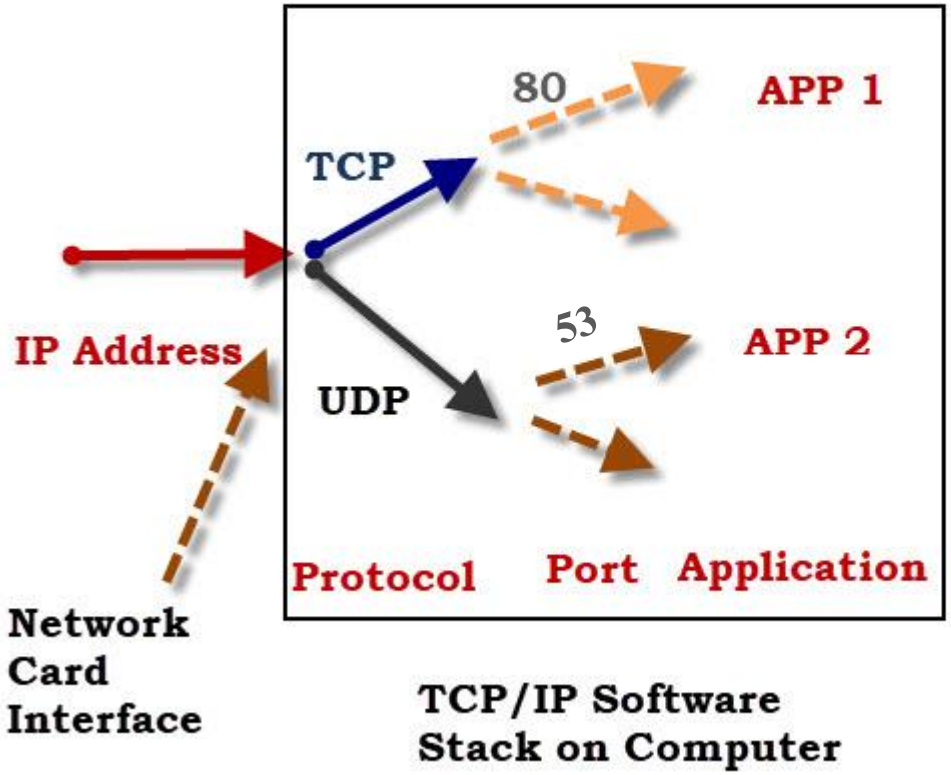


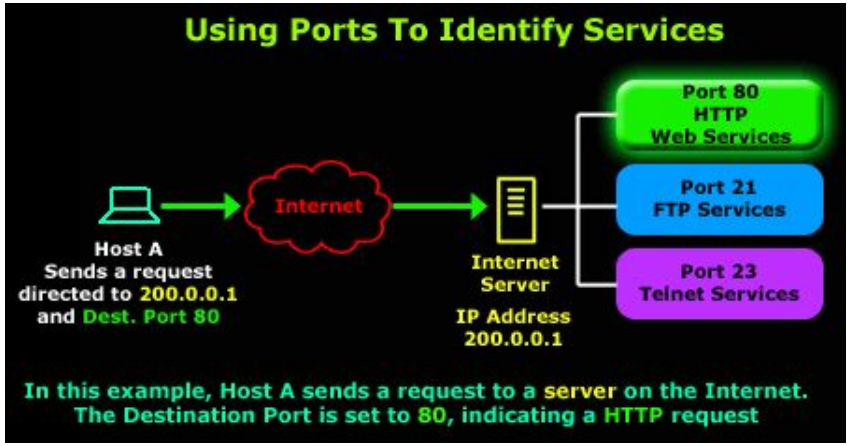
- Well defined Server ports for specific purposes/applications
- Clients connect to the specific server ports based on the application
  - Web browsers (clients) connect to HTTP server ports (80) on the Web server
  - Port number 8080 is also used for web services, as an alternative port for HTTP traffic, often used for proxy servers, for development, and testing

Services mapped to Ports on a Host

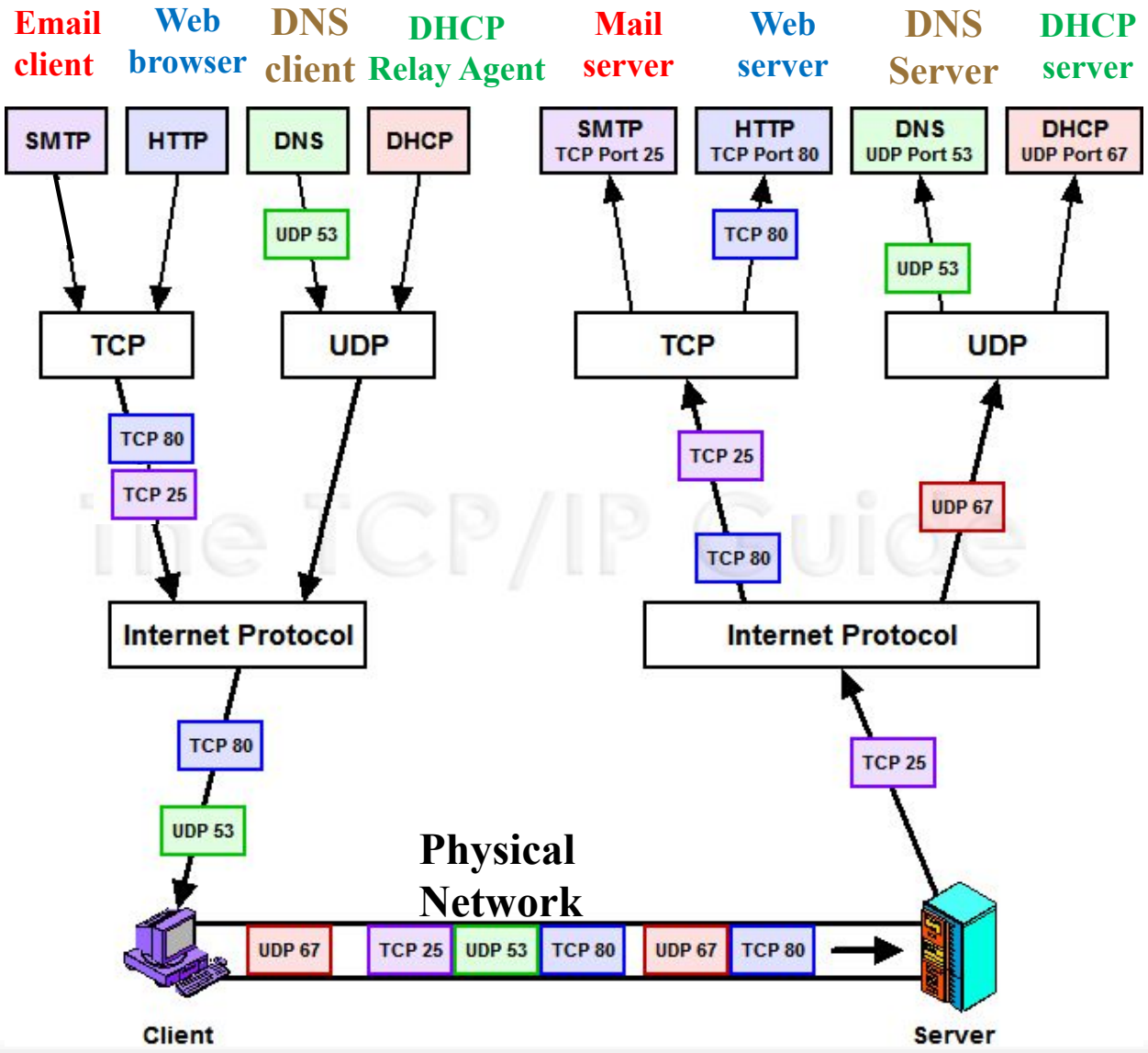
RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Packet Routing - IP,Protocol and Port





Flow of TCP and UDP Segments (Client-server model)

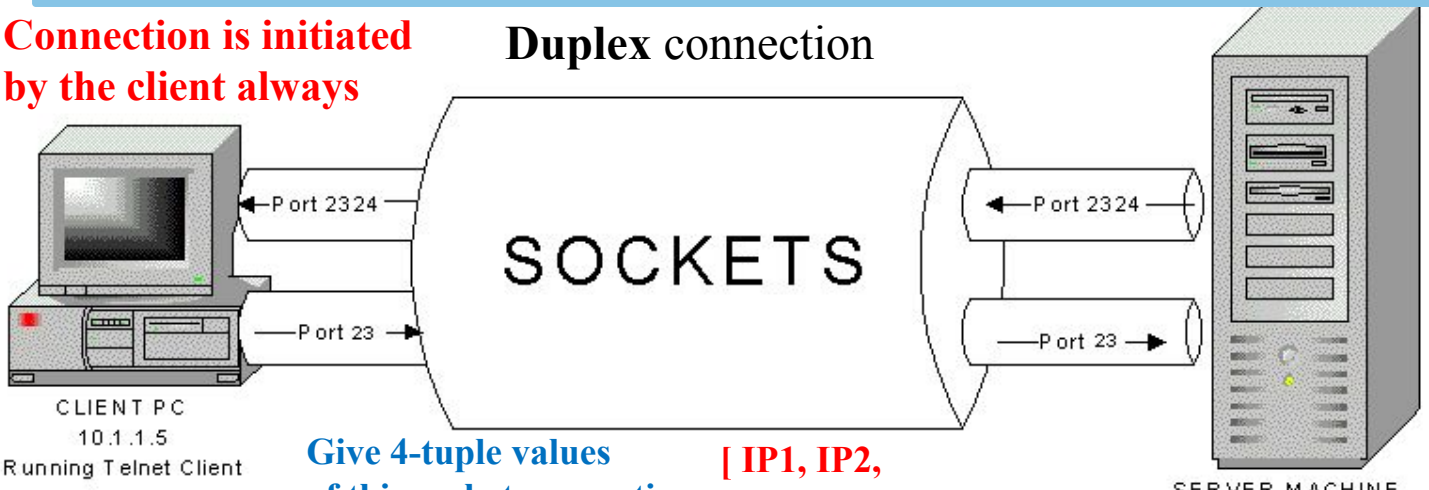


TCP/IP Sockets

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Connection is initiated by the client always

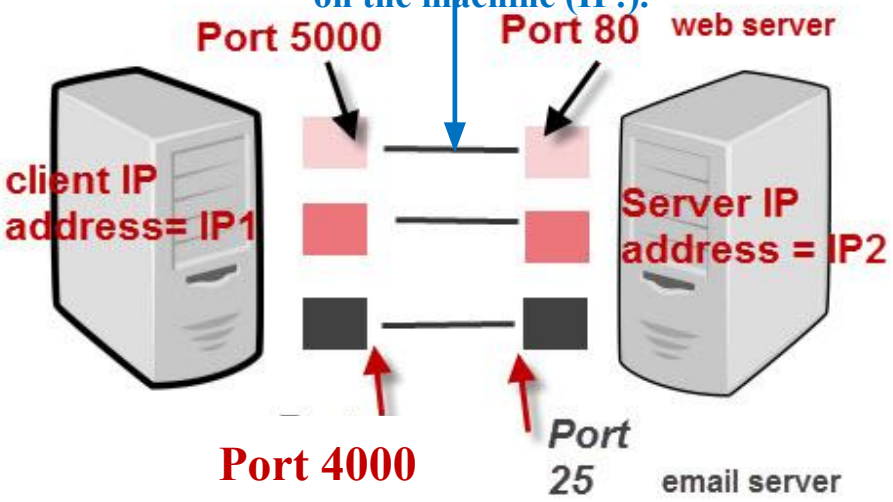
Duplex connection

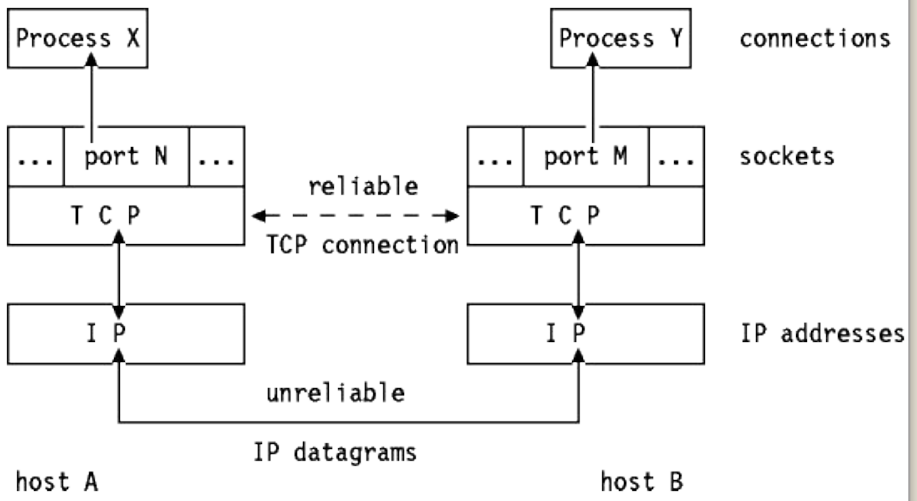


Telnet: Remote login

Telnet is a protocol used on the Internet or LAN to provide a bidirectional interactive text-oriented communication facility using a virtual terminal connection

Give 4-tuple values of this socket connection, on the machine (IP!). **[IP1, IP2, 5000, 80]**

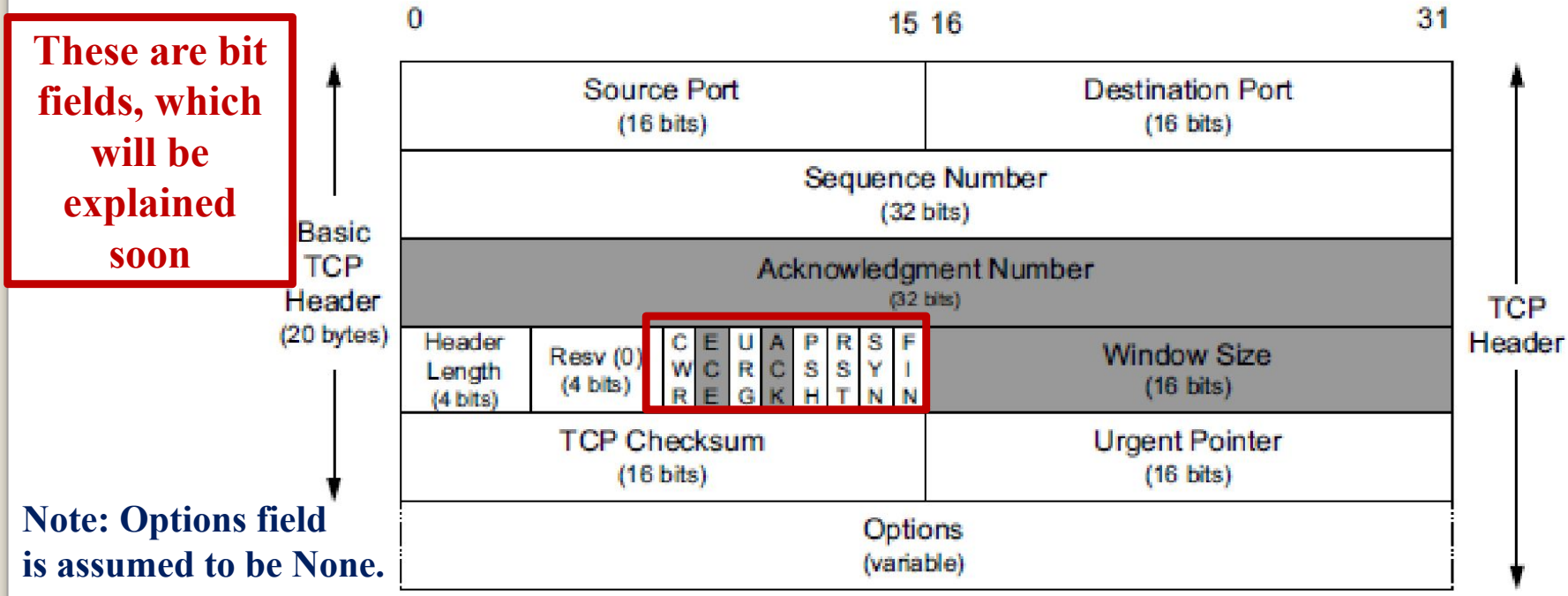




IP Address + Port number = Socket

TCP Segment Structure

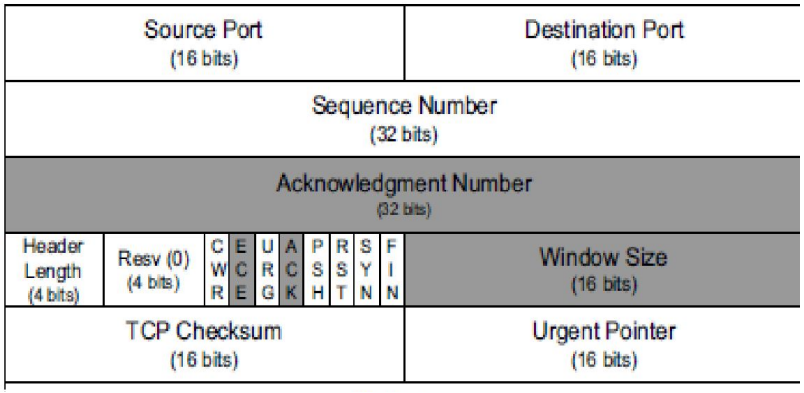
TCP Segment Structure (Header)



- The **shaded fields** (Acknowledgment Number, Window Size, plus ECE and ACK bits) **refer to** the **data flowing** in the **opposite direction relative** to the **sender of this segment**.
- Remember, TCP is a duplex connection, between two end points. The segments from a sender carry some information about the data the sender has received from the other end, in the shaded fields. We will study shortly

TCP Header Fields: Sequence Number

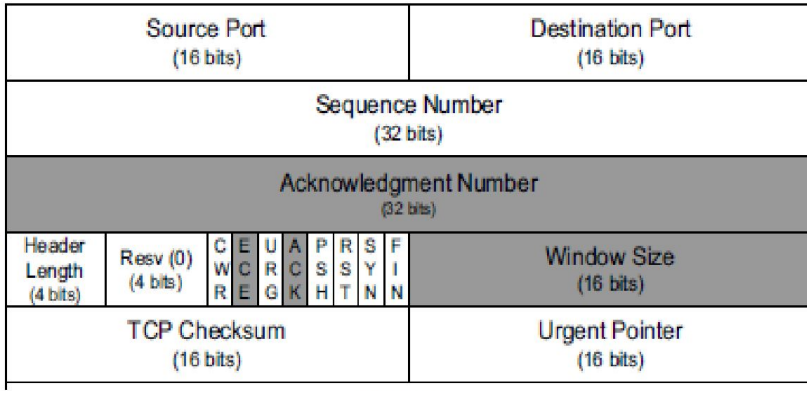
- Recall, TCP is used to exchange a byte-stream of data. The data could be a file, email, HTML page, etc.
- TCP needs to make sure that every byte sent by the sender reaches the other end reliably.



- The **Sequence Number** field identifies the **byte** in the **stream of data** from the **sending TCP** to the **receiving TCP.** It **refers** to the **first byte of data** being carried by the segment, of which this header is part of.
- If we consider the stream of bytes flowing in one direction between two applications, TCP numbers each byte with a sequence number.
- This **sequence number** is a **32-bit unsigned** number that wraps back around to 0 after reaching $(2^{32}) - 1$.
- When a host initiates a TCP session, the **Initial Sequence Number (ISN)** is always chosen at **random**. As the data gets exchanged, it wraps around.

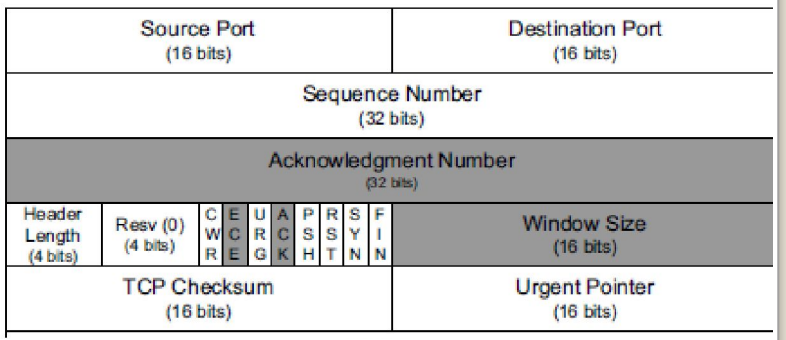
TCP Header Fields: Acknowledgement Number

- As you are aware, the TCP connection is reliable, the receiver sends acknowledgment to the sender when the data is received correctly by it
- Since the connection is duplex, normally acknowledgment is sent along with the data flowing in the other direction



- Because **every byte** exchanged is **numbered**, the Acknowledgment Number field (also called the **ACK Number** or **ACK field** for short) contains the **next sequence number** that the **sender** of the **acknowledgment expects** to **receive**.
- This is therefore the sequence number of the last successfully received byte of data plus 1.
- This field is valid only if the ACK bit field (described later in this section) is ON (set to one), which it usually is ON for all but initial and closing segments.

