Chapter Four

The Transport Layer

Chapter Outline

Why Transport Layer?

1. The network layer exists on end hosts ***and routers in the network***
   - end-user cannot control what is in the network.
   - So the end-user establishes another layer, only at end hosts, to provide a transport service that is more reliable than the underlying network service
2. While the network layer deals with only a few transport entities, the transport layer allows several **concurrent applications** to use the transport service
3. It provides a **common interface to application writers**, regardless of the underlying network layer.
   - an application writer can write code once using the transport layer primitive and use it on different networks (but with the same transport layer).

Expected Properties

- Guaranteed message delivery
- Message order preservation
- No duplication of messages
- Support for arbitrarily large messages
- Support for sender/receiver synchronization
- Receiver based flow control
- Support multiple application processes per host

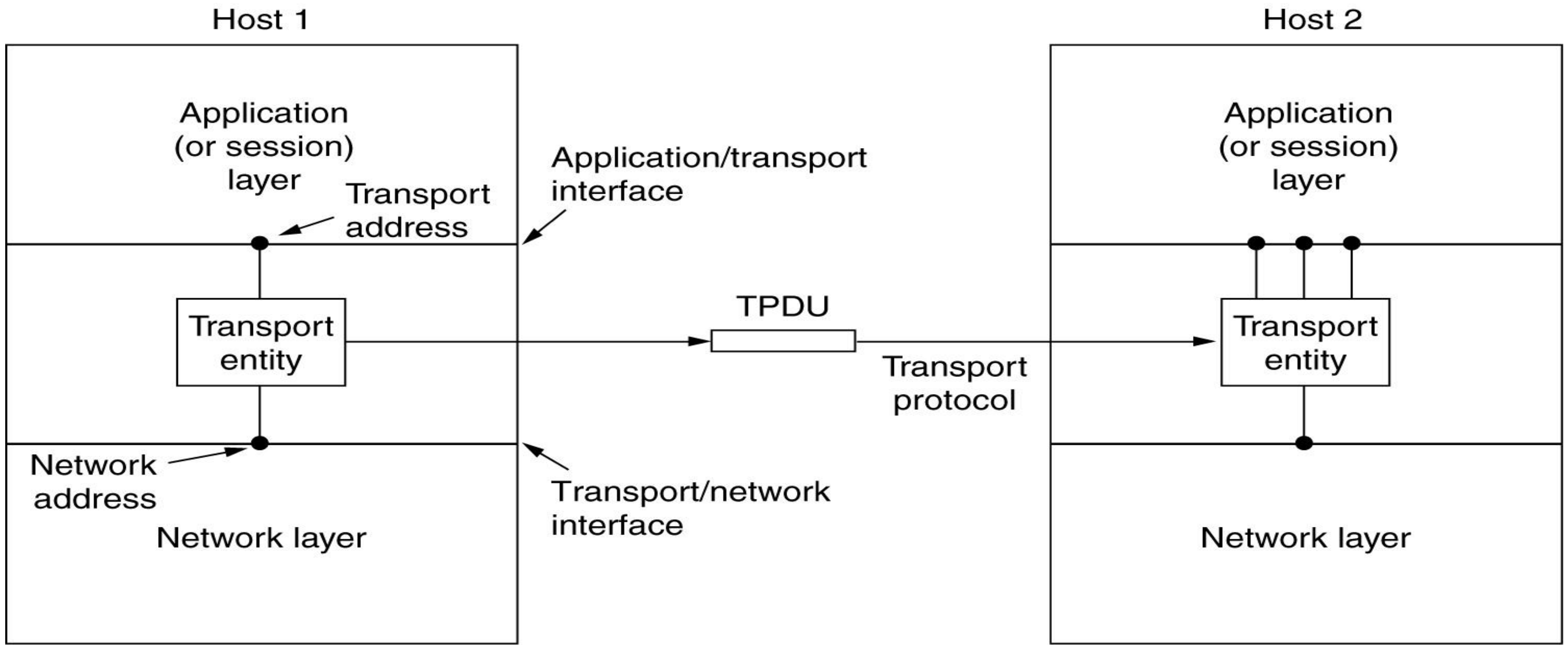
Relationship with other layers

Application	What are the expectations of the functionality provided by underlying layers?
Transport	Provides end-to-end features
Network	May drop, reorder, or duplicate messages. May introduce arbitrarily long delays. May limit the size of messages to a maximum value.

Transport Services and protocols

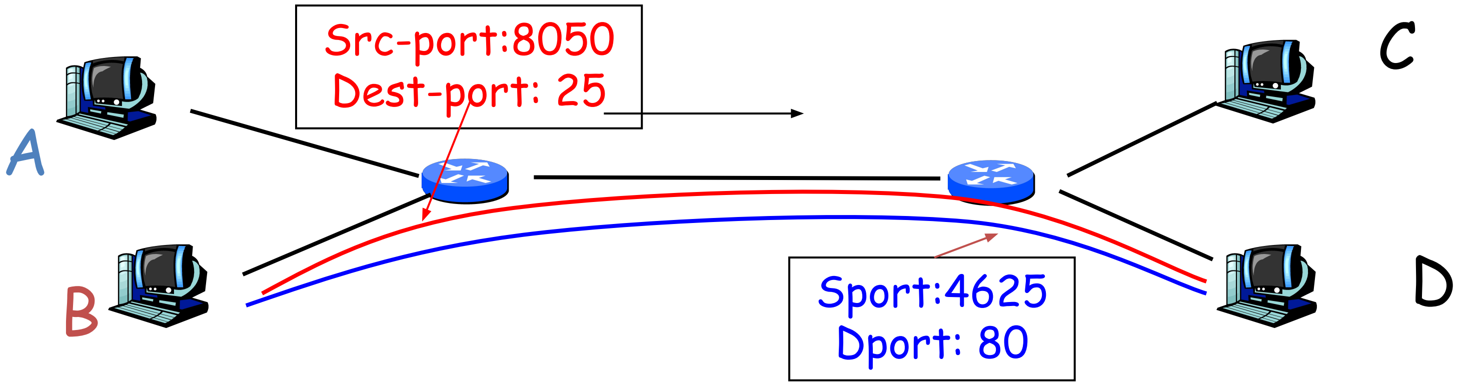
- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP

Services provided to the upper layers



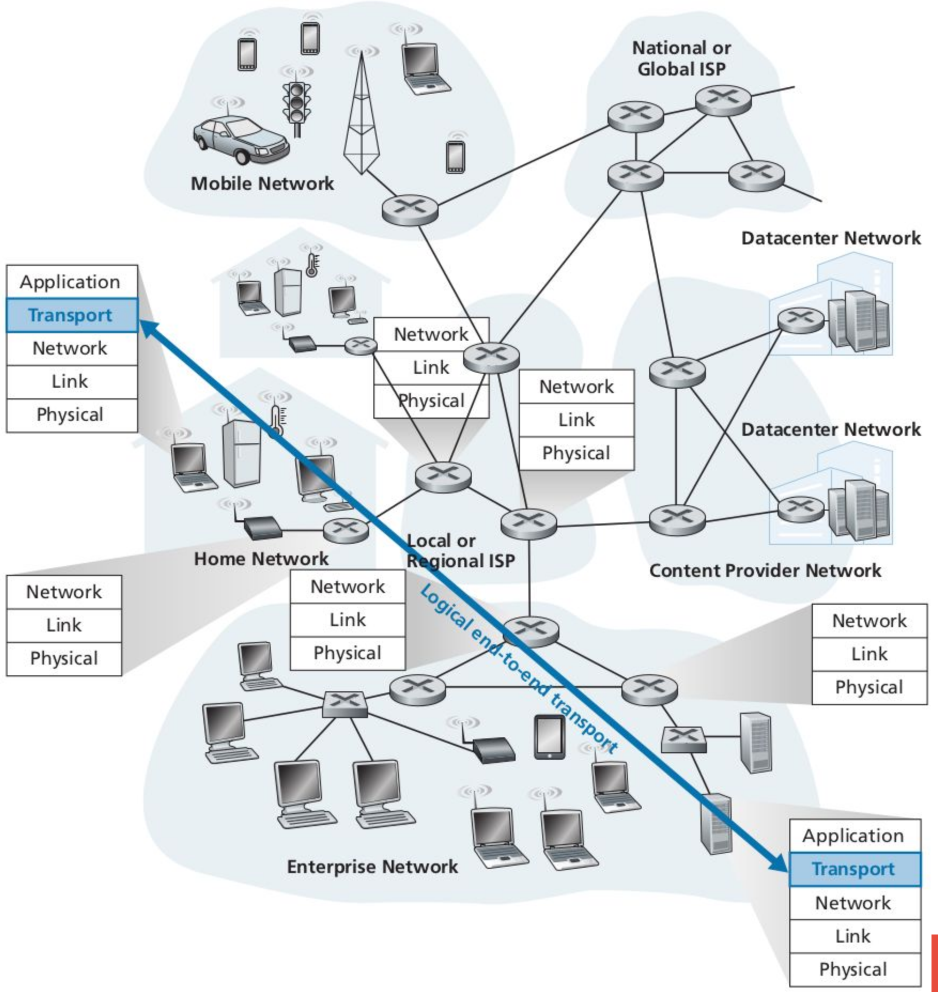
Transport vs Network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services



Internet Transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Chapter Outline

Elements of Transport Protocol

- ✓ Addressing
- ✓ Connection Establishment
- ✓ Connection Release
- ✓ Flow Control and Buffering
- ✓ Multiplexing
- ✓ Crash Recovery

1. Addressing

- When an application (e.g., a user) process wishes to set up a connection to a remote application process, it must specify which one to connect to.
- The method normally used is to define transport addresses to which processes can listen for connection requests.
- In the Internet, these end points are called **ports or TSAP** (Transport Service Access Point).

2. Establishing connection

- Send a **CONNECTION_REQUEST TPDU** to the destination and wait for a **CONNECTION_ACCEPTED** reply.
- The problem occurs when the network can lose, store and duplicate packets.
- Problem: existence of delayed duplicates.
- Solutions:
  1. using **throw-away transport addresses** after releasing connection.
  2. give each connection a **connection identifier**
- This scheme has a basic flaw:
  - it requires each transport entity to maintain a certain amount of history information indefinitely.
  - If a machine crashes and loses its memory, it will no longer know which connection identifiers have already been used.

2. Establishing connection

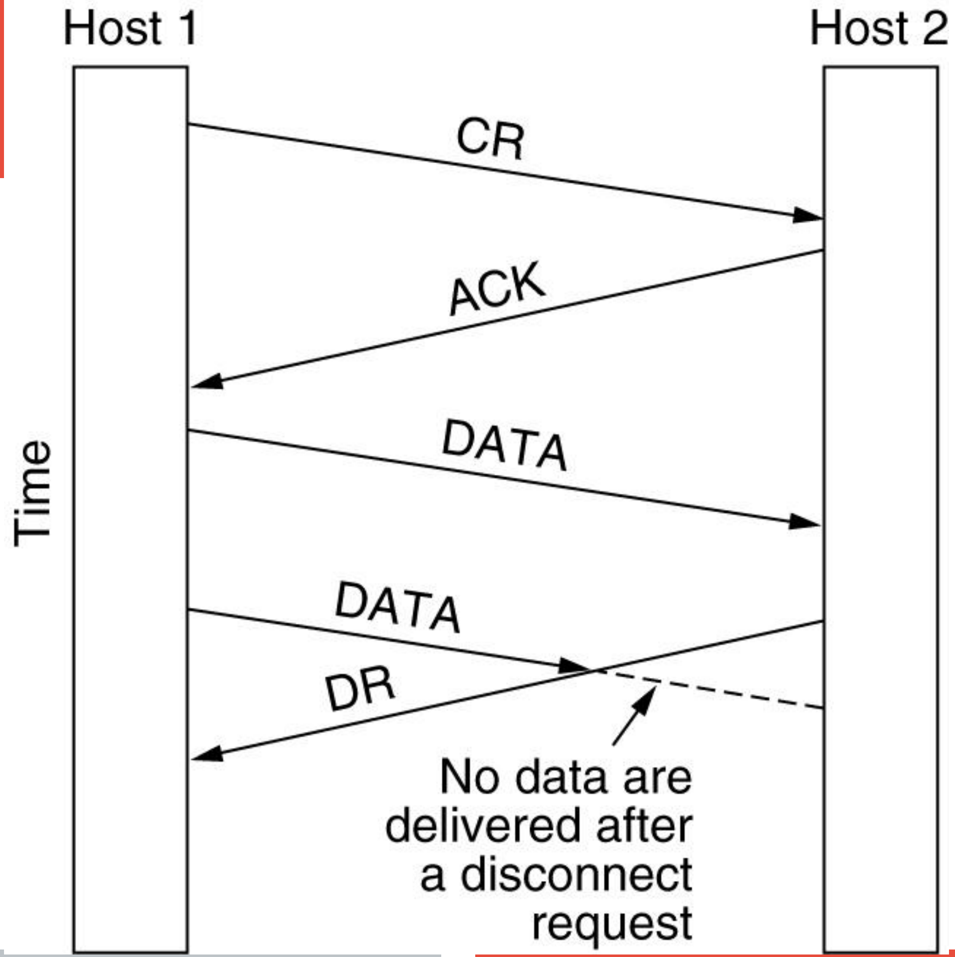
- Rather than allowing packets to live forever within the subnet, devise a mechanism to kill off aged packets that are still hobbling about.
- If no packet lives longer than some known time, the problem becomes somewhat more manageable.
- Packet lifetime can be restricted to a known maximum using one of the following techniques:
  - Restricted subnet design
  - Putting a hop counter in each packet
  - Timestamping each packet

3. Releasing connection

- **Asymmetric release**
  - Like telephone: when one party hangs up, the connection is broken.
- **Symmetric release**
  - It treats the connection as 2 separate unidirectional connections and requires each one to be released separately.