TCP Header Fields: Acknowledgement Number



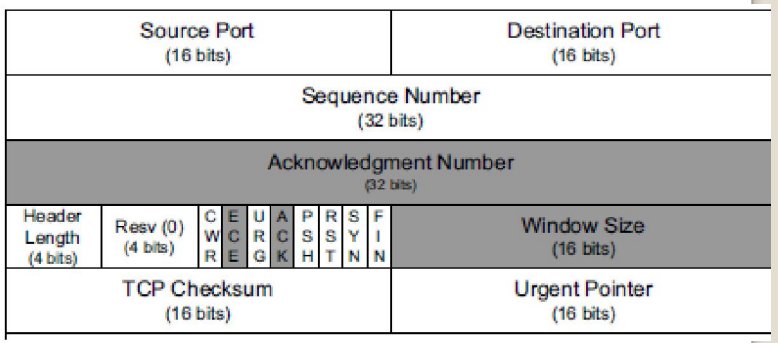
- Sending an ACK costs nothing more than sending any other TCP segment because the 32-bit ACK Number field is always part of the header, as is the ACK bit field.
- ACK and Sequence numbers represent the data flowing in two different directions
- As the **ISN** (Initial Sequence Numbers) are chosen by the senders **at random** while establishing the connection, the ACK and Sequence numbers will be totally different though they are part of a Segment header field
  - Because the first sequence number of the data flowing in the reverse direction would have been chosen by the receiver at random
- Suppose, a client machine receives a TCP segment, with a sequence number 100, and a data size of 50 bytes.

What would be the values of ACK no. and ACK bit when the client sends a TCP segment to the other end?

- When this client machine has some data to be sent to other end of the connection, it would make the ACK number as **150** and set ACK bit, in the TCP segment that is carrying its own data to the machine on the other end

TCP Header Fields: TCP Checksum

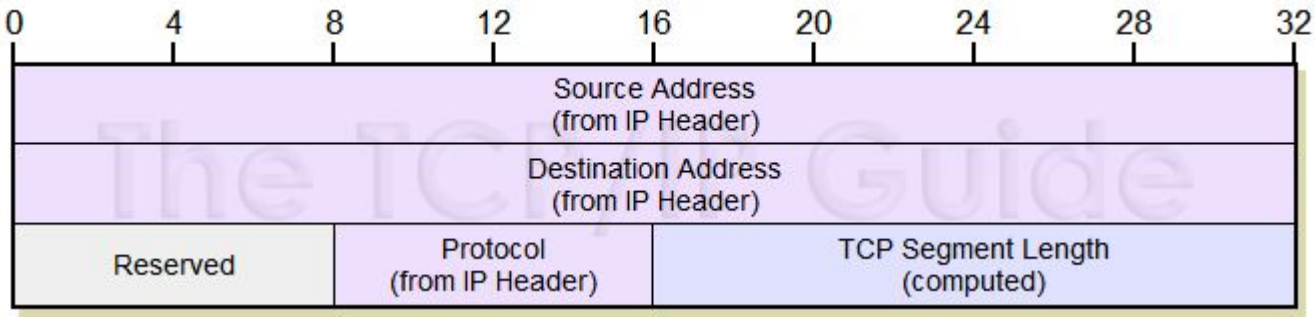
- The TCP/IP checksum is used to detect corruption of data over a TCP or IPv4 connection.
- If a bit is flipped, a byte mangled, or some other badness happens to a packet, then it is highly likely that the receiver of that broken packet will notice the problem due to a checksum mismatch.



- This provides end-to-end assurance that the data stream is correct.
- The TCP protocol includes an extra checksum that protects the packet "payload" as well as the header. This is in addition to the header-checksum of IP.
- The algorithm for the TCP and IPv4 checksums is identical. The data is processed a word (16 bits, two bytes) at a time.
- The TCP Checksum field covers the TCP header and data and some fields in the IP header, which is called a pseudo-header.

Pseudo-Header from IP Header Used for TCP Checksum

TCP "**Pseudo Header**" for TCP's Checksum Calculation

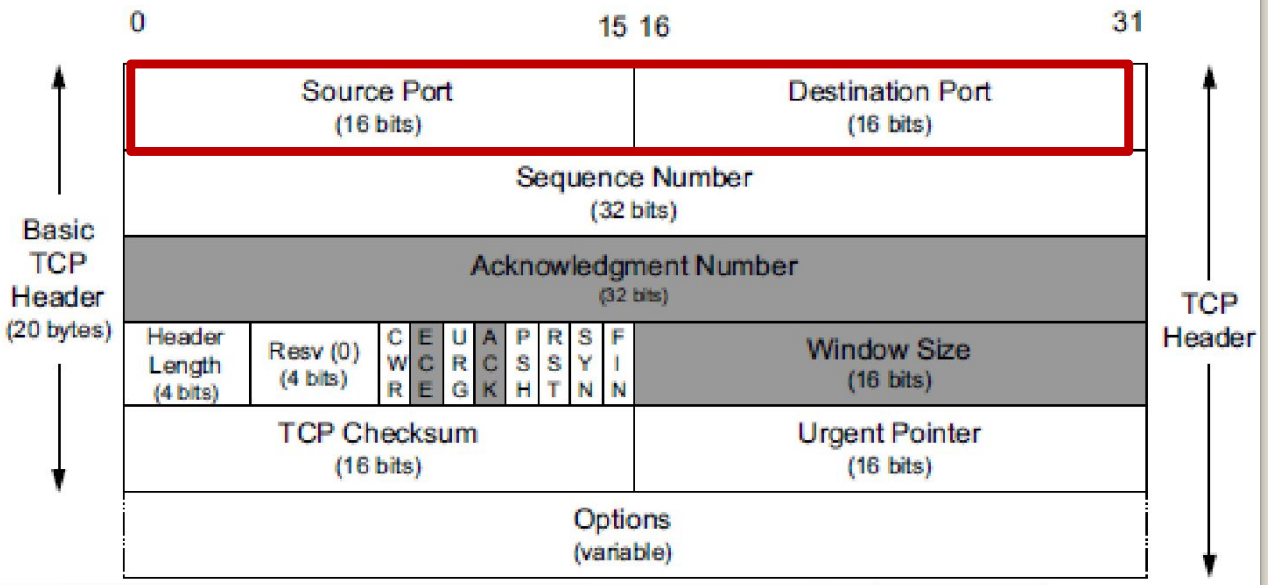


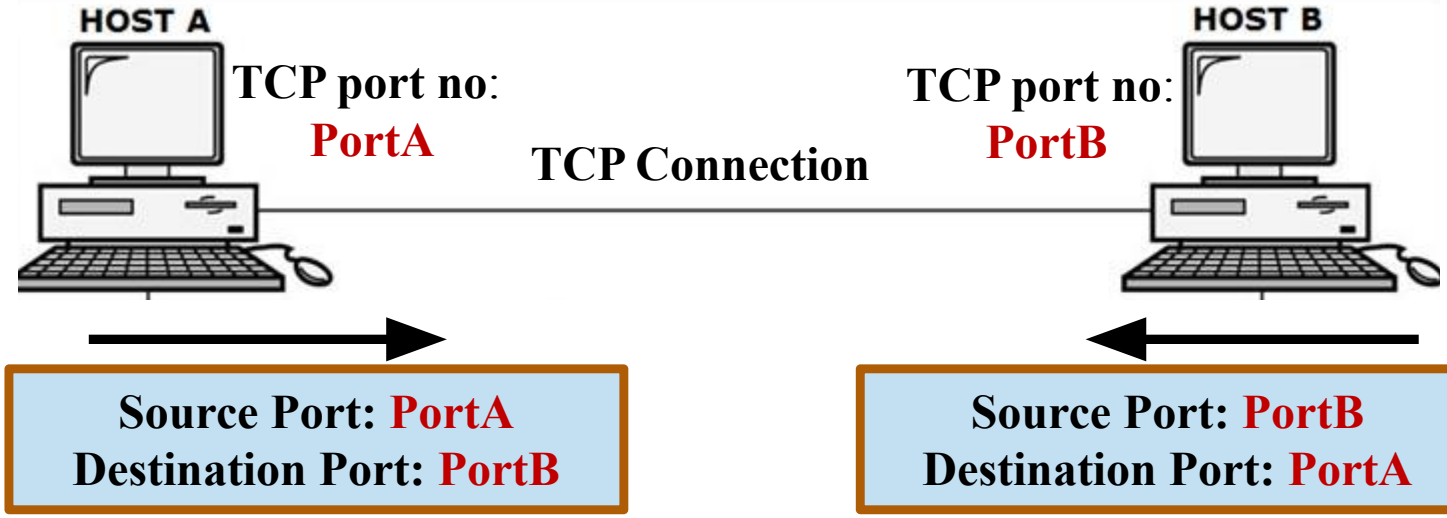
- Instead of computing the checksum over only the actual TCP header and data fields of the TCP segment, the above 12-byte from the IP header, on which this TCP segment is going to be part of, is also included in the checksum calculation
- Addition of "pseudo header" is done to make sure that at the receiving end, the host can be assured that the received TCP segment is indeed from the original sender that is addressed to it
  - This takes care of detecting "man-in the middle attack" if the TCP segment is tampered by someone enroute or corrupted IP packets being accepted by TCP

TCP Header Field: Source and Destination Ports

Note: The 16-bit values of **Server ports** and **Client ports**

Note: Based on the role played by the host either as a Client or a Server, the **TCP connections** are **made** to **relevant ports**.

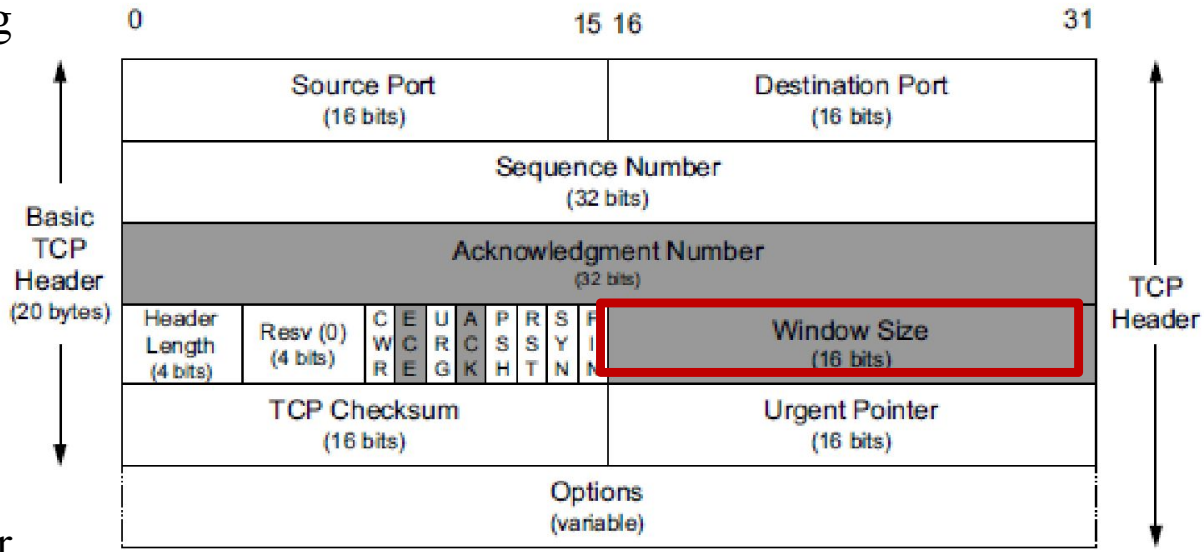




TCP Header Field: Window Size

Flow control: The receiver making sure, that the sender does not overflow its buffers by sending data to the receiver at a higher rate than it can accept and process.

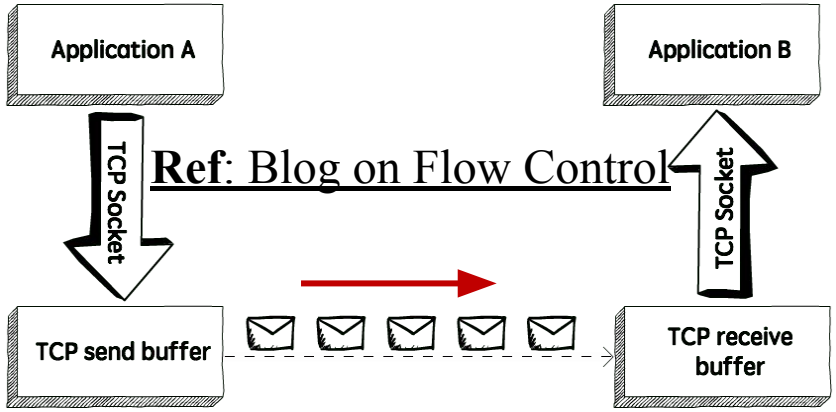
Reason: The sender could be a faster and powerful machine than the receiver, thus capable of generating data much faster than the rate of consumption by receiver.



- **TCP's flow control** is provided by each end **advertising** a **window size** using the Window Size field. This is the number of bytes, starting with the one specified by the ACK number, that the receiver is willing to accept.
- This is a 16-bit field, limiting the window to **65,535 bytes**, and thereby limiting TCP's throughput performance.
  - Even if the underlying network technology supports a higher rate of transmission, this window size limits what can be sent in one go, until the acknowledgment for the data that has been sent already, is received.

Flow Control: Using Window Size

- TCP stores the data it needs to send in the **send buffer**, and the data it receives in the **receive buffer**.
- Application and TCP/IP stack run as separate processes on the host and they exchange data between them.

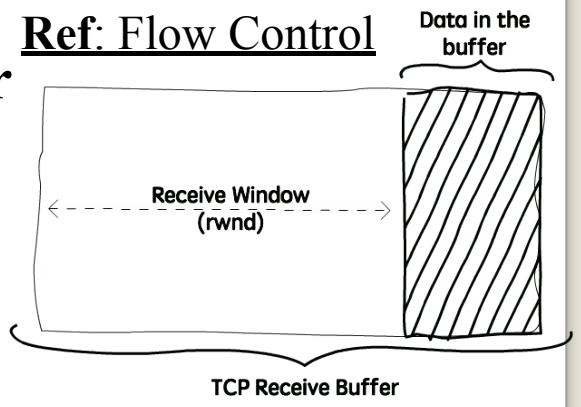


- When the application is ready, it will then read the data from the receive buffer.
- **Flow Control** is all about making sure sender doesn't send more packets when the receive buffer is already full, as the receiver wouldn't be able to handle them and would need to drop those packets.
- Dropping segments would mean that they need to be sent by the sender again which would introduce additional delays in completing the data transfer between the hosts.

Flow Control: Using Window Size ..contd..

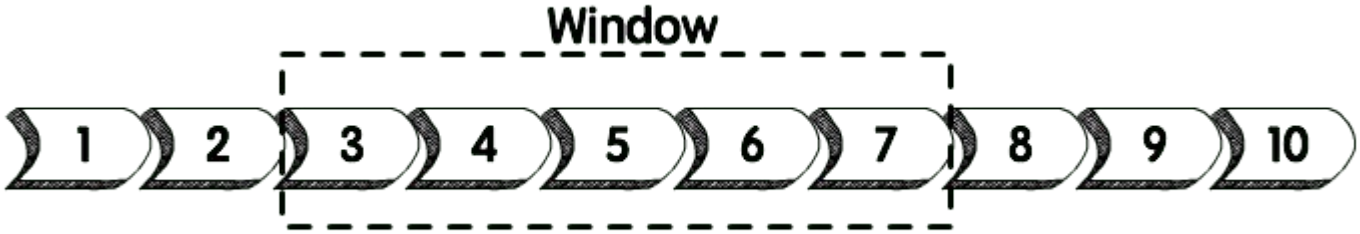
- To control the amount of data that TCP can send, the receiver will advertise its Receive Window (**rwnd or Window Size**), that is, the spare room in the receive buffer.
- "Data in the buffer" (shaded region) is the data for which the receiver has already sent the ACK to the sender.

Ref: Flow Control



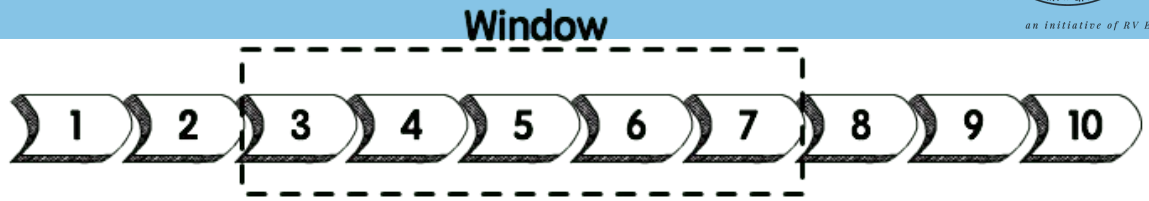
- Every time TCP receives a packet, it needs to send an **ACK** message to the sender, acknowledging that it received that packet correctly.
- And with this **ACK** message it sends the value of the **current receive window (in the window size field)**, so that the sender knows whether it can keep sending data and if yes, how many more bytes can be accommodated in the receive buffer, before receiving another ACK from the receiver .
- This is an adaptive flow control based on the current buffer available at the receiver.

Sliding Window Protocol



- TCP uses a **sliding window protocol** to control the number of bytes in flight, it can have. In other words, the number of bytes that were sent but not yet **ACKed** by the receiver
- Let's say we want to send a file with a size of 150,000 bytes from a node A to another node B.
- TCP could break this file down into 100 packets, 1500 bytes each.
- Assume that after the connection between node A and B is established, node B advertises a receive window of 45,000 bytes (based on it receive buffer size)
- Seeing that, TCP at the sender knows it can send the first 30 segments (1500 * 30 = 45000) before it receives an acknowledgment from node B.

Sliding Window Protocol … contd.



- If it gets an ACK message for the first 10 packets (meaning we now have only 20 packets in flight), and if the receive window present in these ACK messages is still 45,000, it can send the next 10 segments.
- This makes the number of segments in flight back to 30, that is the limit defined by the receive window.
- In other words, at any given point in time, there can be 30 segments in flight, that were sent but not yet **ACKed.**
- Now, if for some reason the application reading the data in node B slows down, TCP will still ACK the packets that were correctly received, but as these packets need to be stored in the receive buffer until the application decides to read them, the receive window will be made smaller (say 30,000) to slow down the sender.

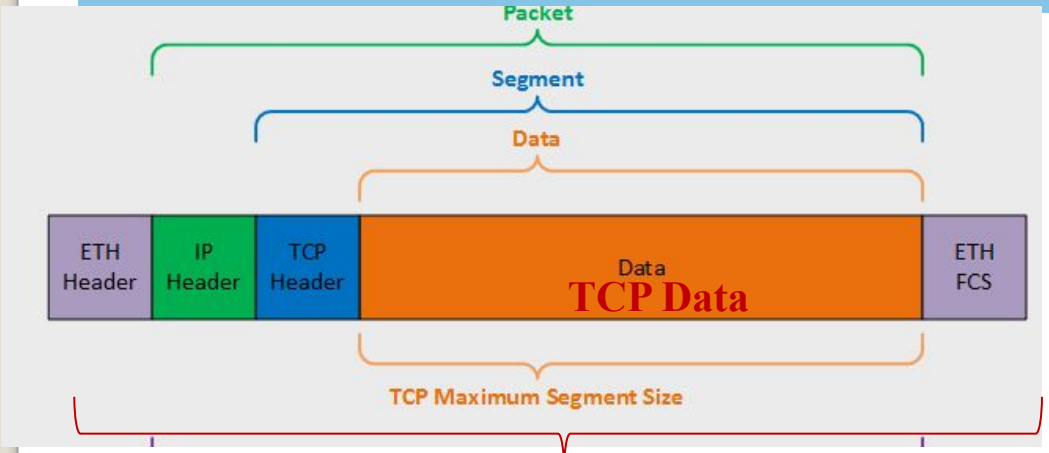
TCP Header: Options Field & Header Length

TCP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset **HDR Len**				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																	Urgent pointer (if URG set)														
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) **Max 40 bytes**																															

- The TCP header (without the Options field) is 20 bytes in length.
- The **Header Length** field gives the **length** of the **header** in **32-bit words**.
- This is required because the **length** of the ***Options*** field is **variable**.
- With a 4-bit Data Offset field, TCP is limited to a 60-byte header. Without options, however, the size is 20 bytes. **Note**: Max Options field length is 0-40 bytes
- It provides a way to deal with limitations of the original **TCP header.**

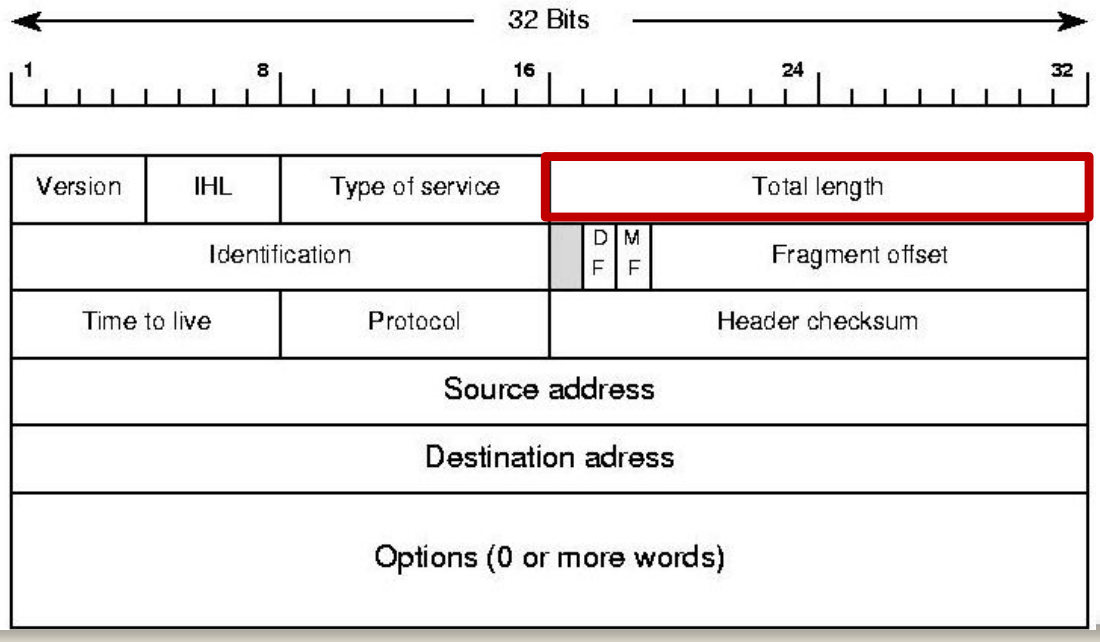
TCP Data Length (not part of TCP header)