Asymmetric Release

Abrupt disconnection with loss of data.



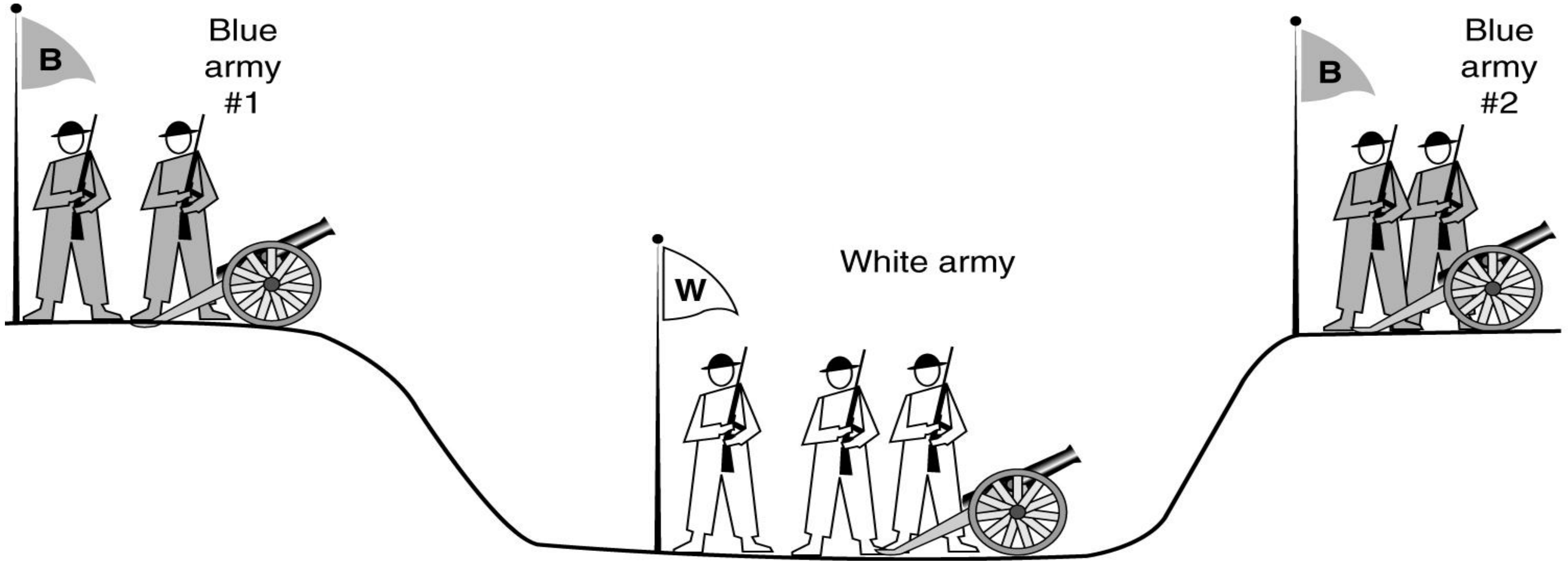
Asymmetric Release

- After the connection is established, host 1 sends a **TPDU** that arrives properly at host 2.
- Then host 1 sends another **TPDU**.
- Unfortunately, host 2 issues a **DISCONNECT** before the second **TPDU** arrives.
- The result is that the connection is released and data are lost.

Symmetric Release

- A more sophisticated release protocol is needed to avoid data loss.
- One way is to use **symmetric release**, in which each direction is released independently of the other one.
- Here, a host can continue to receive data even after it has sent a DISCONNECT TPDU.
- Symmetric release does the job when each process has a fixed amount of data to send and clearly knows when it has sent it.
- One can envision a protocol in which host 1 says: **I am done. Are you done too**? If host 2 responds: I **am done too. Goodbye**, the connection can be safely released

The Two Army Problem



The Two Army Problem

- Imagine that a white army is encamped in a valley, as shown in fig.
- On both of the surrounding hillsides are blue armies.
- The white army is larger than either of the blue armies alone, but together the blue armies are larger than the white army.
- If either blue army attacks by itself, it will be defeated, but if the two blue armies attack simultaneously, they will be victorious.

The Two Army Problem

- The blue armies want to synchronize their attacks.
- However, their only communication medium is to send messengers on foot down into the valley, where they might be captured and the message lost (i.e., they have to use an unreliable communication channel).
- Does a protocol exist that allows the blue armies to win?

The Two Army Problem

- Suppose that the commander of blue army #1 sends a message reading: "I propose we attack at dawn on March 29. How about it?"
- Now suppose that the message arrives, the commander of blue army #2 agrees, and his reply gets safely back to blue army #1.
- Will the attack happen? Probably not, because commander #2 does not know if his reply got through. If it did not, blue army #1 will not attack, so it would be foolish for him to charge into battle.

The Two Army Problem

- Now improve the protocol by making it a three-way handshake.
- The initiator of the original proposal must acknowledge the response.
- Assuming no messages are lost, blue army #2 will get the acknowledgement, but the commander of blue army #1 will now hesitate.
- After all, he does not know if his acknowledgement got through, and if it did not, he knows that blue army #2 will not attack.
- We could make a four-way handshake protocol, but that does not help either.