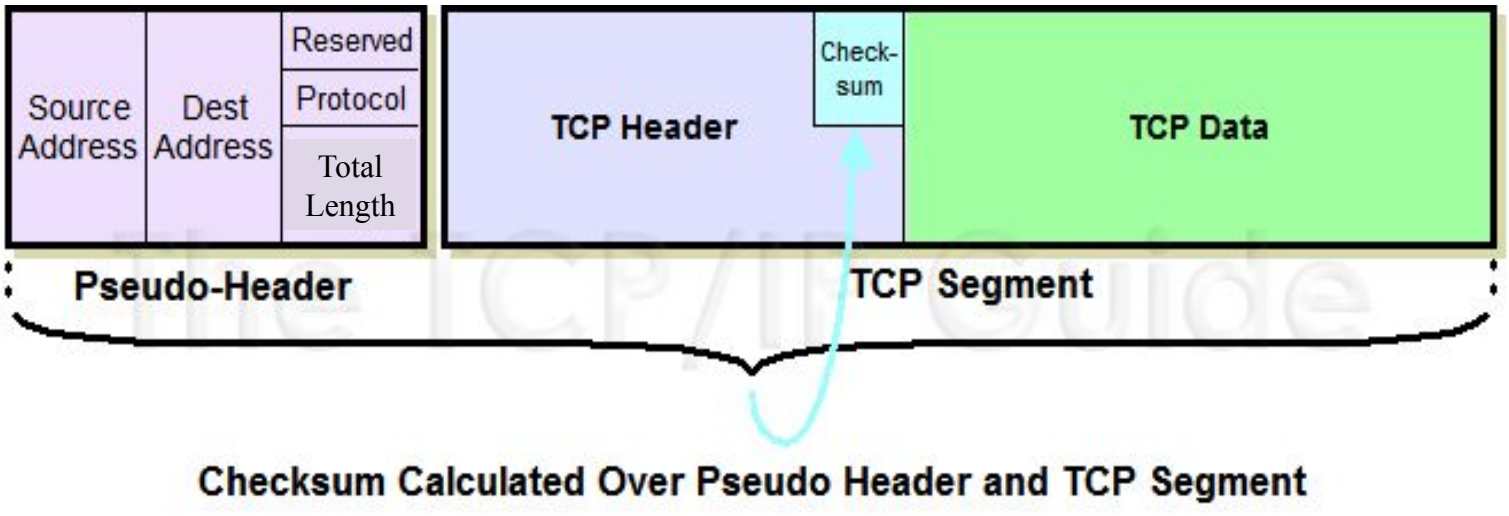
- The "**Total Length**" field in **IP header** captures the **length** of TCP Data + Header, in **bytes**.
- There is no separate field in TCP header for TCP data length.

Total length: Field in the IP header contains the Length in bytes of the IP header and its payload

IP Header



TCP Checksum Calculation with "Pseudo Header values" from IP Header



- It shows how the TCP check sum is calculated by including the contents of part of the IP header
- Pseudo-header is of 12 bytes:
  - Source and Destination IP Addresses in the IPv4 header: 4 + 4 = **8 bytes**
  - Total length (IP header): **2 bytes**
  - Protocol field: **1 byte**
  - Reserved (filled with all zeros): **1 byte**

Quiz 1a: TCP Header Fields

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

If the host is generating the below TCP **Segment 2**, after receiving the **Segment 1** on the right, then answer the questions given.

Segment 1

Source port										Destination port
Sequence number: 1450										
Acknowledgment number (if ACK set): 7320										
Data offset / HDR Len: 5	Reserved: 000	NS	CWR	ECE	URG	ACK: 1	PSH	RST	SYN	FIN	Window Size: 200
Checksum										Urgent pointer (if URG set)

TCP Data length: 100 bytes

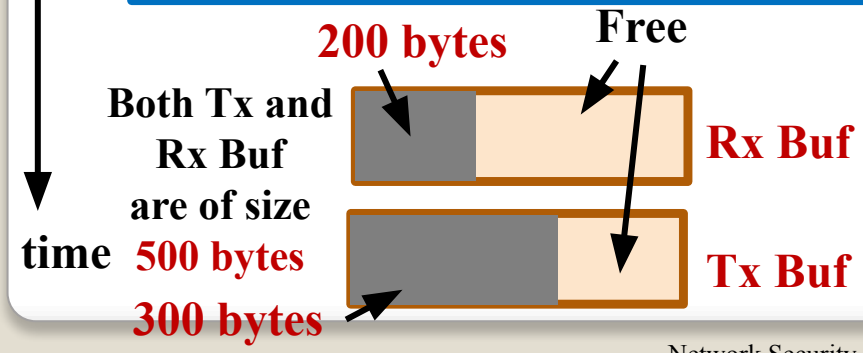
Segment 2

Source port										Destination port
Sequence number: 7320										
Acknowledgment number (if ACK set): 1550										
Data offset / HDR Len	Reserved: 000	NS	CWR	ECE	URG	ACK: 1	PSH	RST	SYN	FIN	Window Size: 300
Checksum										Urgent pointer (if URG set)

TCP Data length: 200 bytes

Q1: Fill in the Window size, ACK no. and Seq no. in the Segment 2

Note: Assume no other segment is in transit from this host to the other end as well as from the other end to this host, at this moment.



The **status** of **Rx buffer** after **receiving** the above **Segment 1** is shown here.

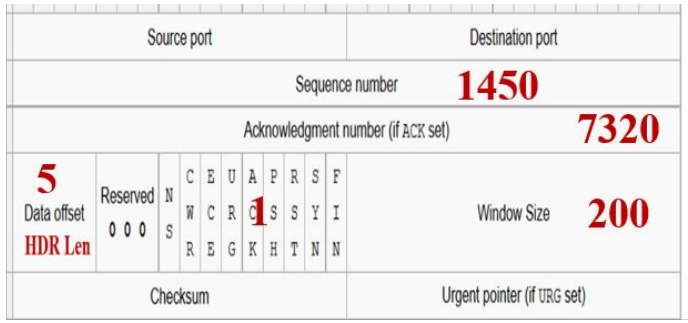
The **status** of **Tx buffer** after **completing** the above **Segment 2,** which is ready to be sent, is shown here.

Quiz 1b: TCP Header Fields contd.

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Q2: The Rx buf was empty before receiving the Segment 1. (Yes/No) ANS2: NO. It had 100 bytes.

Q3: The Tx buf was full prior to preparing the Segment 2. (Yes/No) ANS3: Yes.



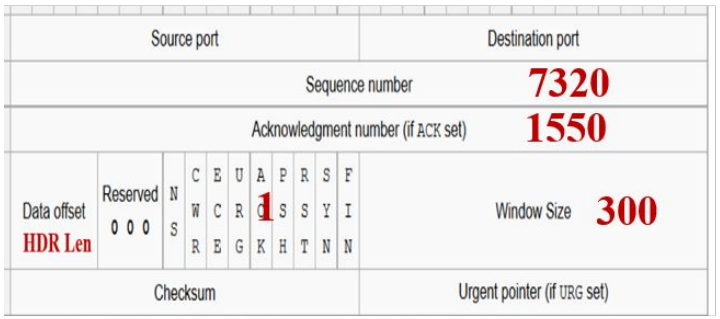
TCP Data length: 100 bytes

Segment 1

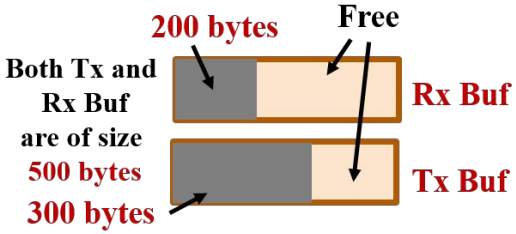
Q4: What can you say about the status of the Rx buf of the receiver after receiving this Segment 2. ANS4: It will be full

Note for Q4: Assume that the application receiving this data on the other end has not read any data from the moment Segment1 was sent out, till the Segment 2 is received.

Segment 2



TCP Data length: 200 bytes



The **status** of **Rx buffer** after **receiving** the above **Segment 1** is shown here.

Note: Remember that nothing can be said about the total size of the Tx and Rx bufs at the receiver, because it need not be same as what is on this machine.

The **status** of **Tx buffer** after **preparing** the above **Segment 2** is shown here.

time

TCP Header: With 9 Control Bits

Ref: From Wikipedia

RV UNIVERSITY
Go, change the world

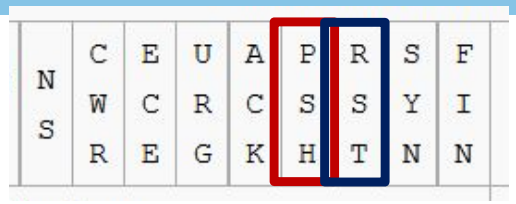
TCP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																	Urgent pointer (if URG set)														
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

- You can see a difference in the number of flags defined. There are 9 flags (control bits) including the NS bit. We will not study about **all** the control bits here. They are shown here for completeness.
- Currently **nine bit fields** are defined for the TCP header, although some older implementations understand only the last six of them.
- One or more of them can be turned on at the same time.

NS: This is not covered.

TCP Header Field: PSH (PUSH)

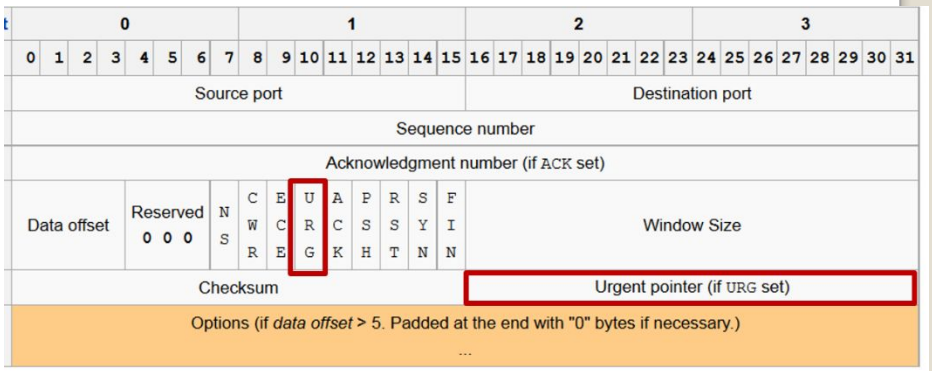


- **PSH: Push**
- It helps when the receiver receives a segment with PSH flag set, the receiving TCP stack should send the data immediately to the upper application layer, without waiting for the "Rx buf" to become full.
- The practical example of this is the **telnet** application where the application sends data in the form of few keystrokes or command lines.
- The telnet will become unusable if it waits for the **Rx buffer to become full** and then sends the data to the Telnet application.
- It should be sent as soon as one line of command has been typed by the user.
- User will be waiting for the response after typing a command.
- This flag has no influence on the protocol stack on the sender side.

RST: **Reset**: Resets the connection, typically used to abort a connection due to errors.

TCP Header Field: URG and Urgent Pointer

- **URG**: Urgent - The Urgent Pointer field is valid only if the URG bit field is set.
- This "pointer" is a positive offset that must be added to the Sequence Number field of the segment to yield the sequence number of the last byte of urgent data

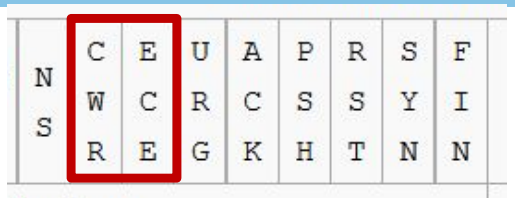


Example: When we press **Ctrl+C** during a telnet session URG flag would be set!!

Ref: Blog on URG and PSH

- The sender will not wait for the entire byte stream to be transmitted which is ahead of the urgent data. It is also called, **out-of-band data**.
- TCP's urgent mechanism is a way for the sender to provide specially marked data to the other end, out of sequence from the rest of the data.
- This also causes the receiving TCP to forward the urgent data on a separate channel to the application, ahead of already buffered data in the "Rx buffer".
- This allows the application to process the data out-of-band.

TCP Header Field: CWR and ECE



ECN: Explicit Congestion Notification (part of IP header).

CE: Congestion Experienced (Part of IP header)

Note: This feature needs to be supported by the network though which the TCP segment passes from the sender to receiver.

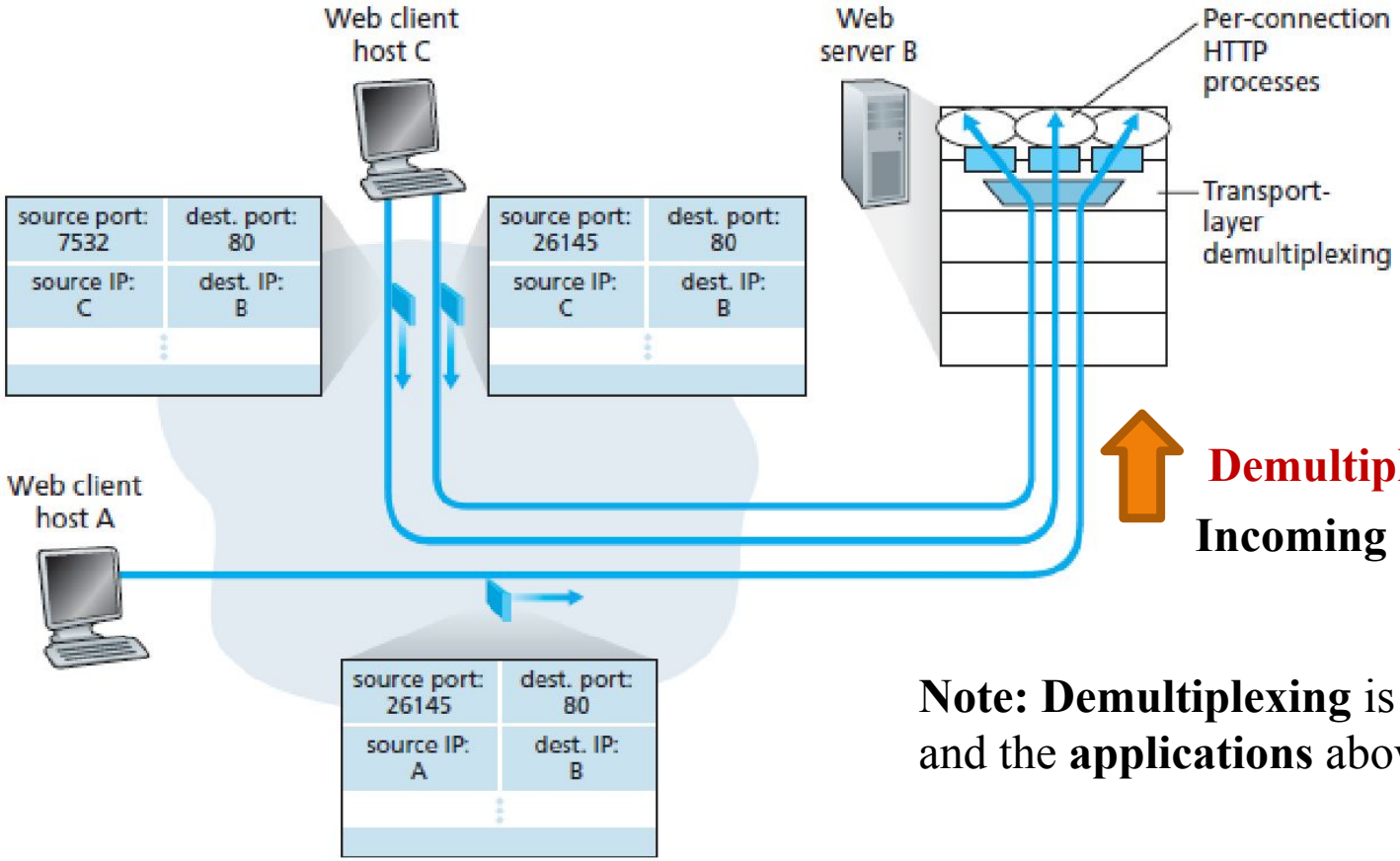
- **CWR**: Congestion Window Reduced. the sender reduced its sending rate
- **ECE**: Explicit Congestion Experienced. The sender received an Earlier Congestion Notification
- These two flags work together to enable the sender and the receiver of a TCP connection to be aware that there is a congestion in the network so that the senders can reduce their rates thus avoiding dropping of IP datagrams by the routers en-route.
- When a router detects congestion, rather than dropping packets destined to a receiver, it marks them with the CE flag in the IP header and delivers the packet to the receiver.
- Prior to acknowledging the receipt of the packet, the receiver sets the ECE flag in the TCP header of the ACK and sends it back to the sender.
- The sender having received the ECE marked ACK, responds by **halving the send window** and reducing the data rate thus reducing the congestion.
- During the connection setup this capability is exchanged.

ECN Workflow in Congestion Control

1. **Negotiation During Connection Setup**:
   During the TCP three-way handshake, both endpoints negotiate ECN support using specific bits in the TCP SYN packet:
   - **ECE Flag**: Indicates ECN capability.
   - **CWR Flag**: Reserved for congestion notifications.
2. **Marking by Routers**:
   If a router detects congestion (e.g., its queue is filling up), it:Marks the packet with the CE (11) codepoint in the IP header instead of dropping it.
3. **Receiver's Role**:
   When the receiver detects the CE mark in the packet, it sets the ECE flag in its next acknowledgment (ACK) to notify the sender about congestion.
4. **Sender's Role**:
   Upon receiving the acknowledgment with the ECE flag, the sender: Adjusts its congestion window (reduces the sending rate).
   Sends an acknowledgment with the CWR flag to inform the receiver that it has responded to the congestion.

TCP Features

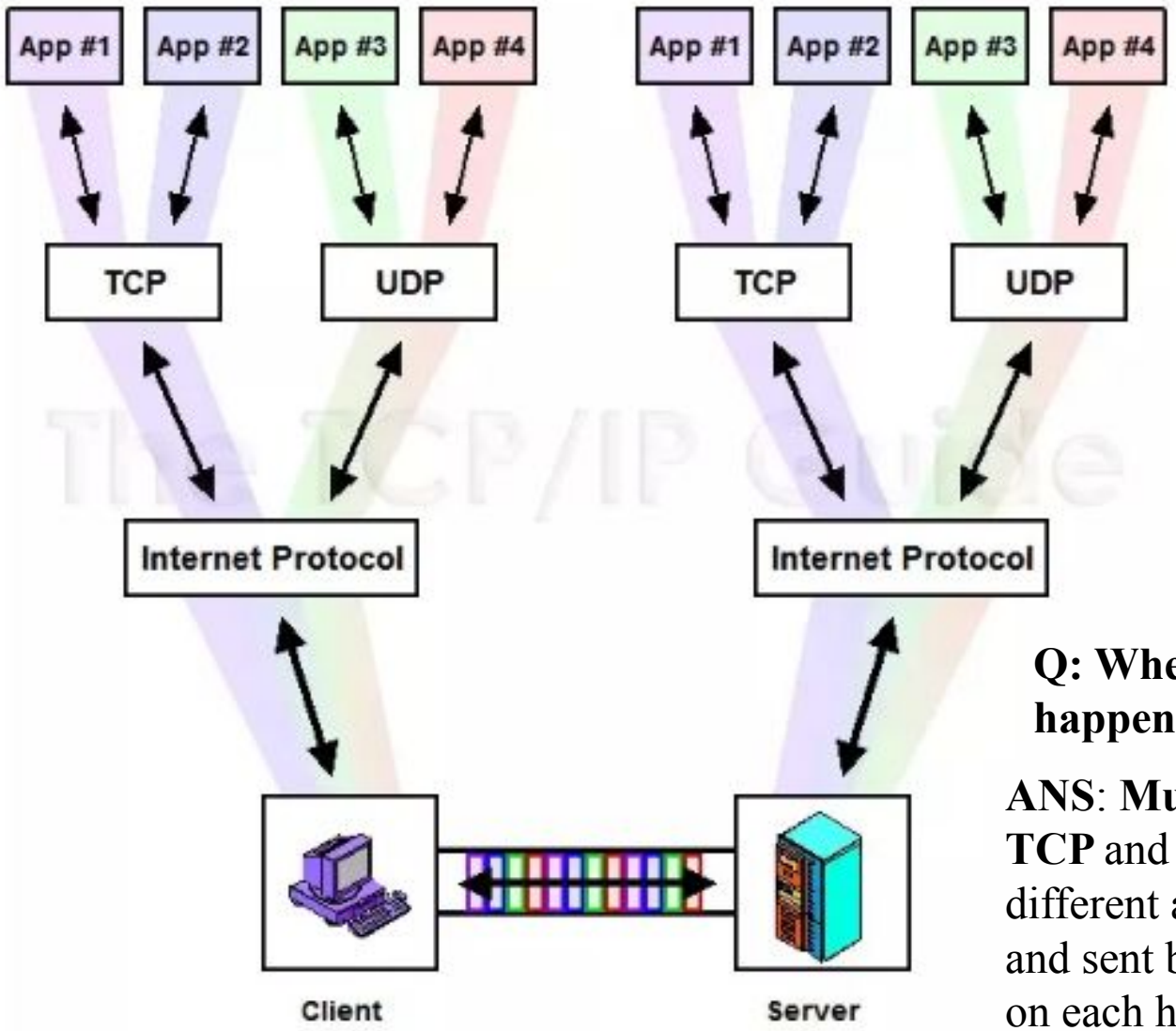
TCP: Demultiplexing



Note: Demultiplexing is between **TCP** and the **applications** above it

- The server host may support many simultaneous TCP connection sockets, with each socket attached to a process, and with each socket identified by its own four tuple.
- When a TCP segment arrives at the host, all four fields (source IP address, source port, destination IP address, destination port) are used to direct (**demultiplex**) the segment to the appropriate socket

TCP: Multiplexing



Q: Where does **Multiplexing happen** here?