The Two Army Problem

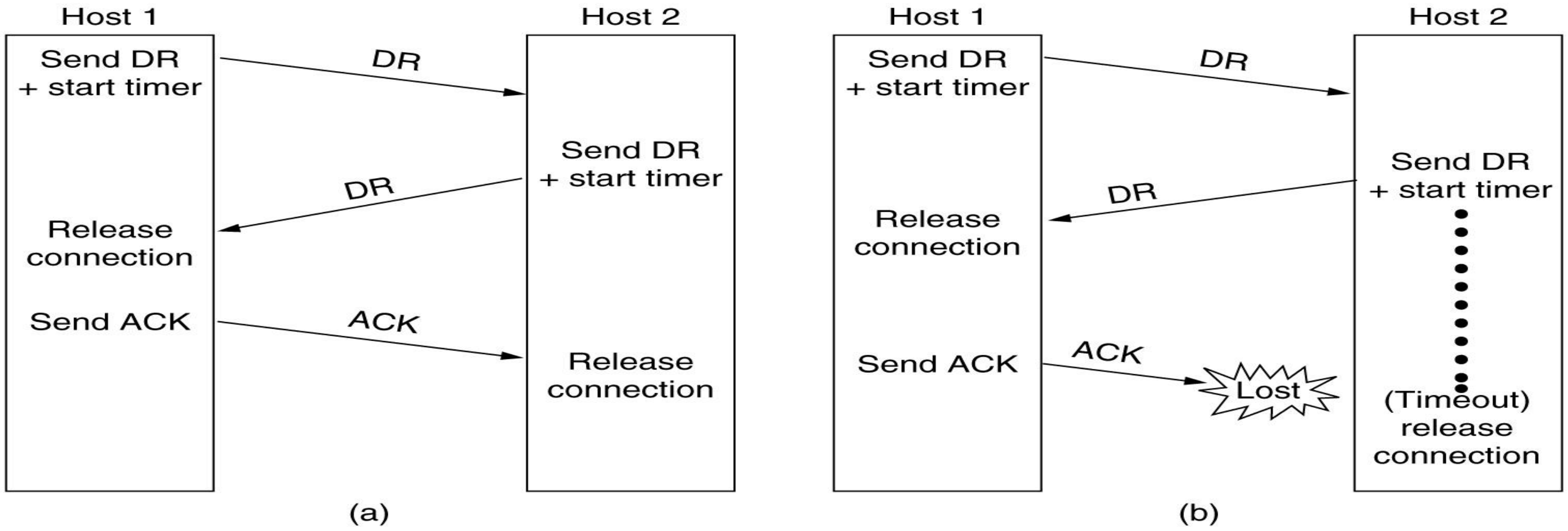
- To see the relevance of the two-army problem to releasing connections, just substitute "disconnect" for "attack."“
- If neither side is prepared to disconnect until it is convinced that the other side is prepared to disconnect too, the disconnection will never happen.
  - **DR** – Disconnect Request

3. Releasing Connection

Four protocol scenarios for releasing a connection

(a) Normal case of a three-way handshake.

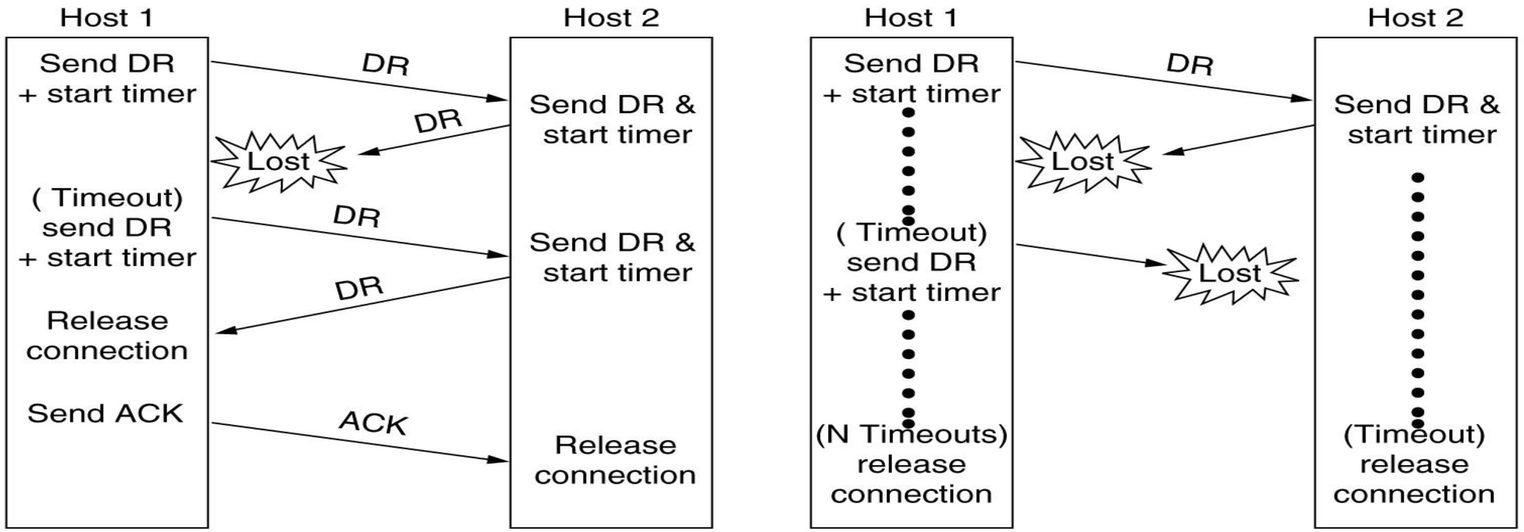
(b) final ACK lost.



3. Releasing Connection

(c) Response lost

(d) Response lost and subsequent DRs lost.



(c)

(d)

3. Releasing Connection

- While this protocol usually works, in theory it can fail if the initial **DR** and **N** retransmissions are all lost.
- The sender will give up and release the connection, while the other side knows nothing at all about the attempts to disconnect and is still fully active.
- This situation results in a **half-open connection**.
- We could avoid this problem by not allowing the sender to give up after **N** retries but forcing it to go on forever until it gets a response.
- One way to kill off half-open connections is to have a rule saying that if no **TPDUs** have arrived for a certain number of seconds, the connection is then automatically disconnected

4. Flow Control and Buffering

- **flow control** - make sure a fast sender cannot swamp a slow receiver
- If the network service is unreliable, the sender must buffer all **TPDUs** sent,
- If the receiver cannot guarantee that every incoming **TPDU** will be accepted, the sender will have to buffer anyway.
- The sender cannot trust the network layer's acknowledgement, because the acknowledgement means only that the **TPDU** arrived, not that it was accepted.

4. Flow Control and Buffering

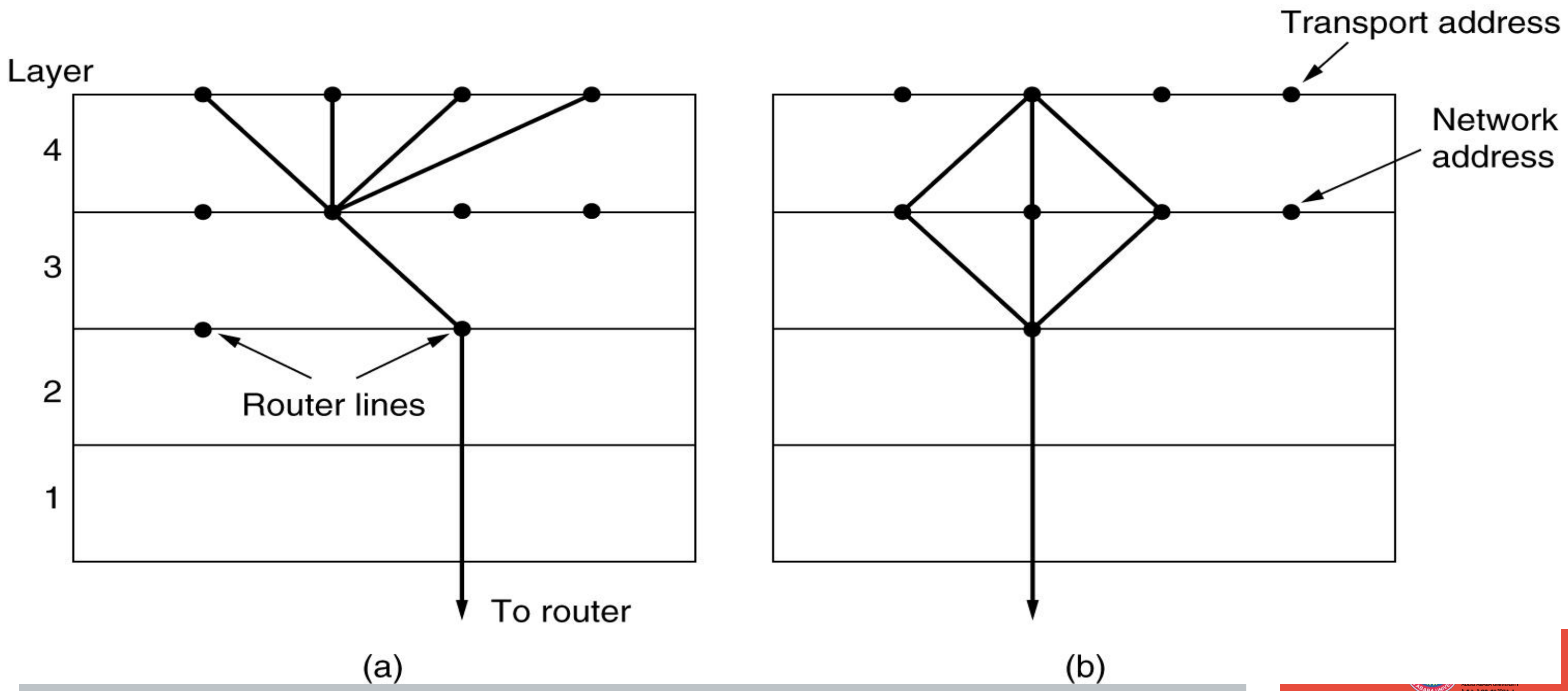
- Even if the receiver has agreed to do the buffering, there still remains the question of the buffer size.
- If most **TPDUs** are nearly the same size, it is natural to organize the buffers as a pool of identically-sized buffers, with one **TPDU** per buffer
- If there is wide variation in **TPDU** size, a pool of fixed-sized buffers presents problems.
- If the buffer size is chosen equal to the largest possible **TPDU**, space will be wasted whenever a short **TPDU** arrives.
- If the buffer size is chosen less than the maximum **TPDU** size, multiple buffers will be needed for long **TPDUs**, with the attendant complexity.

5. Multiplexing

- In the transport layer the need for multiplexing can arise in a number of ways.
  - only one network address (e.g., IP address) is available on a host, all transport connections on that machine have to use it.
- When a **TPDU** comes in, some way is needed to tell which process to give it to.
  - **upward multiplexing**
- If a user needs more bandwidth than one virtual circuit can provide, a way out is to open multiple network connections and distribute the traffic among them on a round-robin basis
  - **downward multiplexing**

5. Multiplexing

(a) Upward multiplexing. (b) Downward multiplexing.



Chapter Outline

UDP: User Datagram Protocol

- User Datagram Protocol – UDP
  - RFC 768
- “best effort” service, UDP segments may be:
  - lost
  - delivered out of order to app
- Why use UDP?
  - No connection establishment cost (critical for some applications, e.g., DNS)
  - No connection state
  - Small segment headers (only 8 bytes)
  - Finer application control over data transmission

UDP: User Datagram Protocol

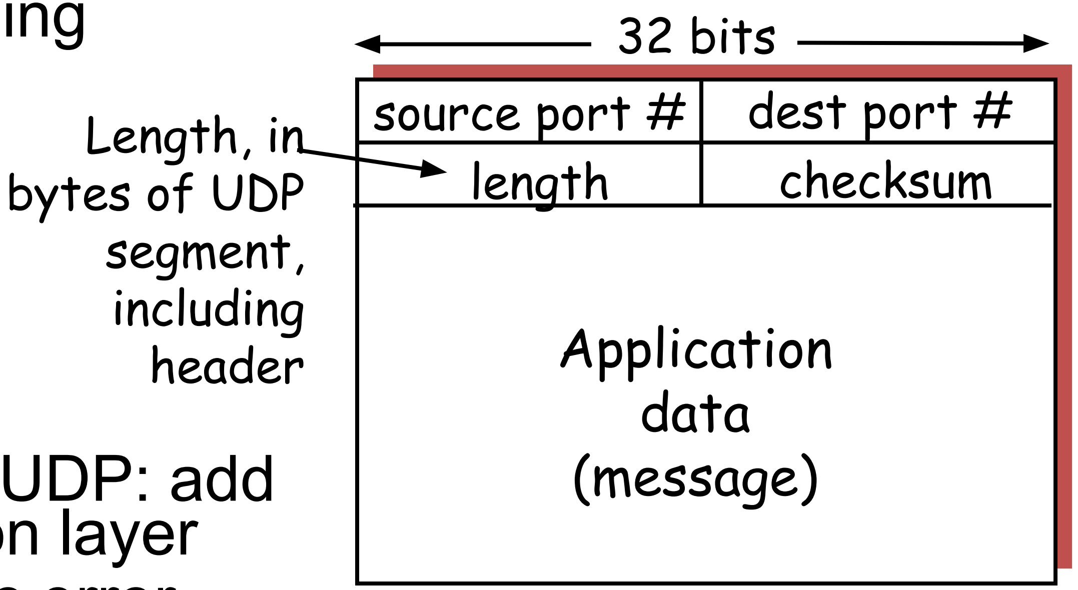
- End-to-End communication for user processes
- Processes must communicate with packets (byte arrays)
- Processes must be prepared:
  - for some packets not to arrive
  - for other packets to arrive out of order

UDP: User Datagram Protocol

- How do we send and reconstruct multiple large objects with UDP?
- Fragment objects in to byte blocks
  - agree a size and add delimiters between objects which share the same block, number each block and say which object it belongs to
- Send blocks one at a time
- When a block is received check which object it is for, place it in a buffer for that object
- When the end block of an object is found, if that object is complete, return it to the application,

UDP Segment Structure

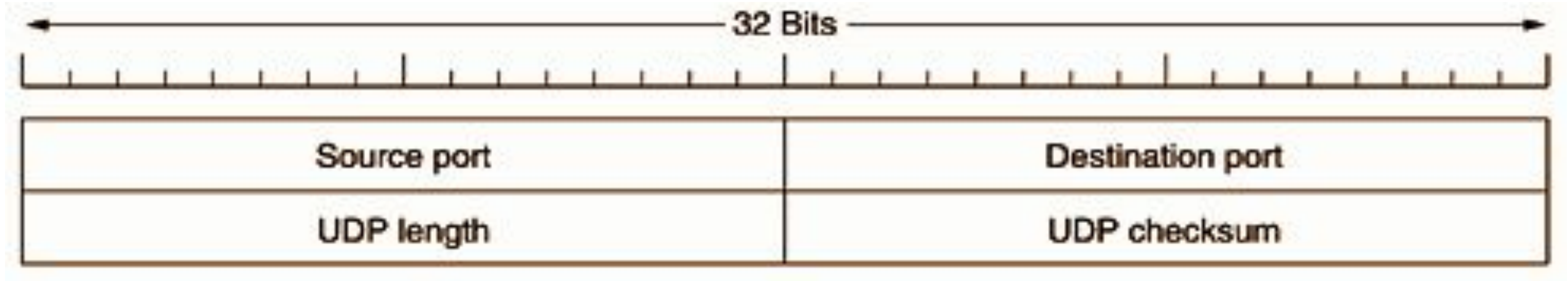
- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!