ANS: **Multiplexing** happens between **TCP** and the **IP layer**. Segments from different applications are **multiplexed** and sent by a single IP layer running on each host

TCP Segment without any Data

- Suppose the receiver (host) of a TCP data segment does not have any data to send to the sender (host), the receiver still should ACK the data received, so that the sender can keep sending the data.
- In this scenario, the receiver of the TCP data segment sends a TCP header with **ACK bit set** and **Acknowledgement number** of the next data it is expecting from the sender.
- There is no data sent along with the segment from the receiver.
- The length in the IP header is adjusted accordingly.
- Note that the sequence number is not advanced, because no data is being sent here.

An ACK segment, if carrying no data, consumes no sequence number.

Quiz 1: Choose all the correct Options

- Why should the receiver of TCP data segments should ACK even if it does not have any data to be sent to the other end?

A. The receiver's Rx buffer will overflow if ACK is not sent to the sender

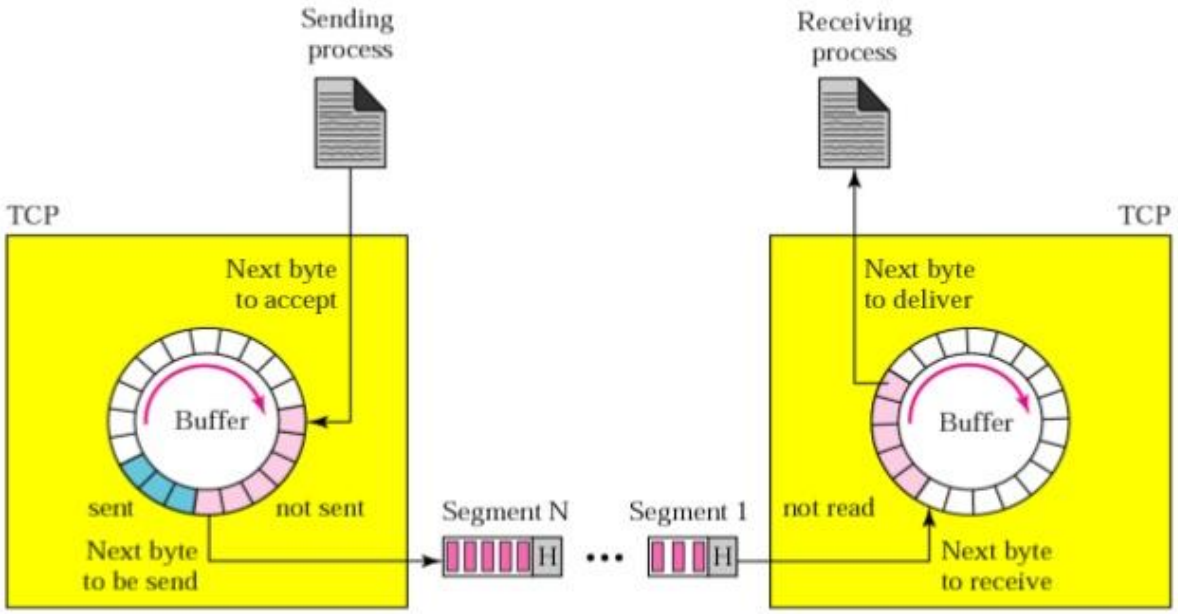
B. The sender's Tx buffer cannot be emptied or cleared if no ACK is received from the Rx side

C. The sender cannot take more data from the application, if it does not get ACKs from the receiver and it empties its Tx buf

D. If ACK is not received by the sender, timeout will happen on the sender and it will start resending the data again which is a waste of network bandwidth and time loss.

ANS: B, C and D

TCP: Tx Buf and Rx Buf



Both the Tx and Rx buffers are **circular buffers**, with read and Write pointers.

A **circular buffer**, **circular queue**, **cyclic buffer** or **ring buffer** is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end.

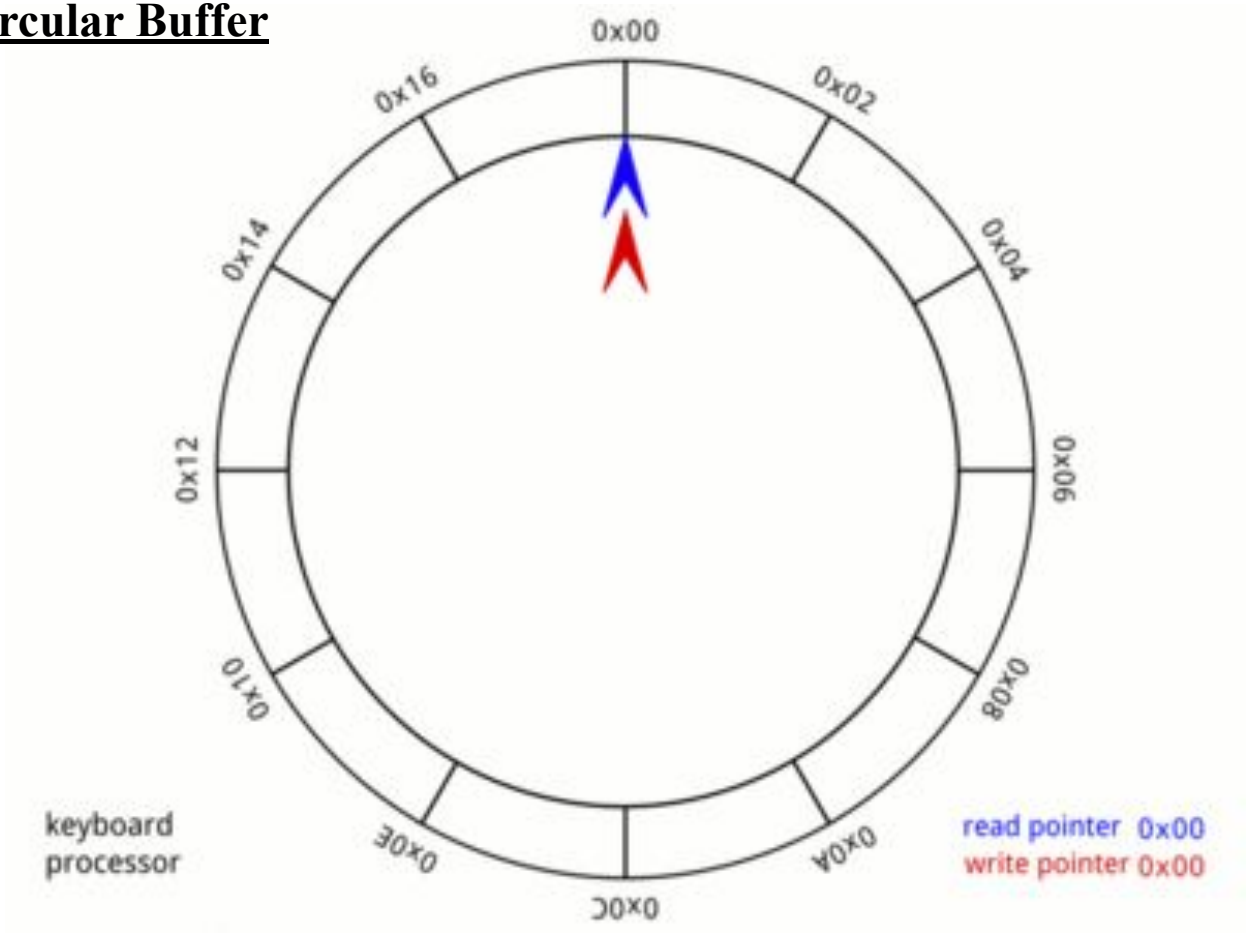
This structure lends itself easily to buffering data streams

- Since TCP is a byte streaming protocol and every bytes sent is to be acknowledged, the buffers maintained by the Tx and Rx need to be able to handle these requirements.
- Remember that TCP does not do any marking of data, unless some urgent data is to be sent to the other end.
- As the ACKs are received the Tx buffers are advanced, creating space for the writing of data by the sender process.

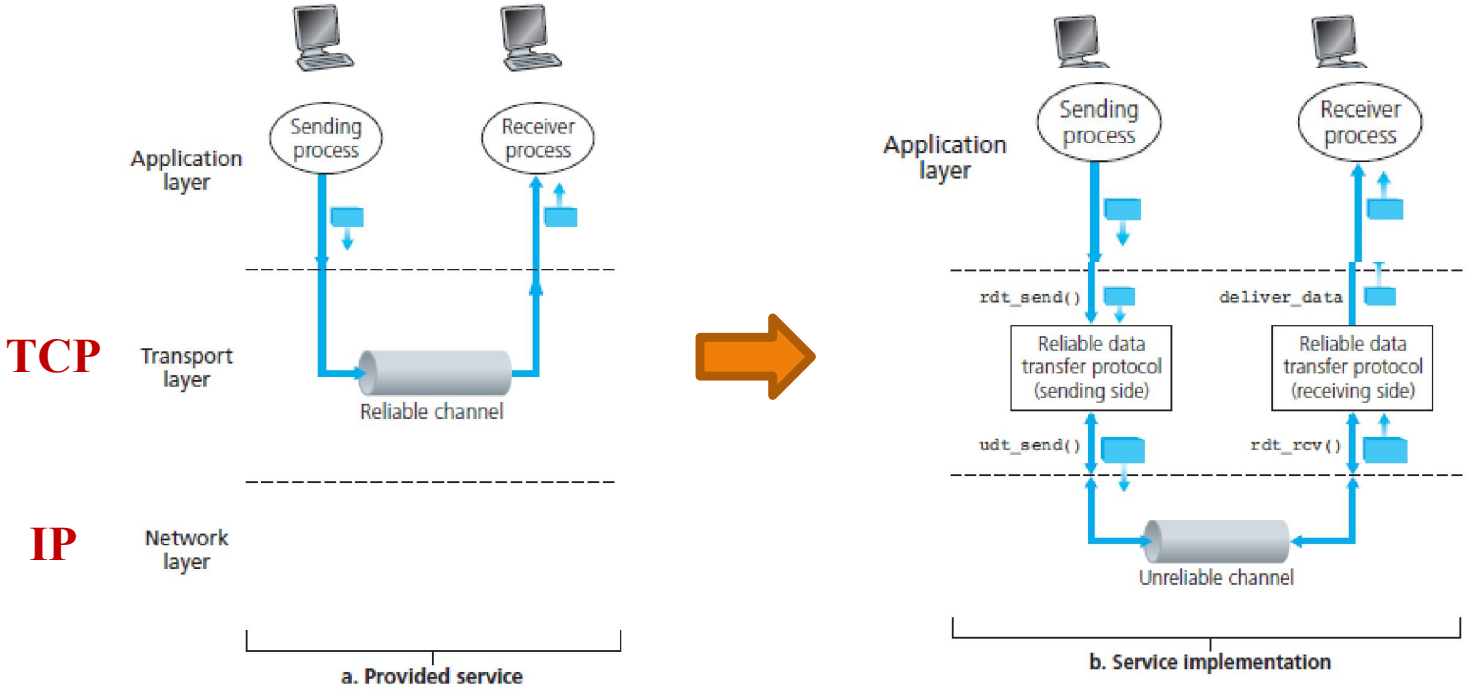
Example: Circular Buffer

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Ref: Circular Buffer



Reliable (TCP) using Unreliable (IP) Protocol

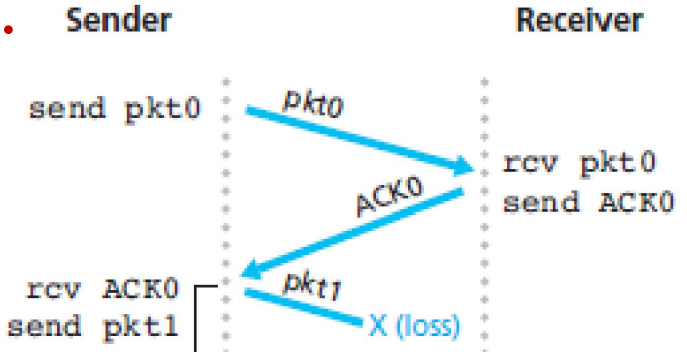


- "**Service provided**" by the TCP to the application layer above is to give a reliable channel between the two ends, using the unreliable protocol underneath (IP).
- It is the **responsibility** of a **reliable data transfer protocol (TCP)** to **implement** this **service abstraction**.

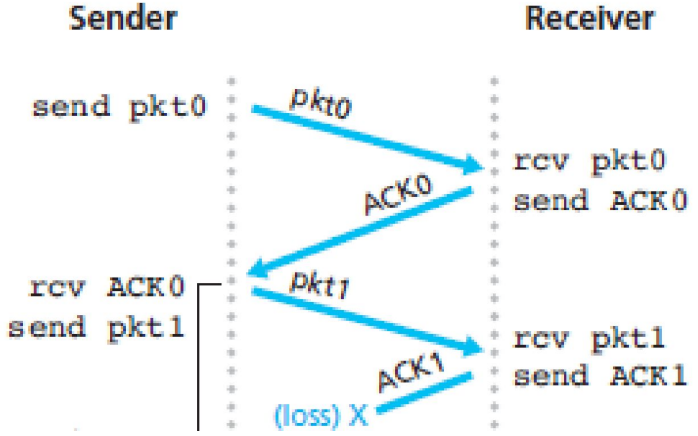
Issues with Unreliable Channel

1. Packet loss (data)
2. Packet loss (ACK)
3. Long delay in packet delivery

1.

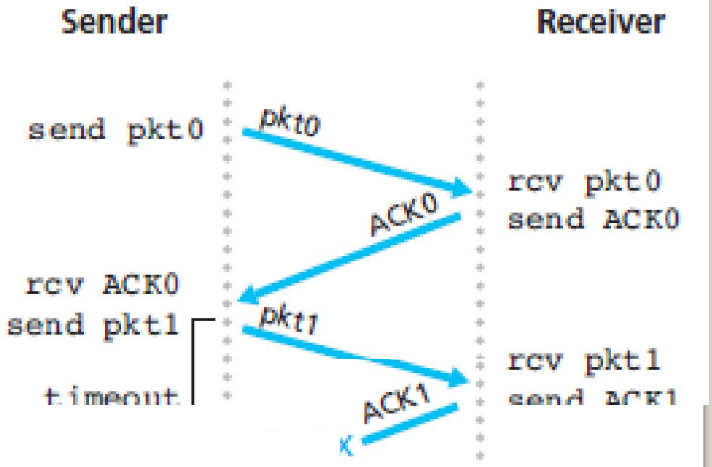


2.



Long delay in Receiving the ACK from Rx

3.



Q1: These issues are with which protocol layer?

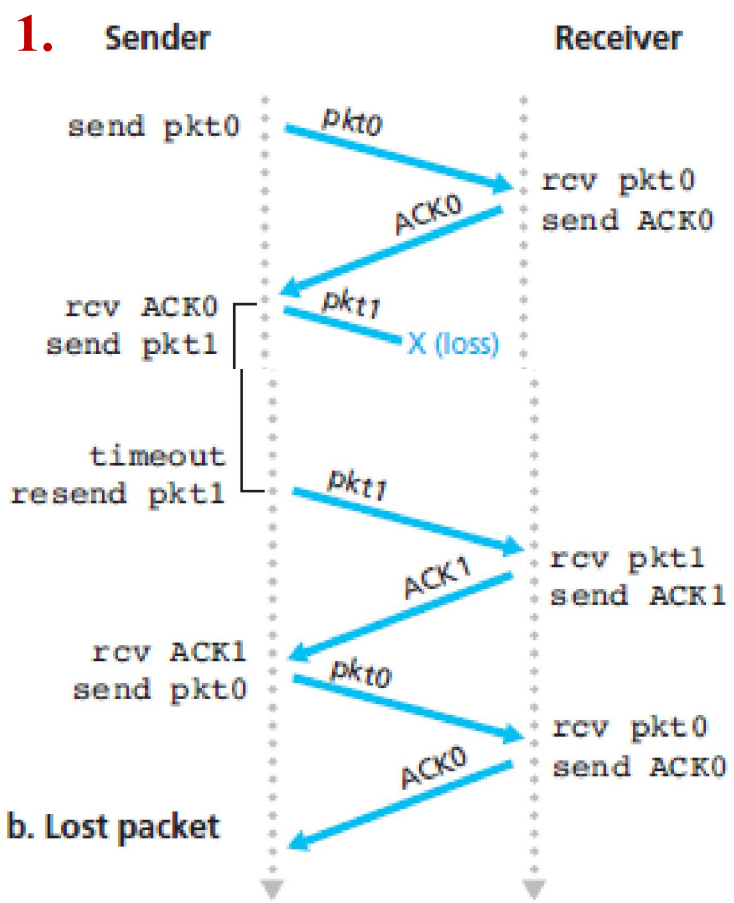
ANS: IP (Network Layer)

What are these above **diagrams** called as?

Protocol Sequence diagrams or Event Diagrams

TCP Handling: 1. Packet loss (data)

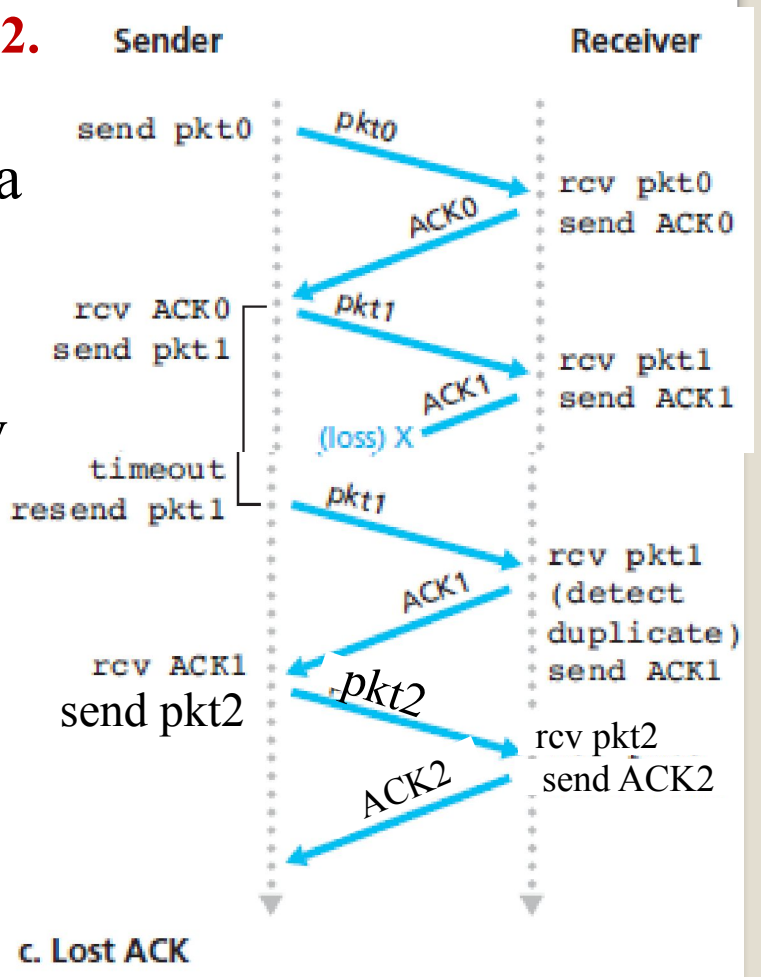
- Sender (TCP) starts a timer for each TCP segment sent over the network
- If an ACK for that particular segment is not received and the timer expires, the sender sends the same TCP segment again
- If the ACK is received for a particular single segment or a collective ACK for a set of segments, the relevant timers are switched off, and thus no timeout happens, and so, no re-transmission of that segment as well.