32 bits
Length, in bytes of UDP segment, including header
source port #
dest port #
length
checksum
Application data (message)

UDP segment format

UDP Header

- UDP transmits segments consisting of an 8-byte header followed by the payload
- When a UDP packet arrives, its payload is handed to the process attached to the destination port



UDP Header

- The source port is primarily needed
  - when a reply must be sent back to the source
- The UDP length field
  - includes the 8-byte header and the data.
- The UDP checksum
  - is optional
  - stored as 0 if not computed (a true computed 0 is stored as all 1s).
  - Turning it off is foolish unless the quality of the data does not matter (e.g., digitized speech).

Chapter Outline

The Internet Transport Protocols: TCP

- TCP (Transmission Control Protocol) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork.
- An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters.
- TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.
- TCP was formally defined in RFC 793.
  - As time went on, various errors and inconsistencies were detected, and the requirements were changed in some areas.
  - These clarifications and some bug fixes are detailed in RFC 1122.
  - Extensions are given in RFC 1323
  - Latest RFC 9293

The Internet Transport Protocols: TCP

- A TCP entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64 KB (in practice, often 1460 data bytes in order to fit in a single Ethernet frame with the IP and TCP headers), and sends each piece as a separate IP datagram.
- When datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which reconstructs the original byte streams

The TCP Service Model

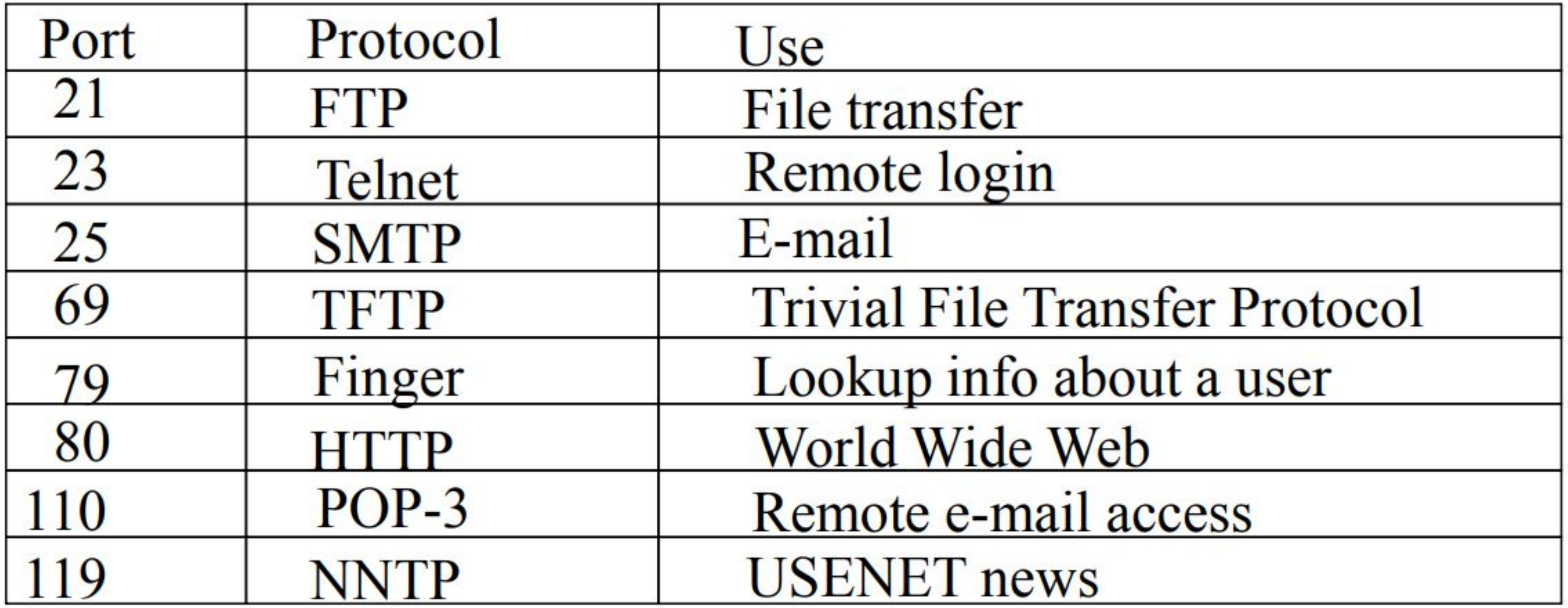
- TCP service is obtained by both the sender and receiver creating endpoints, called **sockets.**
- Each socket has a socket number (address) consisting of
  - the IP address of the host and
  - a 16-bit port number.
- For TCP service to be obtained, a connection must be explicitly established between a socket on the sending machine and a socket on the receiving machine.
- A socket may be used for multiple connections at the same time.
- two or more connections may terminate at the same socket.

The TCP Service Model

- Port numbers below 256 are called well-known ports and are reserved for standard services.
- All TCP connections are full duplex and point-to-point.
    - Full duplex means that traffic can go in both directions at the same time.
    - Point-to-point means that each connection has exactly two endpoints.
- TCP does not support multicasting or broadcasting.

The TCP Service Model

Some assigned ports.