TCP Handling: 2. Packet loss (ACK)

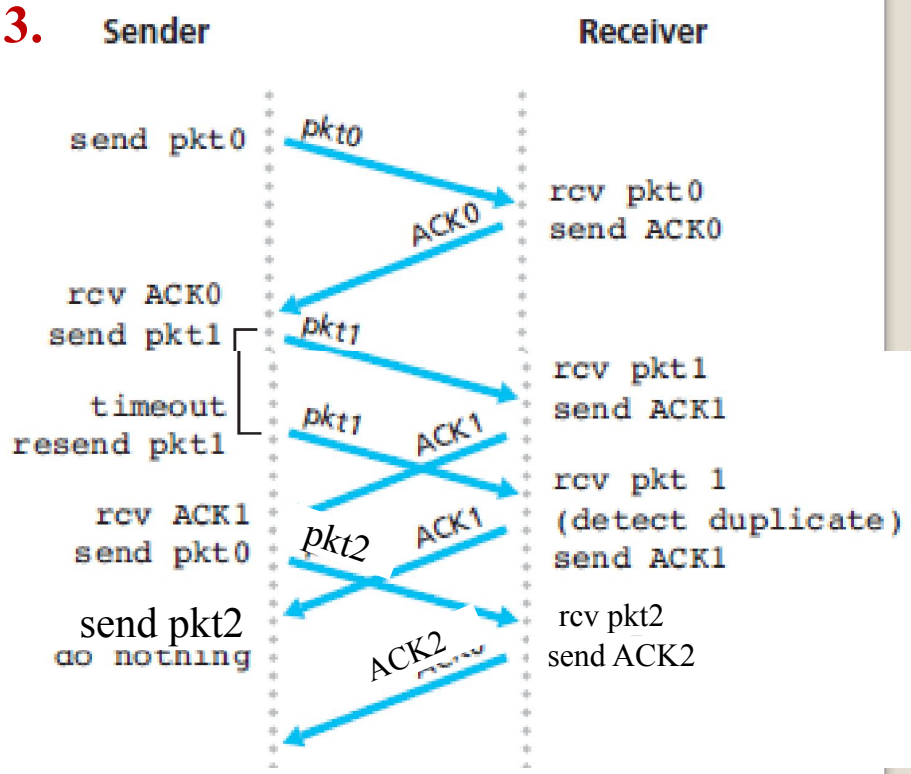
- When the sender does not receive the ACK for a data segment, the sender has no way of finding whether the data segment got lost or the ACK for that data segment got lost.
- Similarly, the end which is sending the ACK has no way of finding whether the ACK that was sent by it was received by the other end or not.
- So, the only way to resolve this issues is by sending the TCP segment again.
- When the receiver gets a duplicate segment, it understands that it's earlier ACK was not received by the sender and
- And thus, it again acknowledges the duplicate segment, though it has that data already in its Rx buffer.

2.



TCP Handling: 3. Long delay Packet delivery

- When the ACK for a data segment sent by the receiver takes more time to reach the sender, the sender assumes that either the data segment sent was lost or the ACK for the segment from the receiver was lost.
- The only way to resolve this issue is for the sender to re-send the same segment again, after the timeout.
- Now, the delayed ACK gets received, thus the sender proceeds with the sending of next data.



Note: This will cause duplicate ACKs to be received, in this case the sender is receiving two ACK1s. But the duplicate ACKs are ignored.

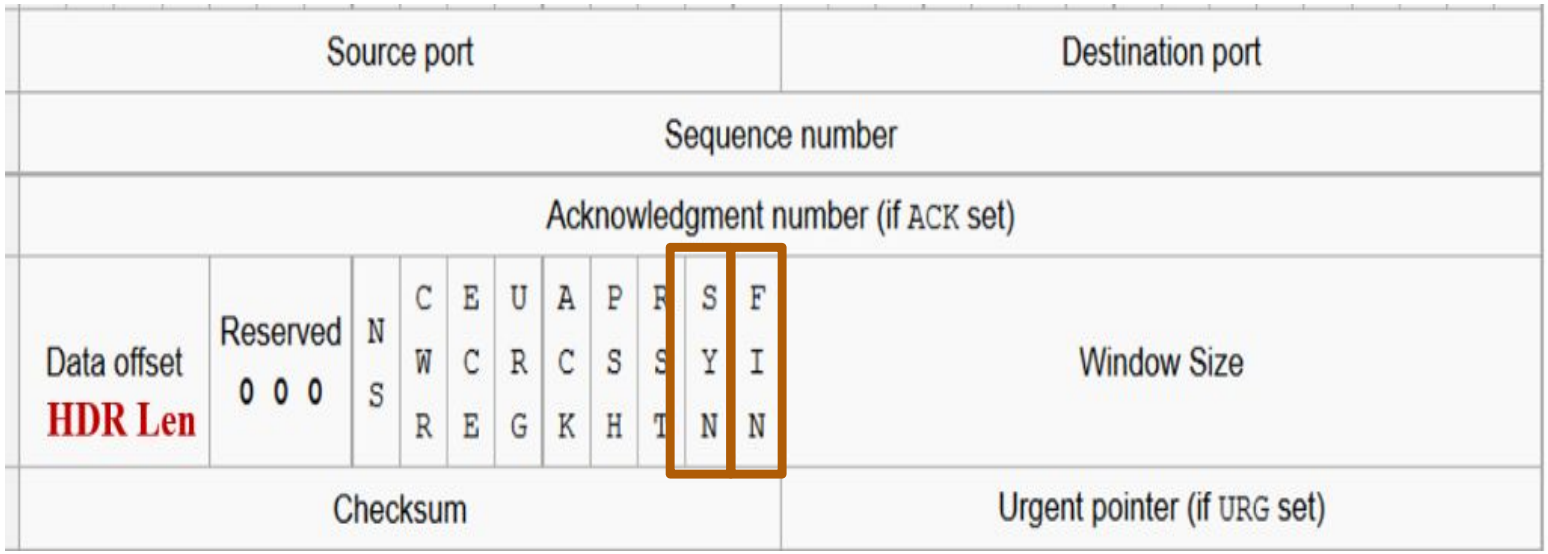
TCP Connection Establishment

Reference: Ref1: TCP/IP Illustrated-Volume 1:
Chapter 13: TCP Connection Establishment and Termination

TCP: Connection Management

- Let us now take a detailed look at what a TCP connection is, how it is established, and how it is terminated.
- Recall that TCP's service model is a **byte stream**.
- TCP detects and repairs essentially all the data transfer problems that may be introduced by packet loss, duplication, or errors at the IP layer below.
- Because of its management of ***connection state*** (information about the connection kept by both endpoints), TCP is a considerably more complicated protocol than UDP.
  - UDP is a *connectionless* protocol that involves no connection establishment or termination.
- In TCP, during connection establishment, several *options* can be exchanged between the two endpoints regarding the parameters of the connection. (Example: Window size, Maximum Segment size, etc.)

Connection Establishment: Control Bits



- When a **new connection** is being **established**, the **SYN bit** field is **turned on** in the **first segment** sent from **client** to **server**.
  - Such segments are called **SYN segments**, or simply **SYNs.**
- The **FIN flag** indicates the **end of data transmission** to **finish** a **TCP connection**.
  - The sender of the segment is finished sending data to its peer

SYN: It is a **short-form** of **Synch** or **Synchronization**

FIN: It is a **short-form** of the **Final segment** of a **TCP connection.**

Quiz 1: SYN and FIN Flags

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

What should the TCP/IP stack do when it receives a TCP segment with both SYN and FIN flags set from the network?

A. Consider it as a new connection request and termination of the last existing connection.

B. Consider only the SYN flag as valid and ignore FIN flag.

C. Consider only the FIN flag as valid and ignore SYN flag.

D. Drop the segment and take no action, since having both SYN and FIN flags set is an invalid condition.

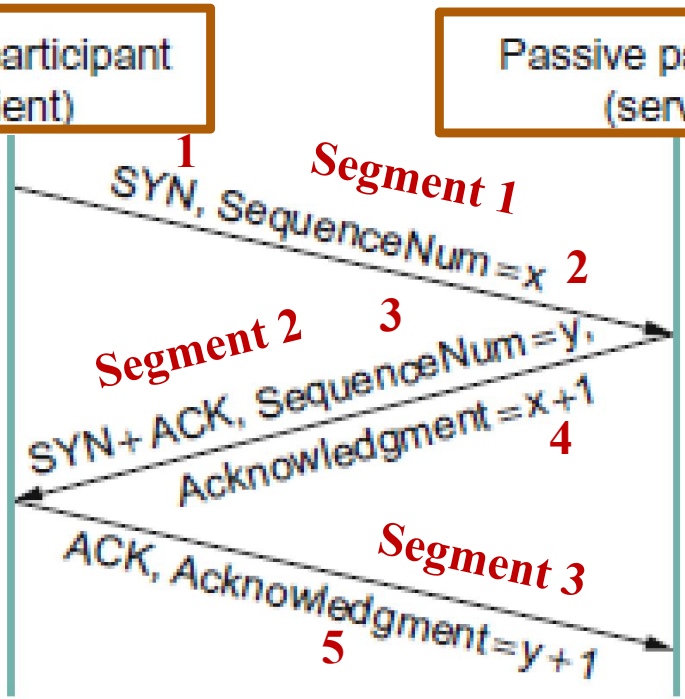
ANS: D

Note: Their **purposes** are **mutually exclusive**. A TCP header with **both** the **SYN** and **FIN flags set** is **anomalous TCP behavior**, causing various responses from the recipient, depending on the particular implementation of TCP/IP stack in an OS.

Interesting note :An attacker can send a segment with both flags set to see what kind of system reply is returned and thereby determine what kind of OS is on the receiving end. The attacker can then use any known system vulnerabilities for further attacks For example. **JunOS** checks if the SYN and FIN flags are set in TCP headers. If it discovers such a header, it drops the packet.

JunOS: It is an **OS** from **Juniper Networks**, running on Network equipment from Juniper.

TCP Connection Set-Up: Timeline



Active Participant: Who **initiates** a TCP connection

Passive Participant: Who **accepts** the TCP connection

Note: Normally **Servers** wait for **TCP connections** to be **initiated** by **multiple clients**.

1. SYN packet from **Client** with **SYN bit set**.

2. Seq no.(x): **Initial Sequential Number** from the **Client**

3. Seq no. (y): **Initial Sequential Number** from the **Server**

4a. ACK Acknowledgement from the Server with ACK num = x + 1

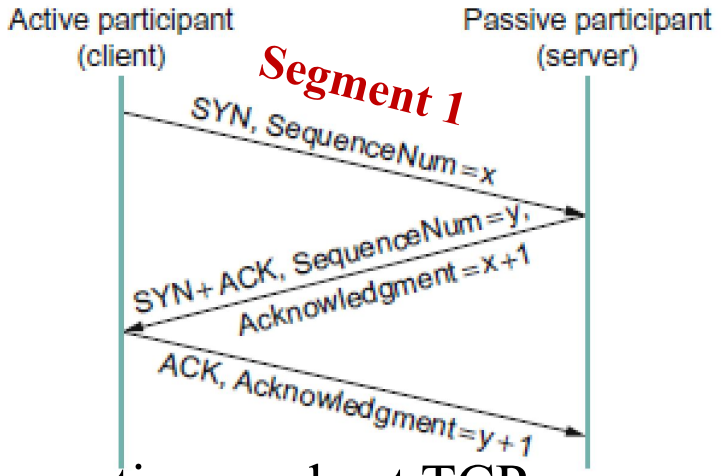
4b. Which means that **SYN** packet is considered to **consume** one byte of data.

5. ACK Acknowledgement from the **Client** with **ACK num = y + 1**

Note: ACK does not consume a Sequence number. It is not required to retransmit an ACK segment if it is lost or not delivered to the intendent recipient.

TCP Connection Establishment

- A connection typically goes through **three phases**:
  1. Setup,
  2. Data transfer (called *established*), an
  3. Teardown (closing).

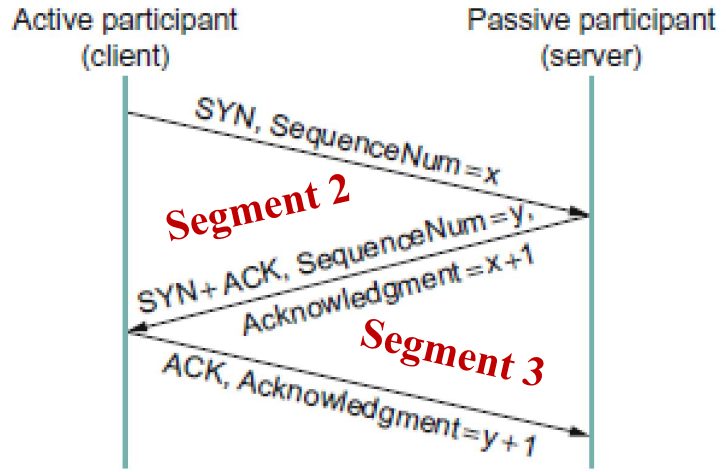


- As we will see, some of the difficulty in creating a robust TCP implementation is handling all of the transitions between and among these phases correctly.
- To establish a TCP connection, the following events usually take place
- 1. The ***active opener*** (normally called the client) sends a SYN **segment 1** (i.e., a TCP/IP packet with the *SYN* bit field turned ON in the TCP header) specifying the port number of the peer to which it wants to connect and the client's **Initial Sequence Number** is **(x)**

TCP Connection Establishment: 2 and 3

2. The server responds with its own SYN segment containing its initial sequence number (**ISN(y)**).

This is the **segment 2**. The server also acknowledges the client's SYN by **ACK'ing ISN(x) plus 1**.



Note that a **SYN consumes one sequence number** and is **retransmitted** if **lost**.

3. The client acknowledges **SYN** from the server by **ACK'ing ISN(y) plus 1**. This is the **segment 3**.

- These three segments complete the connection establishment.
- This is often called the ***three-way handshake***.
- Its main purposes are to let each end of the connection know that a **connection is starting** and the special details that may be carried as **options**, and to **exchange** the **ISNs** of **both ends**.

About Sequence Number and ACK Number

- All bytes in a TCP connection are numbered, beginning at a randomly chosen initial sequence number (ISN).
- The SYN packets consume one sequence number, so actual data will begin at ISN+1.
- The acknowledgement number is the sequence number of the next byte the receiver expects to receive.
- The receiver **ack'ing** sequence number ‘n’ acknowledges receipt of all data bytes less than (but not including) byte number ‘n’.
- The sequence number in the header is always valid, not related to any other control bits in the header.
- Whereas, the acknowledgement number is valid only when the ACK flag is set to one.
- The only time the ACK flag is not set, that is, the only time there is not a valid acknowledgement number in the TCP header, is when the first packet of connection set-up, i.e, SYN packet is sent out.

Quiz 2: Amount of Data Exchanged

Choose the correct option about the total number of bytes exchanged through a socket connection so far from one end to the other, if the segment is having a sequence number currently as **x**.

A. **x** bytes have been sent by the end which is generating this segment.

B. **x+1** bytes have been sent by the end which is generating this segment.

C. **x-1** bytes have been sent by the end which is generating this segment.

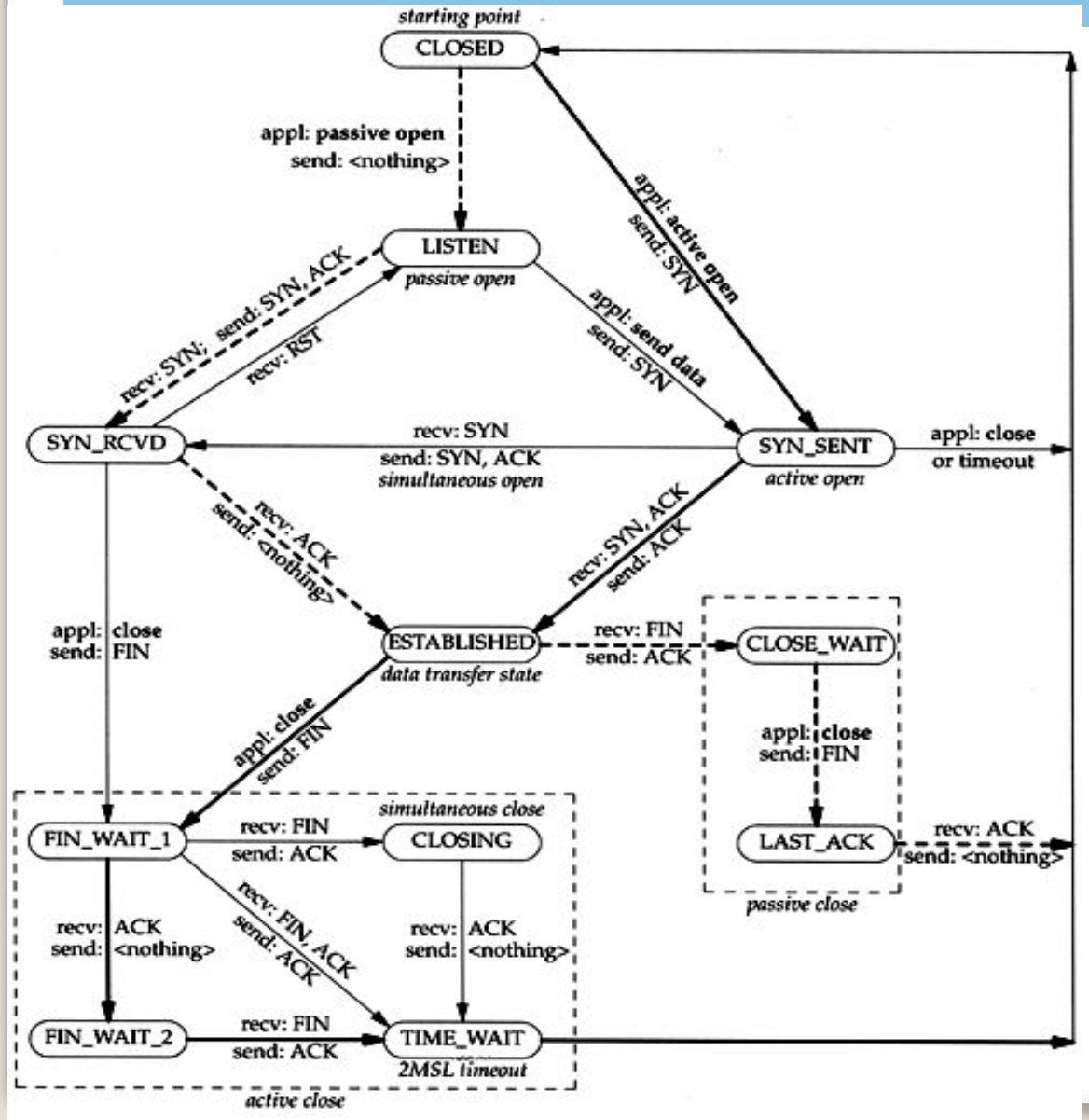
D. Nothing can be said about the number of bytes have been exchanged based on the value of sequence number in the segment.

ANS: D

Nothing can be said about the number of bytes exchanged because of the following reasons.

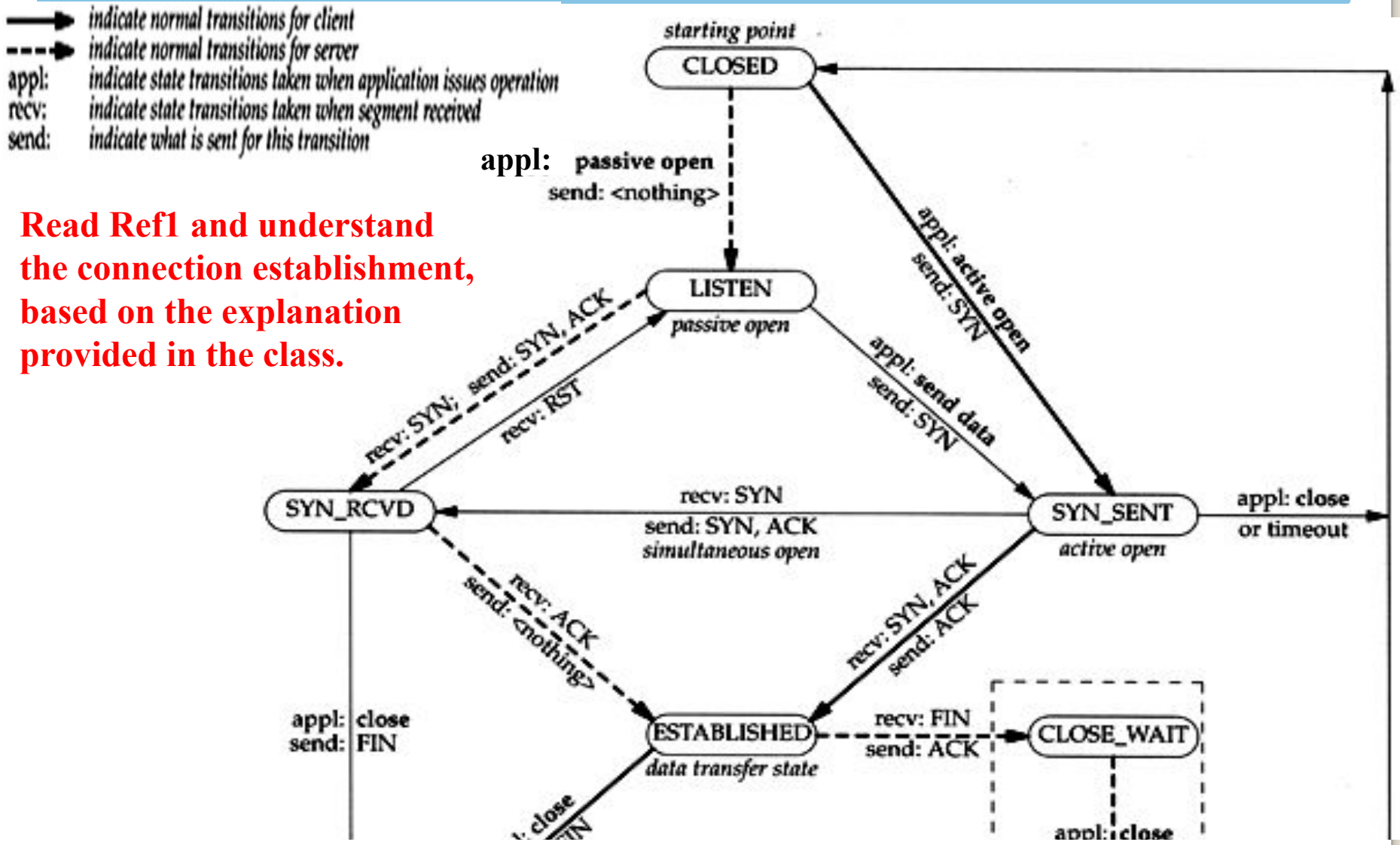
1. The value of **x** does not reveal any information of what was it is ISN (Initial Sequential Number) chosen by the sender.
2. If the ISN is also known, is it possible to find the number of bytes exchanged so far?
   - No, because the sequence number could have wrapped around the max value.