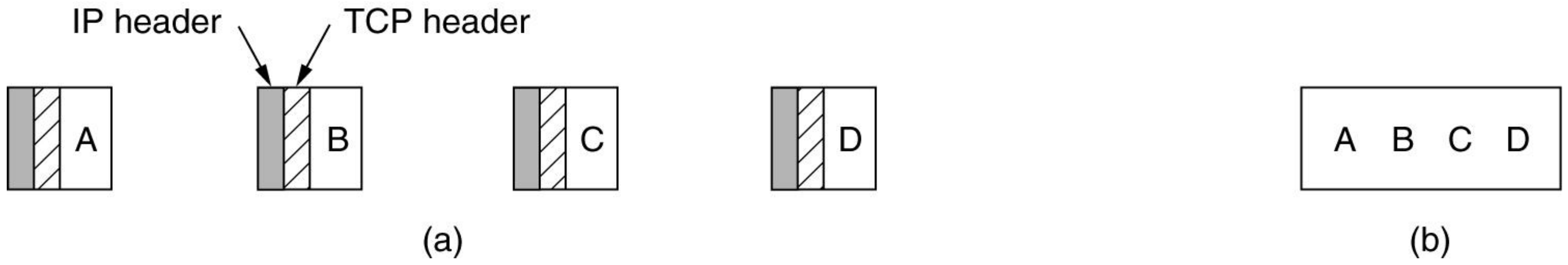
Port	Protocol	Use
21	FTP	File transfer
23	Telnet	Remote login
25	SMTP	E-mail
69	TFTP	Trivial File Transfer Protocol
79	Finger	Lookup info about a user
80	HTTP	World Wide Web
110	POP-3	Remote e-mail access
119	NNTP	USENET news

The TCP Service Model

- A TCP connection is a byte stream, not a message stream.
- Message boundaries are not preserved end to end.
- For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as
  - four 512-byte chunks,
  - two 1024-byte chunks, or
  - one 2048-byte chunk
- There is no way for the receiver to detect the units in which the data were written.

The TCP Service Model

(a) Four 512-byte segments sent as separate IP datagrams.
(b) The 2048 bytes of data delivered to the application in a single READ CALL.



The TCP Protocol

- Every byte on a TCP connection has its own 32-bit sequence number for acknowledgement and window mechanism
- The sending and receiving TCP entities exchange data in the form of segments.
- A TCP segment consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes.
- The TCP software decides how big segments should be:
    - Each segment, including the TCP header, must fit in the 65,515-byte IP payload.
    - Each network has a **Maximum Transfer Unit**, or **MTU**, and each segment must fit in the MTU.
- In practice, the MTU is generally 1500 bytes (the Ethernet payload size) and thus defines the upper bound on segment size.
- A segment that is too large for a network can be broken into multiple segments by a router

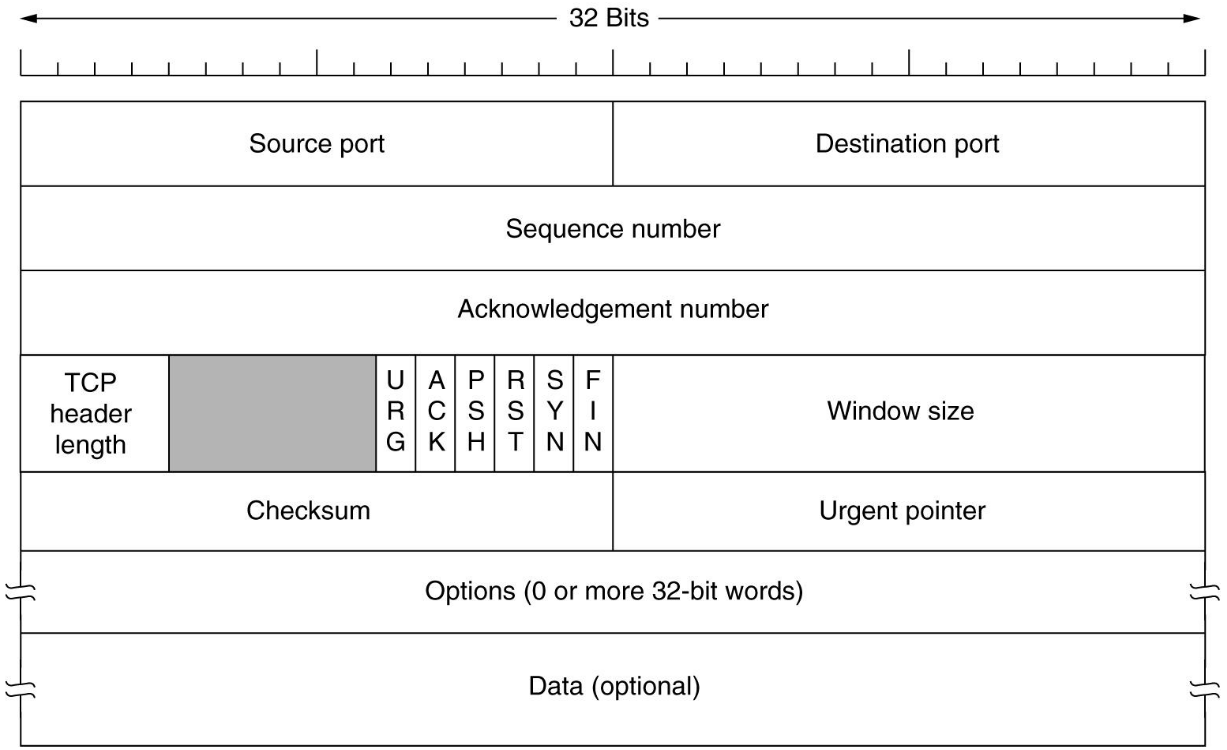
The TCP Protocol

- The basic protocol used by TCP entities is the **sliding window protocol**.
- When a sender transmits a segment, it also starts a timer.
- When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exist, otherwise without data) bearing an acknowledgement number equal to the next sequence number it expects to receive.
- If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again.

The TCP Segment Header

- Every segment begins with a fixed-format 20-byte header.
- The fixed header may be followed by header options.
- After the options, if any, up to 65,535 - 20 - 20 = 65,495 data bytes may follow, where the
  - first 20 refer to the IP header and
  - second to the TCP header.
- Segments without any data are legal and are commonly used for
  - acknowledgements and
  - control messages.

The TCP Segment Header



TCP Header

The TCP Segment Header

- The **Source port** and **Destination port** fields identify the local end points of the connection.
  - A port plus its host's IP address forms a 48-bit unique end point (TSAP).
- **Acknowledgement number** specifies the next byte expected, not the last byte correctly received.
- The **TCP header length** tells how many 32-bit words are contained in the TCP header.
  - Technically, this field really indicates the start of the data within the segment, measured in 32-bit words, but that number is just the header length in words, so the effect is the same
- This information is needed because the Options field is of variable length, so the header is, too.
- Next comes a 6-bit field that is not used

The TCP Segment Header

Next comes six 1-bit flags.

1. **URG** is set to 1 if the Urgent pointer is in use.
    - The **Urgent pointer** is used to indicate a byte offset from the current sequence number at which urgent data are to be found.
2. The **ACK** bit
    - set to 1 to indicate that the Acknowledgement number is valid.
    - If ACK is 0, the segment does not contain an acknowledgement so the Acknowledgement number field is ignored.
3. The **PSH** bit indicates PUSHed data.
    - Applications can use the PUSH flag, which tells TCP not to delay the transmission.
4. The **RST** bit
    - used to reset a connection that has become confused due to a host crash or some other reason.
    - It is also used to reject an invalid segment or refuse an attempt to open a connection.
    - if you get a segment with the RST bit on, you have a problem on your hands.