TCP: State Transition Diagram



indicate normal transitions for client
indicate normal transitions for server
appl: *indicate state transitions taken when application issues operation*
recv: *indicate state transitions taken when segment received*
send: *indicate what is sent for this transition*

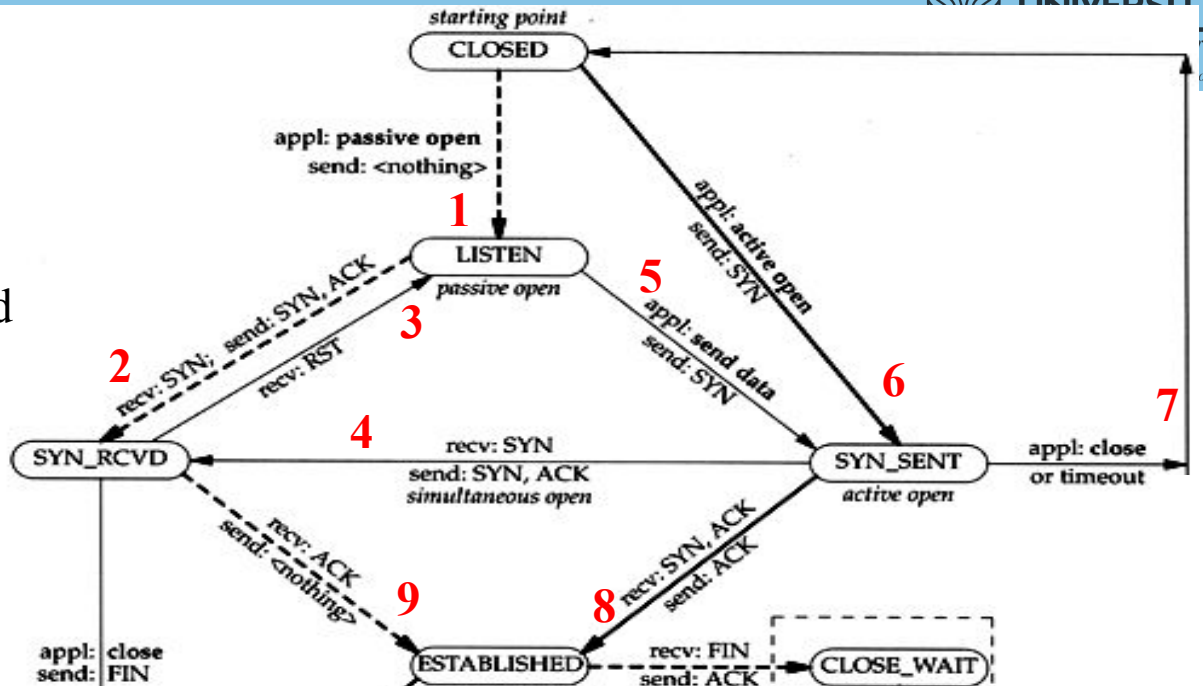
TCP: Partial State Transition Diagram Shown (connection Setup)



Read Ref1 and understand the connection establishment, based on the explanation provided in the class.

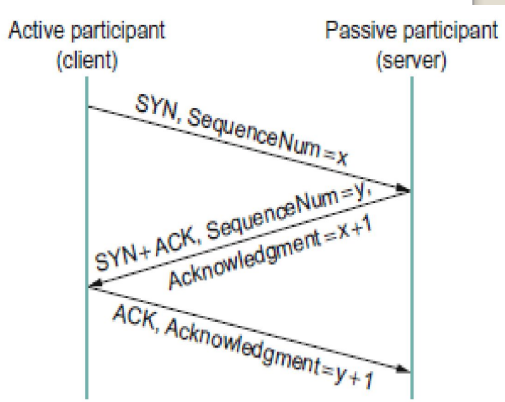
Quiz 1: TCP Connection Establishment– Recap

Abbreviations:
Appl: Application
Reqs: Requests
Rx: Receiving/Received
Tx: Transmitting



- Match the events causing the transitions (**fill in numbers**) shown above.

Event	Transition Number	Event	Transition Number
Appl initiating a connection Req	6	After Rx SYN+ACK from the Server	8
Appl initiating to wait for connection Reqs	1	After Rx ACK from the Client	9
After Rx SYN from the Client	2	After Tx SYN, if SYN from the other end Rx	4

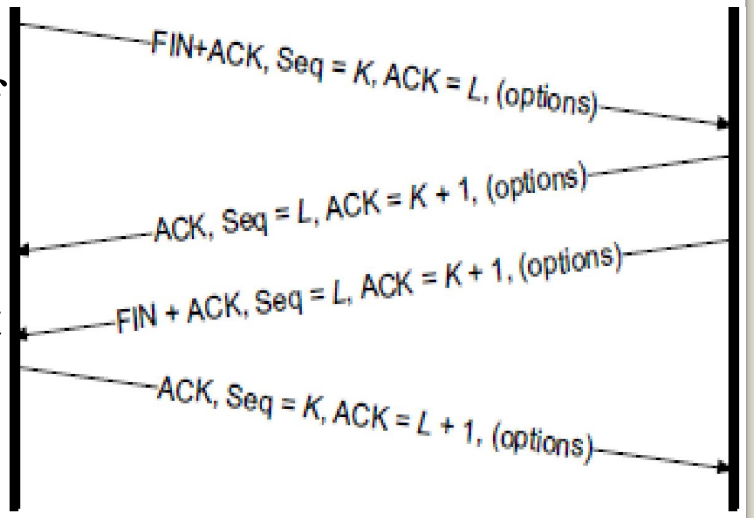


TCP Connection Termination

Reference: Ref1: TCP/IP Illustrated-Volume 1:
Chapter 13: TCP Connection Establishment and Termination

TCP: Connection Termination

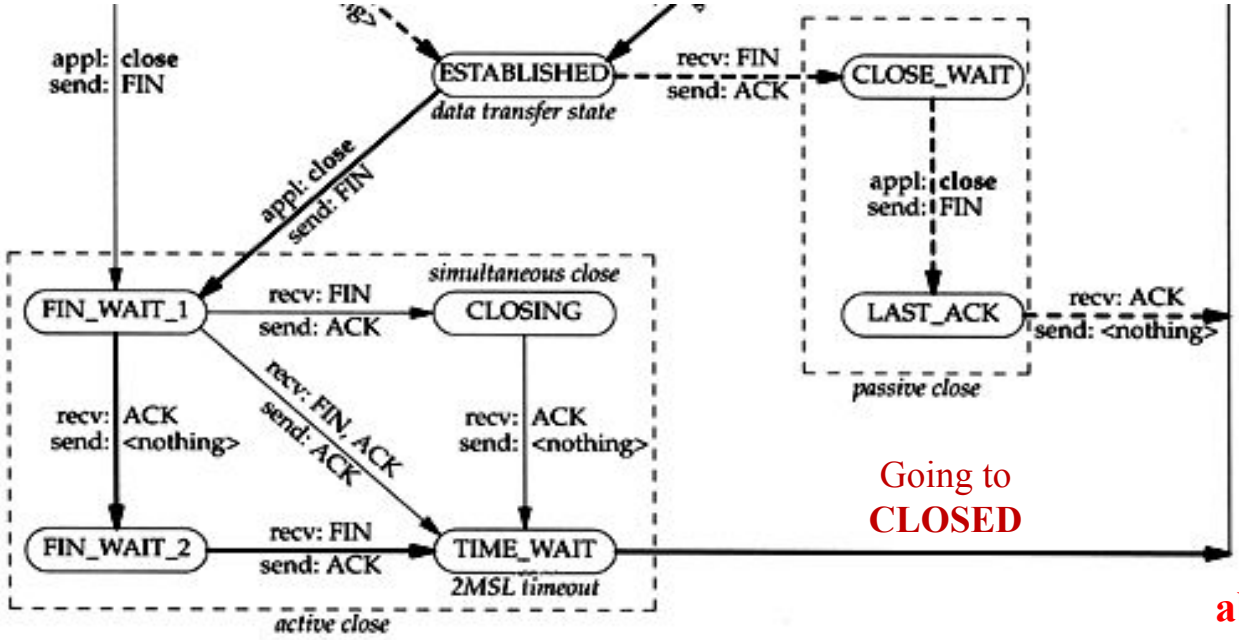
- The application process on both sides of the connection must independently close its half of (or part of) the connection.
- If only one side closes the connection, then this means it has no more data to send, but it is still available to receive data from the other side.



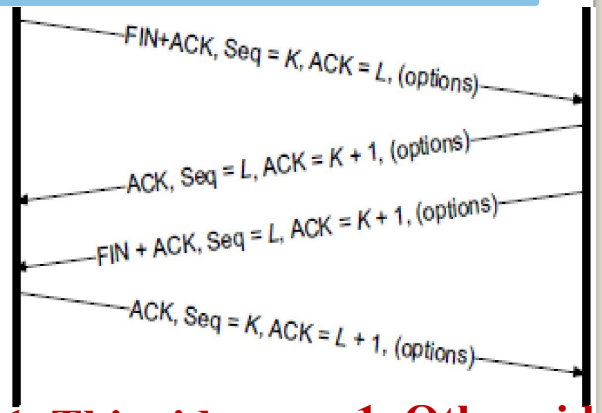
- This complicates the state-transition diagram because it must account for the possibility that the two sides invoke the close operator at the same time,
- As well as the possibility that first one side invokes close and then, at some later time, the other side invokes close.
- Thus, on any one side there are **three combinations** of **transitions** that get a connection from the ESTABLISHED state to the CLOSED state:

TCP: Connection Termination (3 Combinations)

Note: ACK is ignored if it is received for the same ACK no. which was already ACK'ed.



Going to **CLOSED**



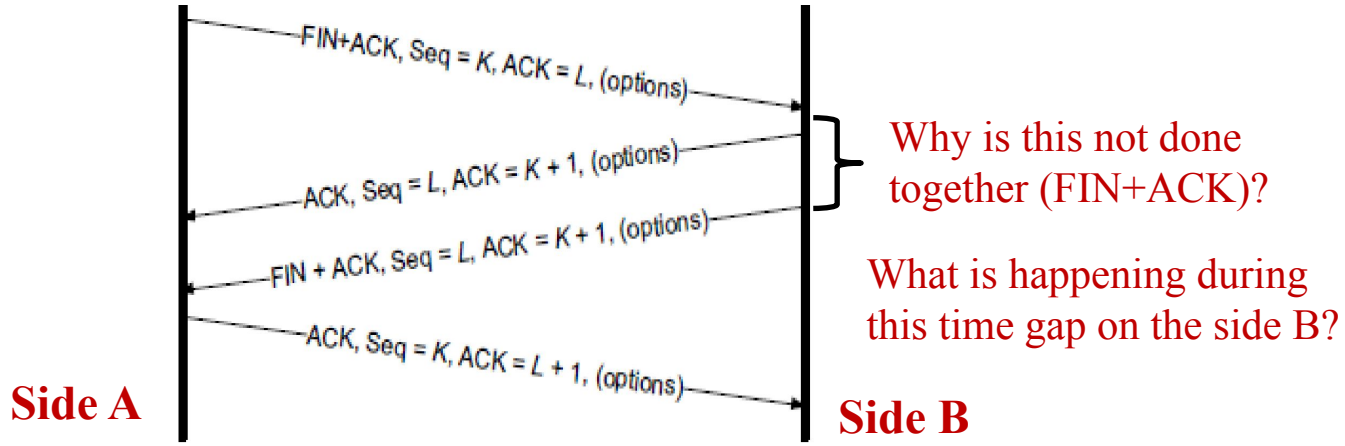
1. This side **1. Other side**

2. Other side **2. This side**

Trace state transitions due to the above msg exchanges for "this side"

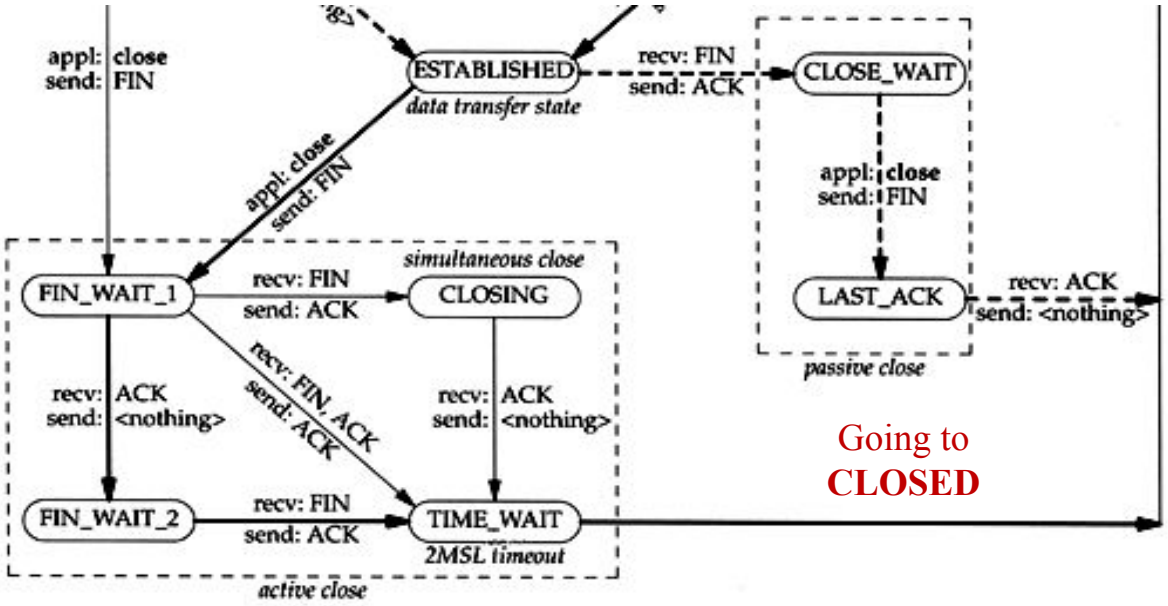
1. **This side closes first**: ESTABLISHED → FIN_WAIT_1 → FIN_WAIT_2 → TIME_WAIT → CLOSED.
   - Follow the state transitions in the state transition diagram above, along with the flow of messages, assuming "**This side**" on the **left**
2. **The other side closes first**: ESTABLISHED → CLOSE_WAIT → LAST_ACK → CLOSED.
   - Follow the state transitions in the state transition diagram above, along with the flow of messages, assuming "**Other side**" on the **left**

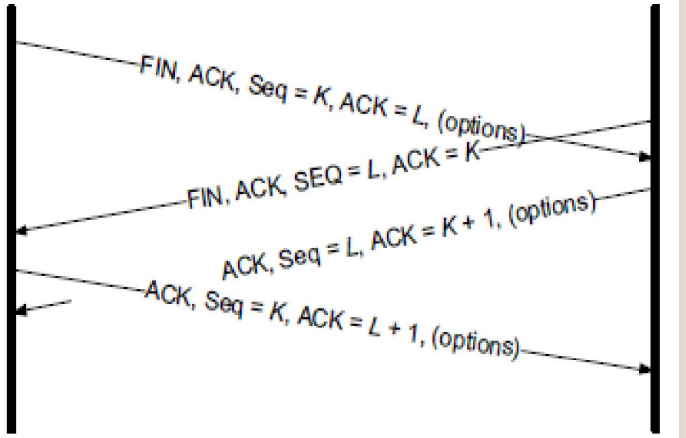
Quiz 2: Why is there a gap?



- What does Side B understand on receiving FIN+ACK from Side A?
- Application on Side A has sent a **Close** request to the TCP/IP stack, because it does not have any data to send to Side B and it wants to close the connection from its end.
- After sending ACK for the FIN from Side A, what does Side B do?
- Side B informs its application that Side A wants to close its part of the connection. The application on Side B has the liberty to decide either to keep its side of the connection and keep sending data, if it has, to Side A, or decide to close its connection as well, by sending a Close command to the protocol Stack.
- When does Side B send FIN+ACK to Side A?
- On receiving a Close command from the Application. Note that protocol stack has no authority to close the connection without the consent of the application ☺

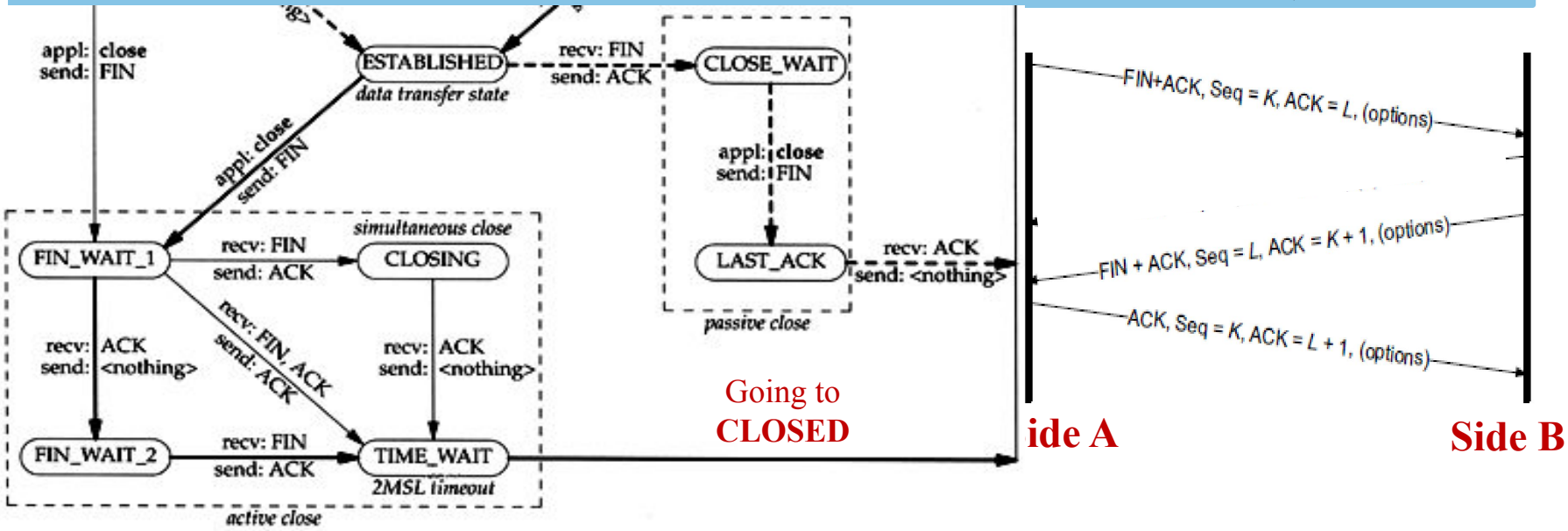
TCP: Connection Termination (3rd Combination: Simultaneous Close)





3. **Both sides close at the same time:** ESTABLISHED → FIN_WAIT_1 → CLOSING → TIME_WAIT → CLOSED.
   - Follow the state transitions in the state transition diagram above, along with the flow of messages shown above for simultaneous close
4. **There is one more combination possible, which one?** ☺

TCP: Connection Termination (4th Combination: FIN+ACK from Other side)

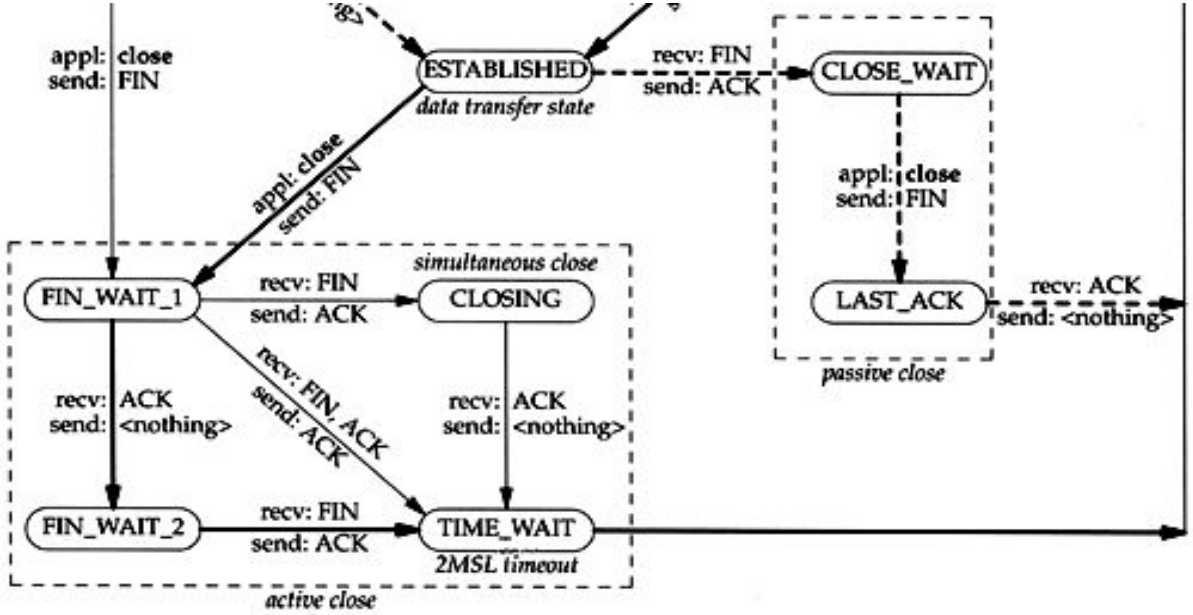


4. **Side A closes first and Side B also closes before sending ACK for FIN:** ESTABLISHED → FIN_WAIT_1 → TIME_WAIT → CLOSED.
   - Follow the state transitions in the state transition diagram above on Side A, along with the flow of messages shown above for the fourth combination, where FIN+ACK is sent from the other side B.

- This combination is possible when the FIN is received from Side A, and the application on Side B also decides to close the connection before the ACK for the FIN is sent from Side B, then Side B sends FIN+ACK together to Side A

TCP Connection Termination: TIME WAIT

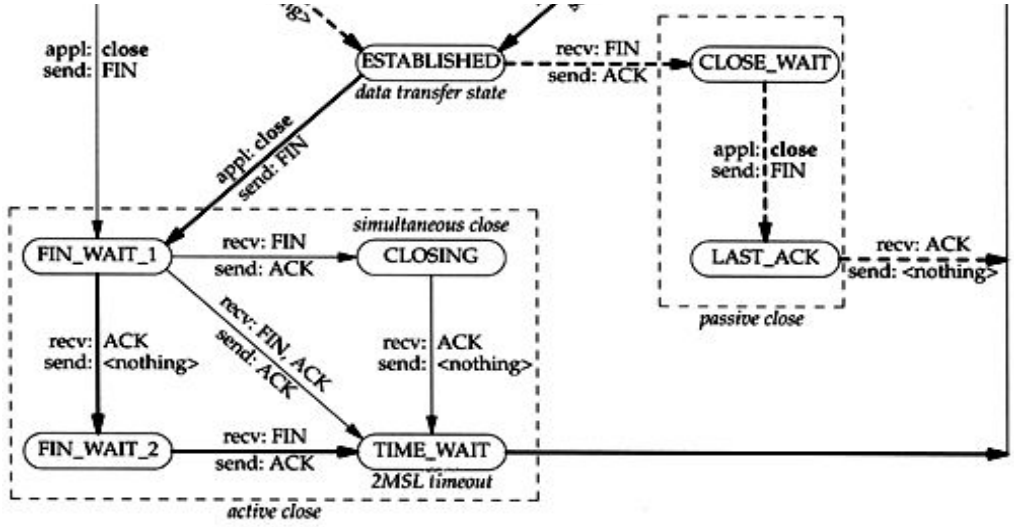
Note: 120 seconds is configurable based on the current behavior of Internet

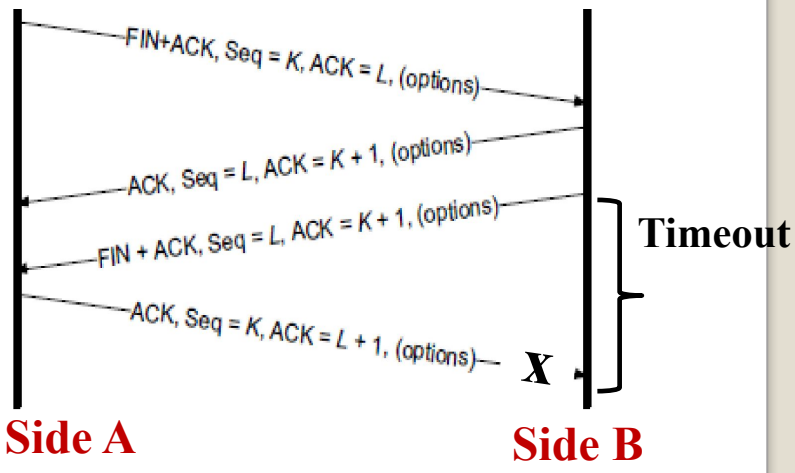


- The main thing to recognize about connection teardown is that a connection in the **TIME WAIT** state cannot move to the **CLOSED** state until it has waited for **two times** the **maximum** amount of **time** an **IP datagram might live** in the **Internet** (i.e., ~**120 seconds**).
- The reason for this is that, while the local side of the connection has sent an ACK in response to the other side's FIN segment, it does not know that the ACK was successfully delivered or not.

TCP Connection Termination: TIME_WAIT

Contd…





- As a consequence, the other side might retransmit its FIN segment, and this second FIN segment might be delayed in the network.
- If the connection were allowed to move directly to the CLOSED state, then another pair of application processes might come along and open the same connection (i.e., use the same pair of port numbers), and the delayed FIN segment from the **earlier incarnation** of the connection would immediately initiate the termination of the **later incarnation** of that connection.

TCP Connection Termination

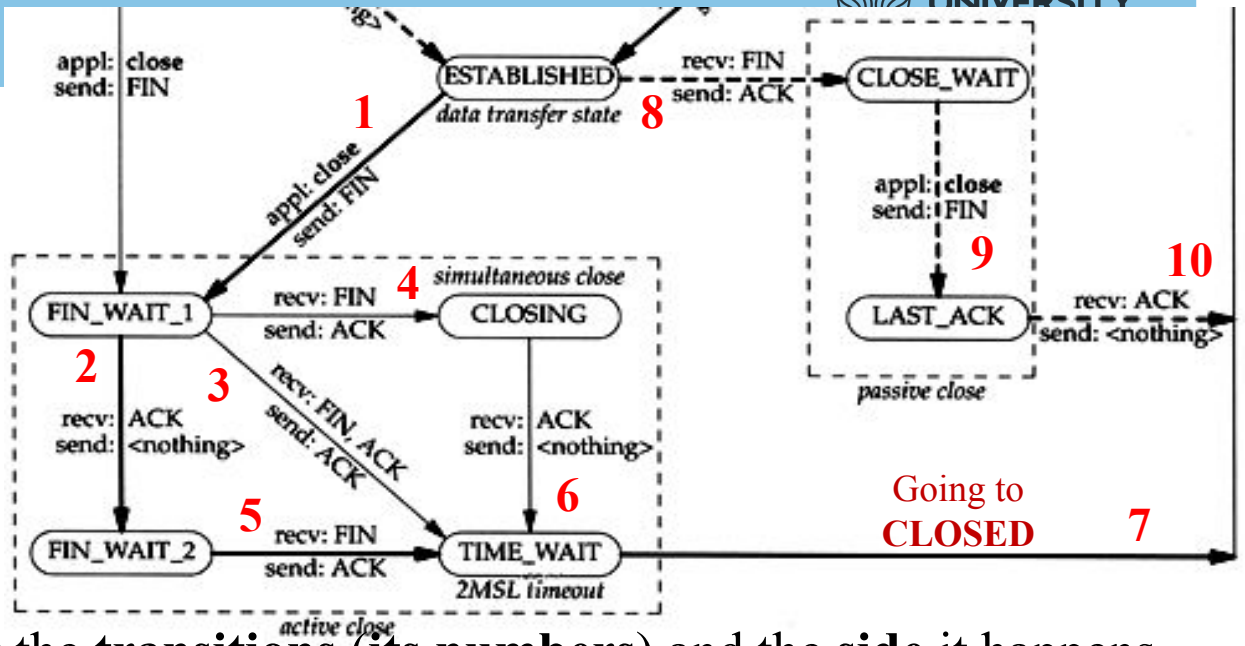
Quiz 2 to 4

Quiz 2: TCP Connection Termination

Abbreviation:
Appl: Application

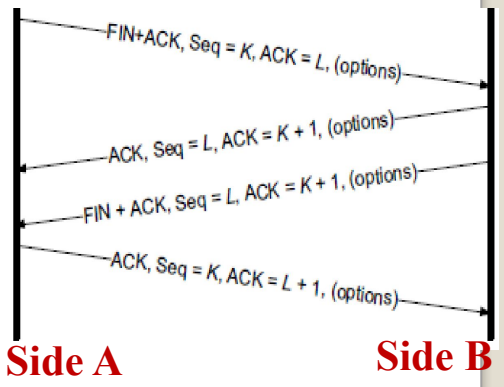
Note: Consider State transitions of both **Side A** and **Side B**.

Assume there was a TCP connection between **Side A** and **Side B** already.



- Match the events causing the **transitions** (**its numbers**) and the **side** it happens.

Event	Side	Num
The **transition** that would **trigger** a **close** notification to **appl**.	B	8
The **transition** due to the **appl** closing **later**	B	9
The **transition** that takes a **side** to **TIME_WAIT**	A	5
The **transition** that takes a **side** to **CLOSED** without **TIME_WAIT**.	B	10
The **transition** triggered on the **side,** which receives **ACK last**	B	10
The **transition** that takes a **side** to **CLOSED** after **TIME_WAIT**	A	7



Message transactions are shown here,

Quiz 3: TIME_WAIT

- Why is there a **TIME_WAIT** state **only** on the **active close** side of the transition and **not on both sides** and why is it on the **Active close** side and **not passive close**? Choose **all the correct options**.

A. It is not possible to make both sides to wait.

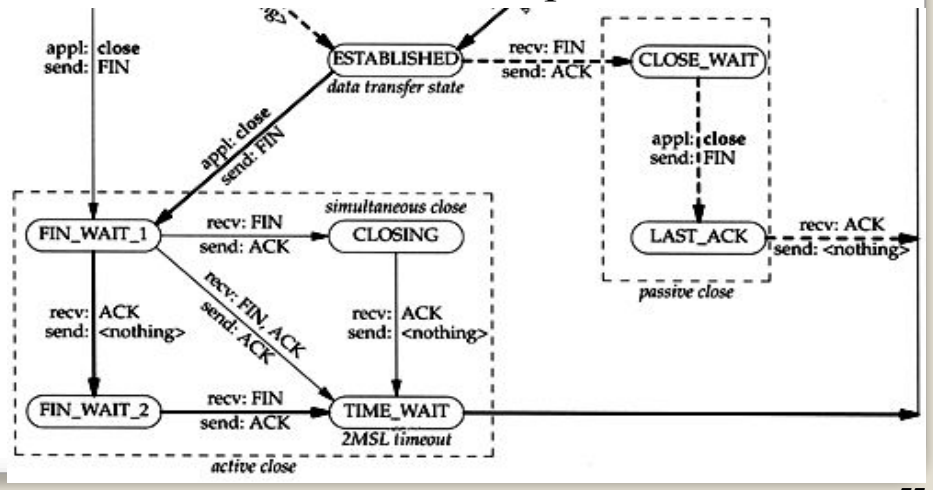
B. It is sufficient to make only one side to wait because it is done to make sure a new connection between the same set of machines and the ports happens only after a fixed delay (2 MSL)

C. It is not correct to make both sides to wait because it won't be possible to establish a connection between the same set of machines and ports later on.

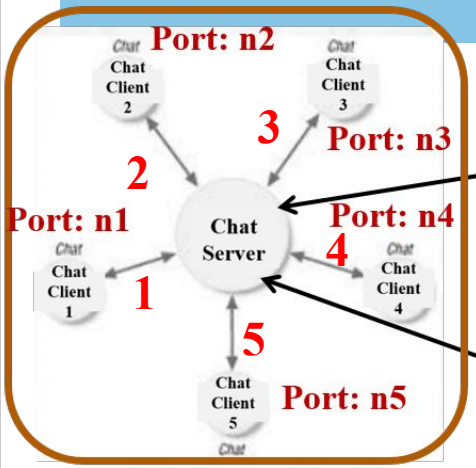
D. Though it is sufficient to introduce delay on any side, active side is chosen because that is most likely to initiate a new connection with the same machine and the port later on.

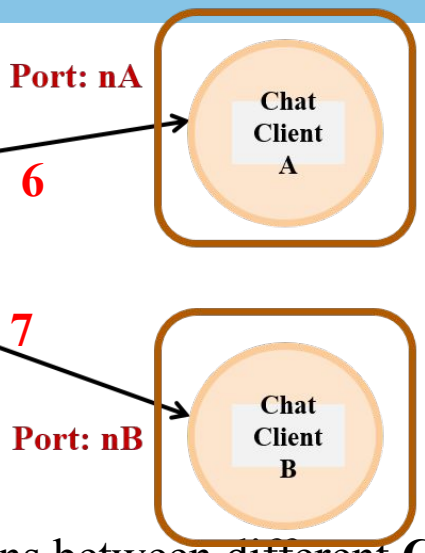
ANS: B and D

MSL: Maximum Segment Lifetime



Quiz 4: Multi-chat Application





Note: IP and port nums are given here.

Let us number each connection from 1 to 7.

Note: Recall the **Server Port ID** was: **9001**

Let us call **loopback IP address 127.0.0.1** as **LBIP**

Note: The OS takes care of allotting unique Client ports.

- The current connections between different **ChatClients** and the **ChatServer** are shown above. Answer the questions and fill in the relevant values below.

1. **How many unique sockets are created here?** ANS1: 7

2. **How many of the connections are in loopback mode?** ANS2: 5 (1 to 5)

3. **Fill in the values of each socket values of the connections in the table:**

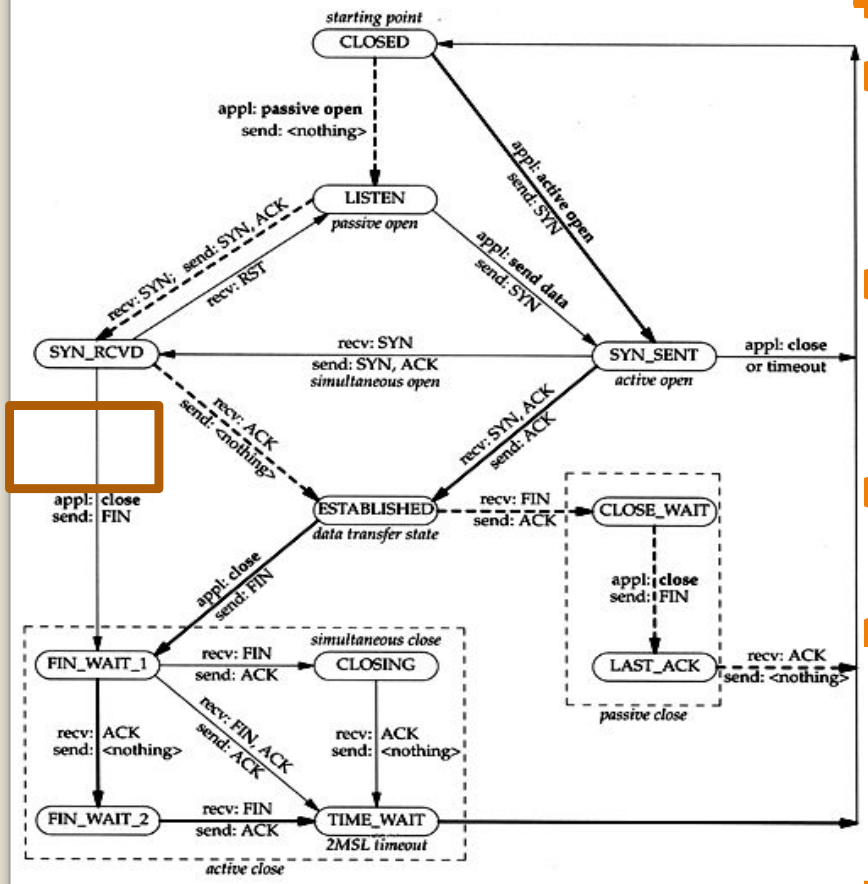
Note: You can notice that for all the **sockets 4-tuple values** are **unique**, with the **server**.

Connections	Server IP	Server Port	Client IP	Client Port
1	LBIP	9001	LBIP	n1
2	LBIP	9001	LBIP	n2
3	LBIP	9001	LBIP	n3
4	LBIP	9001	LBIP	n4
5	LBIP	9001	LBIP	n5
6	IP1	9001	IPA	nA
7	IP1	9001	IPB	nB

Some more Concepts
Based on TCP State Diagram

Quiz 1: Reason for this Transition

Appl: close
Send: FIN

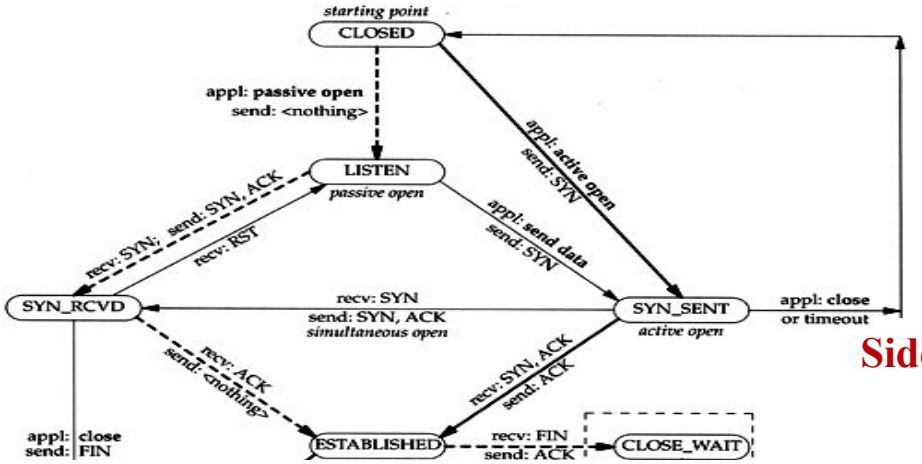


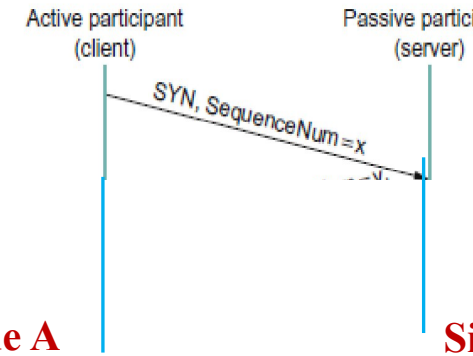
- **Identify what causes this transition.**
- The server application after accepting a connection request from a Client informs the application on its side.
- The Server application can decide to accept or refuse the connection with the client by sending a close message to TCP.
- This will make the state to move from SYN_RCVD to FIN_WAIT1

1. **The potential reasons could be that the server application has reached its limit on number of clients it can be simultaneously connected to. Or**
2. **The server application has black listed the client initiating the request, for some other reasons, so does not want to establish a connection with the client.**

Half-open Condition

Ref: Half-open





Q: If SYN, ACK is not sent by Side B, in response to SYN from Side A, in which state they would be?

Message transactions are shown here

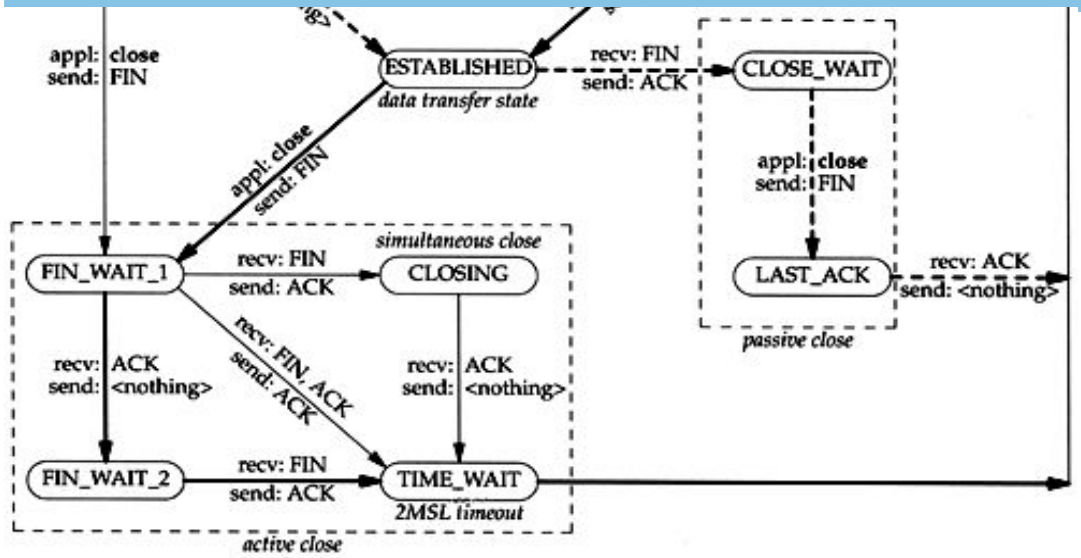
ANS:
Side A (Client): SYN_SENT
Side B (Server): SYN_RCVD

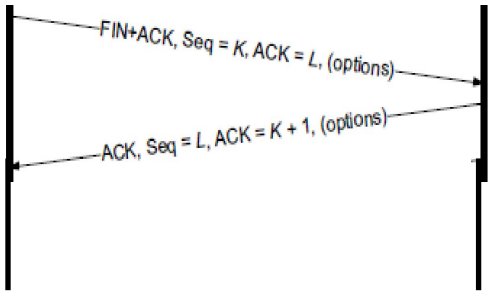
- What is a **Half-open** condition?

1. A connection which is in the process of **being established.** This is **also known as embryonic connection**. It is a **partially created** connection.
2. It **also refers** to TCP connections whose state is **out of synchronization** between the two communicating hosts, possibly due to a crash of one side.

- A TCP connection is referred to as ***half-open*** when the host at one end of that TCP connection has crashed, or has otherwise removed the socket without notifying the other end.
- If the remaining end is idle, the connection may remain in the half-open state for unbounded periods of time.

Half-close Condition





Side A
Active close

Side B
Passive close

Message transactions are shown here

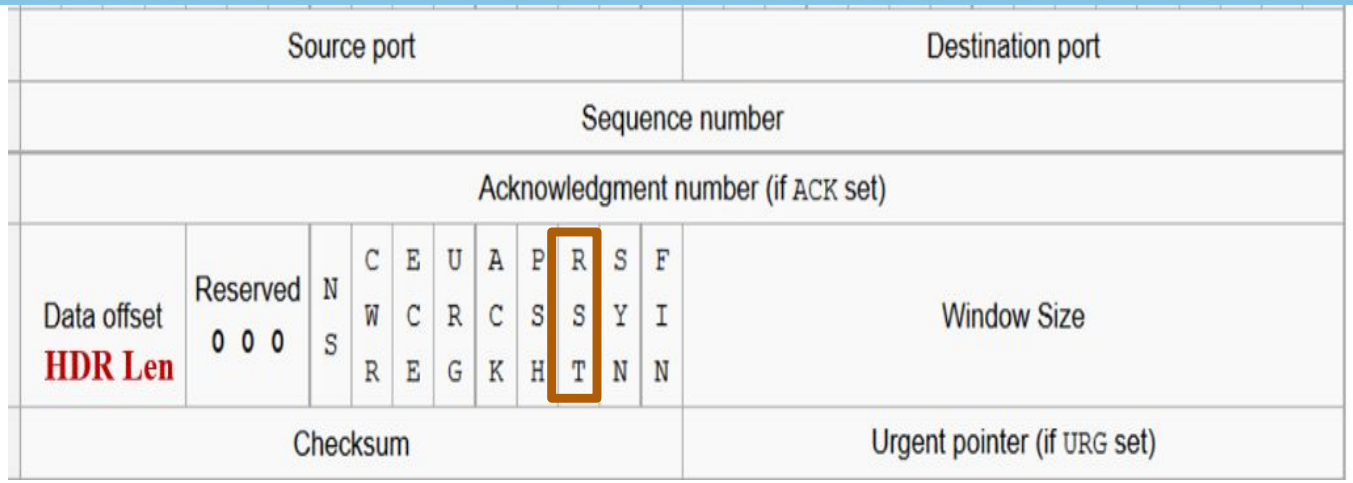
- What is a **Half-close** condition?
- Application on one side has sent a close message and in response to that the TCP has sent a FIN message to the other end and an ACK is also received from the peer
- Which host (Side A or B) could be in **half-close** condition here? **ANS: Both A & B**
- In which state Side A would be while it is in half-close condition?
  - FIN_WAIT_2 state after receiving the ACK from the other end for its FIN message. It is waiting for the other end to close the connection as well.
  - Application on Side B is yet to close the connection, it is in CLOSE_WAIT state.

Quiet Time Concept: While in TIME_WAIT

- 2MSL wait provides a protection against delayed segments from an earlier incarnation of a connection being interpreted as part of a new connection.
- What if a host with connections in the TIME_WAIT state crashes, reboots within the MSL, and immediately establishes new connections using the same local and foreign IP addresses and port numbers corresponding to the local connections that were in the TIME_WAIT state before the crash?
- In this scenario, delayed segments from the connections that existed before the crash can be misinterpreted as belonging to the new connections created after the reboot.
- To protect against this scenario, TCP should wait an **amount of time** equal to the **MSL** before creating any new connections after a reboot or crash. This is called the **quiet time**.

MSL: Maximum Segment Lifetime is the maximum time a TCP segment can exist in a network It is normally set to 2 minutes.

1. Reset: No process waiting on Destination Port



- A segment having this bit set to "ON" is called a "reset segment" or simply a "reset."
- Reset segments do not carry any data but its Sequence number and ACK number (with its ACK bits set) are valid.
- A common case for generating a reset segment is when a connection request arrives (SYN message from the Client) and **no process** is **listening** on the **destination port** at the *Server*.

2. Reset: Incorrect Segment Received

- In general, a **reset is also sent** by TCP whenever a **segment arrives** that **does not appear to be correct** for the **referenced connection**.
  - The term referenced connection means the connection specified by the **4-tuple** in the **TCP and IP headers** of the **segment**.
- Resets ordinarily result in a fast teardown of a TCP connection.
- How to **validate** a **Reset segment** received?
- For a reset segment to be accepted by a TCP, the ACK bit field must be set and the ACK Number field must be within the valid window of the sender of the Reset segment.
- This helps to prevent a simple attack in which any intruder who is able to generate a reset, matching the appropriate connection (4-tuple) could disrupt a connection in progress.

3. Reset: To Abort a Connection

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

- We know that a normal way to terminate a connection is for one side to send a FIN.
- This is sometimes called an **orderly release** because the FIN is sent after all previously queued data has been sent, and there is normally no loss of data.
- But it is also possible to **abort a connection** by **sending** a **reset instead of a FIN** at **any time**. (that is why it needs to be validated)
- This is sometimes called an **abortive release**.
- **Aborting a connection** provides **two features** to the **application**:

1. Any queued data is thrown away and a reset segment is sent immediately, and
2. The receiver of the reset can tell that the other end did an abort instead of a normal close to the application on its side.

Triggering Transmission

Reference: Ref1: Computer Networks A Systems Approach:
Section 5.2.5: Triggering Transmission

Triggering Transmission

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

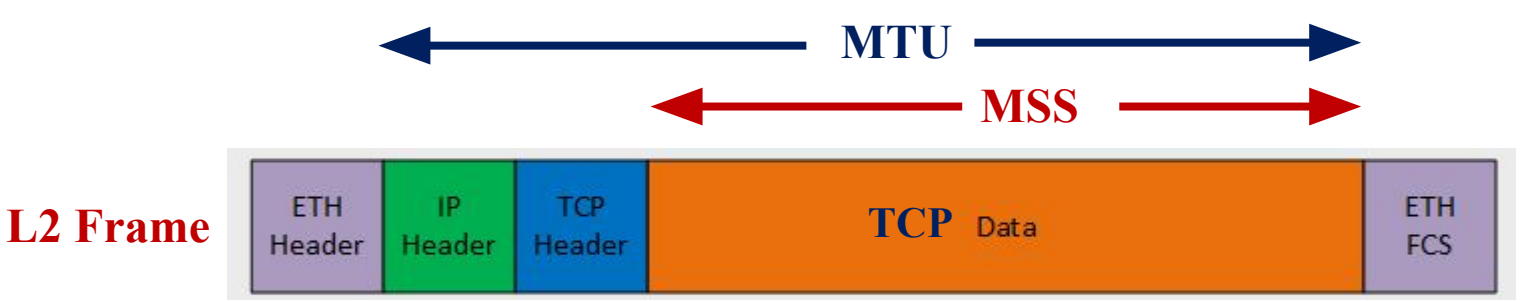
- We next consider a surprisingly **subtle** issue: how TCP decides to transmit a segment.
- As described earlier, TCP supports a byte-stream abstraction; that is, application programs write bytes into the stream, and it is up to TCP to decide that it has enough bytes to send a segment.
- What **factors** govern **this decision**?
- If we **ignore** the possibility of **flow control**, that is, we assume the **window** is **wide open**, as would be the case when a **connection first starts**.
- Then, TCP has **three mechanisms** to **trigger** the **transmission** of a **segment**.

Note: **Window** here **refers to** the available **window size** as per "**Sliding window**" algorithm.

Meaning of Subtle: (especially of a change or distinction) so delicate or precise as to be difficult to analyze or describe.

First Mechanism: Triggering Transmission

- TCP maintains a variable, typically called the **Maximum Segment Size** (**MSS**), and it sends a segment as soon as it has collected MSS bytes from the sending process.
- MSS is usually set to the size of the largest segment TCP can send **without causing the local IP to fragment**. □ **Why should it be avoided?**



Where is TCP Data?

Let us see this in detail soon !!!

- That is, MSS is set to the maximum transmission unit (MTU) of the directly connected network (datalink layer), minus the size of the TCP and IP headers
  - MSS = MTU of connected network – (Size of IP header + TCP header)
  - As you are aware, the TCP and IP header sizes are 20 bytes each, totaling 40 bytes of overhead for a TCP segment, with no optional fields added in both.

Second Mechanism: Triggering Transmission

- The second trigger is when the sending process explicitly asks TCP to send the data that has already been buffered in its Tx buf.
- As you are aware of the control flag PUSH that TCP supports for a PUSH operation, which makes the TCP stack to send whatever data available, without waiting for more data from the sender application.
- TCP can send all the data that it has, as long as the current window size advertised by the receiver is higher than the data that needs to be sent.

Third Mechanism: Triggering Transmission

- The **final trigger** for transmitting a segment is that **a timer fires**; the resulting segment that is sent would contain as many bytes as there are currently in the Tx buf for transmission.
- Time value is decided based on the RTT value of the connection between the hosts and the kind of applications requirement, interactive or data transfer, etc.
- Timer is set to RTT, to allow ACK to be received for the data sent. If the timer fires, that means either the data sent to the other end was lost or the ACK from the peer was lost.
- This result in a retransmission of data.

Silly Window Syndrome

Meaning of Silly: Having or showing a lack of common sense or judgement; absurd and foolish.

Meaning of Syndrome: A group of symptoms which consistently occur together, or a condition characterized by a set of associated symptoms.

Quiz 1: Generation of ACKs

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

- Choose **all the options** related to the **condition(s)** on which a **receiver generates an ACK**.

A. When a data is read by the application on the receiving end and it is removed from the Rx Buf by the TCP/IP stack

B. When the data is received by the TCP/IP stack and it is buffered into the Rx Buf, but it does not wait for the application on its end to read the data from the Rx Buf.

C. The receiver may combine the sending of ACK along with its own data, if there is some data ready to be sent.

D. The receiver sends one combined ACK after all the data in its Rx Buf gets emptied, i.e., the application on its end reads all of them.

ANS: B and C

Note: Rx application consuming the data from Rx Buf influences the Window size not the ACK. Window size element is sent along with the ACK segment.

Note: Remember the ACK is exchanged at the TCP peer level, it does not have to wait for the application to consume the data. In fact if it does, it does not make use of the data buffering at the TCP level and also slows down the sender by not allowing to empty its own Tx buf.

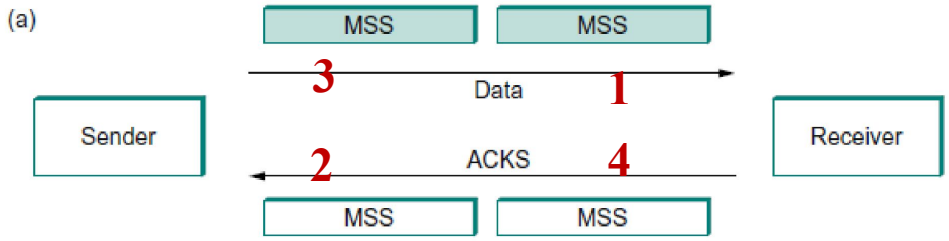
1. Silly Window Syndrome

- Though MSSs are agreed upon between the peers, we can't just ignore flow control, which plays an obvious role in throttling the sender.
- If the sender has MSS bytes of data to send and the window is open at least that much, then the sender transmits a full segment.
- Assume that the sender is accumulating bytes to send, but the window is currently closed.
- Now suppose an **ACK arrives** along with **the window size** that effectively **opens** the **window** enough for the sender to transmit, say, **MSS/2 bytes**. ⇦ **What would have triggered this MSS/2 window size?**

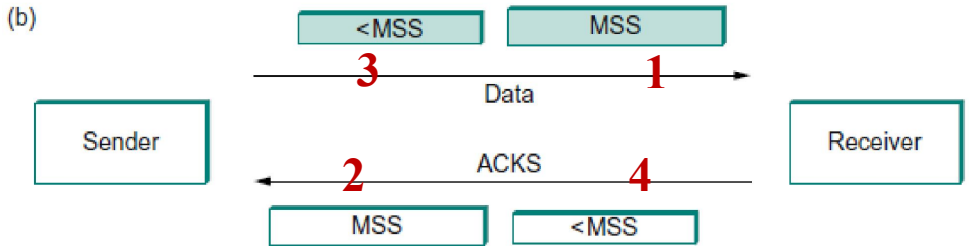
  ANS: The application at the receiver has read MSS/2 data from the Rx buffer.
- Should the sender transmit a half-full segment or wait for the window to open to a full MSS? ⇦ **Why should it wait for full MSS?**

  ANS: If full MSS data is sent in a single TCP segment, it is more efficient.
- Let us see an example …

2. Silly Window Syndrome: Example



- As long as the sender sends MSS-sized segments and the receiver ACKs one MSS at a time and opens window size also by MSS, the system works smoothly.



- As soon as the **sender sends data which is less than one MSS**, or the receiver ACKs less than one MSS, a small "**container**" or "**segment**" enters the system and continues to circulate.
  - Sender sends less than MSS though window size is MSS, because it has only less data in TxBuf which is available to be sent
- Which is not good from the network utilization perspective since smaller amount of data gets transmitted for a fixed overhead of 40 bytes.

3. Silly Window Syndrome

- It turns out that the strategy that **aggressively** takes advantage of any available window, leads to a situation now known as the **silly window syndrome**.
  - Which means that the sender deciding to send whatever the window size available, without waiting for it to be equal to MSS or more.
- The strategy here refers to the sender reacting to the availability of window sizes, which are smaller than MSS, by sending immediately a segment with a smaller size.
- Let us see **Nagle's algorithm** which addresses the problem with the Silly Window Syndrome.

Nagle's Algorithm

Note: Small segments are called **tinygrams**. Nagle's algorithm avoids tinygrams on the network thus improving the network performance.

Nagle's Algorithm: Background

- To avoid Silly window syndrome, if there is data to send but the window is open less than MSS, then TCP may want to wait for some amount of time before sending the available data, but the question is how long?
- If the TCP waits too long, then it hurts interactive applications like Telnet.
- If TCP doesn't wait long enough, then it risks sending a bunch of tiny packets and falling into the silly window syndrome.
- The answer is to introduce a timer and to transmit when the timer expires.
- While we could use a **clock-based timer** -for example, one that fires every 100 ms— **Nagle** introduced an elegant **self-clocking solution**
  - Fixed clock-based timer is not a preferred solution because it does not adapt to the technology advances in the networking and processors.
  - Fixed delay may hurt the performance of recent advanced applications.

Nagle's Algorithm: Background contd

- The idea of this algorithm is that **as long as TCP** has some **data in flight**, from Tx to Rx, the sender will eventually receive an ACK.
- This ACK can be treated like a timer firing, triggering the transmission of more data.
- But, why do we have to check if there is an ACK expected from the other end or not?
- You need to understand that the instance of TCP/IP stack created per connection (could be a process or a module) gets invoked based on any events getting triggered either from the top (application) or from the bottom (IP layer)
  1. **From the top**: When the application has something to transmit or it has consumed some data in the Rx buf causing the Window Size to increase.
  2. **From the bottom**: When the IP layer receives a TCP segment (may be with or without data) from the peer.
- **Nagle's algorithm** provides a simple, unified rule for deciding when to transmit: **Let us see it next …**

Note: Without data could be an ACK coming from the peer without its own data in the other direction.

Nagle's Algorithm (1)

```
When the application produces data to send
  if both the available data and the window ≥ MSS
    send a full segment (1)
  else
    if there is unACKed data in flight
      buffer the new data until an ACK arrives
    else
      send all the new data now
```

1. Full data of **max size MSS** can be sent only when **both** the following **two conditions** are **satisfied:**
   a) There is **sufficient data** (≥ **MSS**) in the Tx buf which is greater or equal to MSS.
   b) The receiver's **window size** (≥ **MSS**) allows full data to be sent.

- It's **always OK** to **send** a **full segment (MSS)** if **both the above conditions** are **true**.

Nagle's Algorithm (2)

```
When the application produces data to send
  if both the available data and the window ≥ MSS
    send a full segment
  else
    if there is unACKed data in flight
      buffer the new data until an ACK arrives
    else
      send all the new data now
```

2

Note: Open-ended means having no predetermined limit or boundary.

2. It's all right to wait for the ACK from the other end if there are currently segments in transit, instead of initiating data transfer immediately which is smaller than MSS.
   - The reason being that an event from the other end is **not open-ended**.
   - If there is a data in transit, the peer is likely to generate an ACK on receiving the data, even if it does not have its own data to send from its end, so the sender can afford to wait for some time. During wait time the window size could also increase.
   - If there is no data in transit, there is no guarantee that some data would come from the application at the top triggering an event too, so it is not advisable to wait. because it could be open-ended wait, impacting interactive application's requirements.

Nagle's Algorithm (3)

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

```
When the application produces data to send
  if both the available data and the window ≥ MSS
    send a full segment
  else
    if there is unACKed data in flight
      buffer the new data until an ACK arrives
    else
      send all the new data now
```

3

3. So, it's also all right to immediately send a small amount of data, even if it is smaller than MSS, if there are currently no segments in transit.

TCP_NODELAY option

- An interactive application like Telnet that continually writes one byte at a time will send data at a rate of one segment per RTT (Round Trip Time).
- Some segments will contain a single byte, while others will contain as many bytes as the user was able to type in one round-trip time.
- Because some applications (**Telnet**) cannot afford such a delay for each write it does to a TCP connection, the socket interface allows the application to **turn off Nagel's algorithm** by **setting** the **TCP_NODELAY** option.
- Setting this option means that data is transmitted as soon as possible.
- RTT is chosen here because that is the maximum delay that would be experienced to get a response for the commands typed on one end. RTT is much lower than delay introduced by Nagel's algorithm.

Datagram Congestion Control Protocol (DCCP)

DCCP is a transport layer protocol defined by **RFC 4340**.

DCCP

- It is designed to provide congestion control for **connectionless datagram-based** applications, bridging the gap between the reliability of TCP and the low-overhead of UDP.
- **Congestion Control:** Unlike UDP, which lacks built-in congestion control, DCCP incorporates mechanisms to manage network congestion, making it suitable for applications that require timely delivery without the overhead of ensuring reliability.
- **Unreliable Delivery:** DCCP does not guarantee the delivery of packets, allowing applications to handle any necessary retransmissions or error corrections.

- DCCP is a separate protocol (like TCP/UDP).
- Protocol number **33** (distinct in the IP header).
- It has its own header (does not reuse the UDP header)
- It has three-way handshake connection establishment
- 48-bit seq number for congestion control nor reliability.

DCCP Header Fields (simplified view)

Field	Size
Source Port	16 bits
Destination Port	16 bits
Data Offset	8 bits
CCVal / ResFlags	8 bits
Sequence Number	48 bits
Acknowledgement	48 bits (optional)
Payload	Variable

DCCP: Explained

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

- DCCP uses acknowledgments to inform the sender if its packets have arrived and if they were marked by Explicit Congestion Notification (ECN)
  - ECN is notified by intermediate routers while forwarding the packets
- Blind attackers try to guess the 32-bite sequence numbers used by TCP, DCCP increases it to 48-bit with additional guard bits, to make it harder for the attackers to predict the sequence numbers used by the connections.
- DCCP provides support for acknowledgments both in bidirectional and uni-directional data transfers.
- DCCP is designed to work well on networks with high levels of bandwidth fluctuation.
- It's intended for applications like streaming media, VOIP, online gaming

Quick UDP Internet Connections (QUIC)

QUIC

- QUIC is a transport layer network protocol developed by Google, built on top of UDP, aiming to reduce latency and improve performance compared to TCP
- Incorporates features like multiplexing, encryption, and improved congestion control within the protocol itself.
- **Fast Handshake:** Reduces connection establishment time, enabling quicker data transmission
- **Web Browsing:** Enhances loading times and responsiveness for web pages.
- **Streaming Services**: Provides smoother video and audio streaming experiences.
- **Real-Time Applications:** Supports applications requiring rapid data transmission, such as video conferencing and online gaming.

QUIC: Explained

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

- QUIC supports multiple independent streams within a single connection, reducing head-of-line blocking
- It has built-in selective acknowledgment, which allows it to adjust its congestion control more granularly than TCP.
- It uses a hybrid slow start algorithm that dynamically adjusts the window size based on round-trip times.
- Slow start uses smaller window size to study the network congestion before using larger window sizes
- It estimates bandwidth in each direction to avoid congestion.
- It can seamlessly migrate connections between different networks, such as Wi-Fi and mobile data.

Dynamic Adaptive Streaming over HTTP (DASH)

DASH

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

- DASH is an application layer protocol designed for streaming multimedia content over the Internet.
- It dynamically adjusts the quality of the video stream based on network conditions, ensuring smooth playback without buffering.
- **Adaptive Bitrate Streaming**: Adjusts video quality in real-time to match the user's available bandwidth.
- **Segmented Media Delivery**: Breaks video content into small segments, allowing for quick adjustments to streaming quality.
- **HTTP-Based**: Utilizes standard HTTP protocols, ensuring compatibility across various devices and platforms.
- Applications: Video streaming, live broadcasting, etc.

Host-centric Vs Router-centric Protocols

Is TCP host-centric or router-centric protocol? **ANS: Host-centric**

Host-centric Vs Router-centric Protocols

Host-centric	Router-centric
This category of protocols that focus on the end devices (hosts) **Note**: TCP is a good example of a protocol which is Host-centric	It describe approaches, protocols, or functionalities that primarily focus on the network infrastructure (routers)
Examples: TCP, application-layer protocols such as, HTTP, FTP, etc.	**Examples**: Routing protocols such as, OSPF and BGP, Quality of Service (QoS)
Protocols and mechanisms that are managed and controlled by the end hosts. These often involve data transmission, flow control, error handling, and application-specific functionalities	Protocols and mechanisms that are managed by network devices to control traffic flow, manage congestion, enforce policies, and optimize routing
Emerging protocols such as QUIC, SDN provide a more integrated approach between host-centric and router-centric functionalities. QUIC allows for more seamless interactions with modern network features like Explicit Congestion Notification (ECN) from routers, that enables more optimized routing, bridging host and router-centric functionalities.	