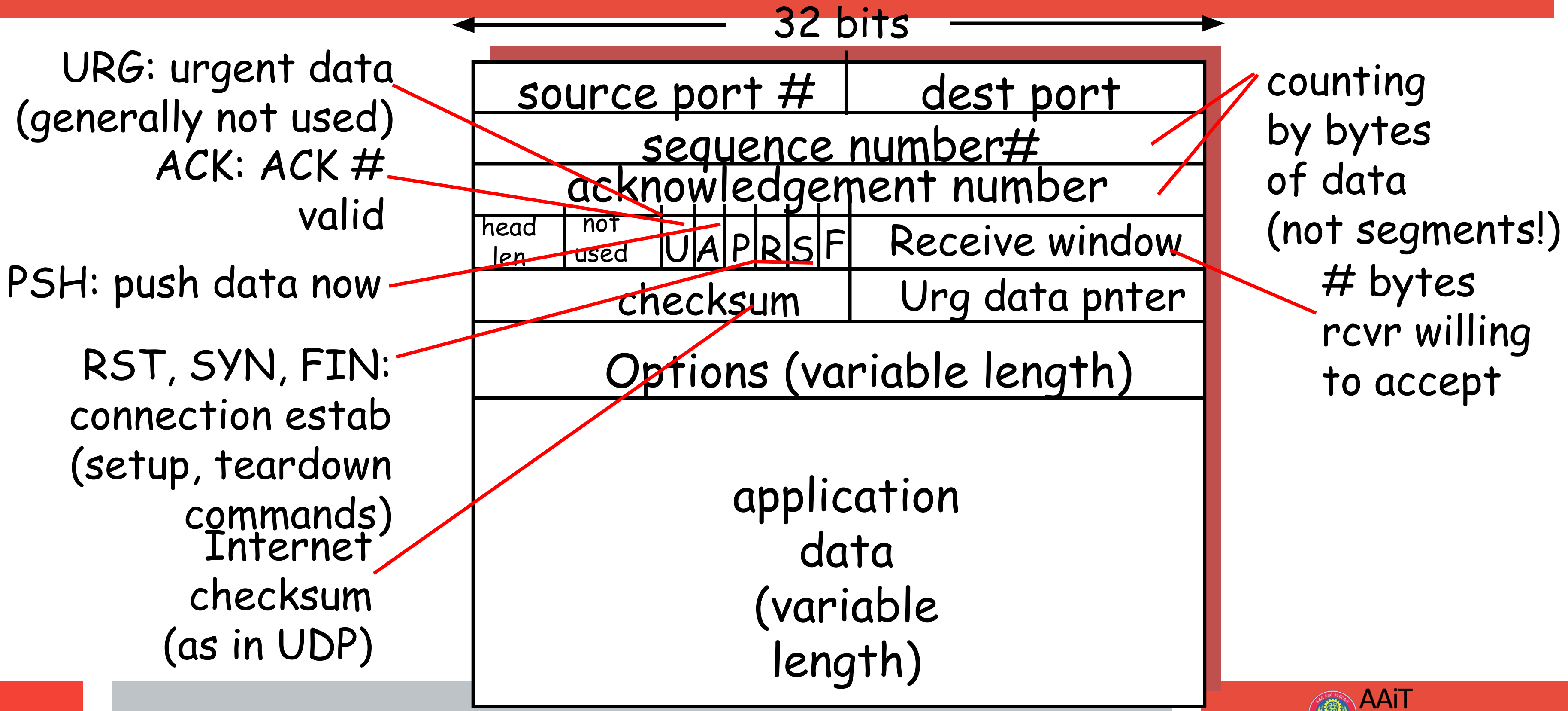
The TCP Segment Header

5. The **SYN** bit is used to establish connections.
   - The connection request has SYN = 1 and ACK = 0 to indicate that the piggyback acknowledgement field is not in use.
   - The connection reply bears an acknowledgement it has SYN = 1 and ACK = 1.
   - In essence the SYN bit is used to denote CONNECTION REQUEST and CONNECTION ACCEPTED, with the ACK bit used to distinguish between those two possibilities.
6. The **FIN** bit is used to release a connection. It specifies that the sender has no more data to transmit.

The TCP Segment Header

- Flow control in TCP is handled using a variable-sized sliding window.
- The **Window size** field tells how many bytes may be sent starting at the byte acknowledged.
- A **Checksum** is also provided for extra reliability.
- The **Options** field provides a way to add extra facilities not covered by the regular header.
- The most important option is the one that allows each host to specify the maximum TCP payload it is willing to accept.

TCP Segment Structure


32 bits
source port #
dest port
sequence number#
acknowledgement number
head len
not used
U
A
P
R
S
F
Receive window
checksum
Urg data pnter
Options (variable length)
application data (variable length)
URG: urgent data (generally not used)
ACK: ACK # valid
PSH: push data now
RST, SYN, FIN: connection estab (setup, teardown commands)
Internet checksum (as in UDP)
counting by bytes of data (not segments!)
# bytes rcvr willing to accept

TCP Connection Setup - Three-way Handshaking

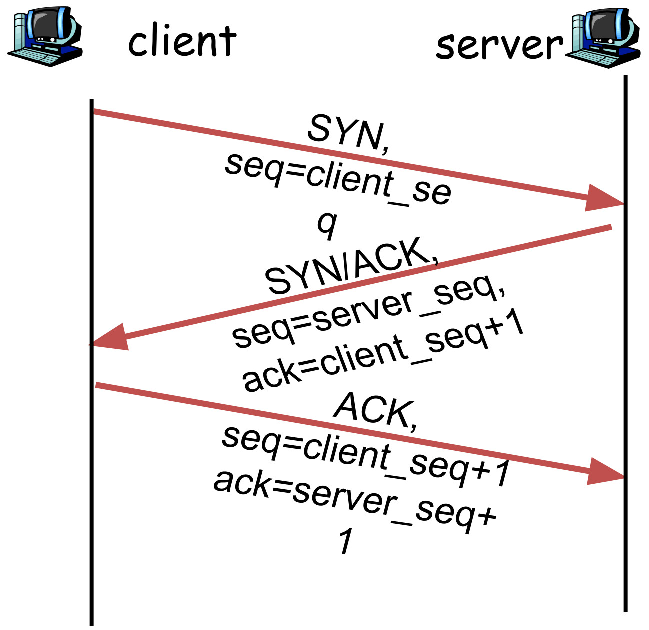
Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYN/ACK segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYN/ACK, replies with ACK segment, which may contain data


client
server
SYN, seq=client_seq
SYN/ACK, seq=server_seq, ack=client_seq+1
ACK, seq=client_seq+1 ack=server_seq+1

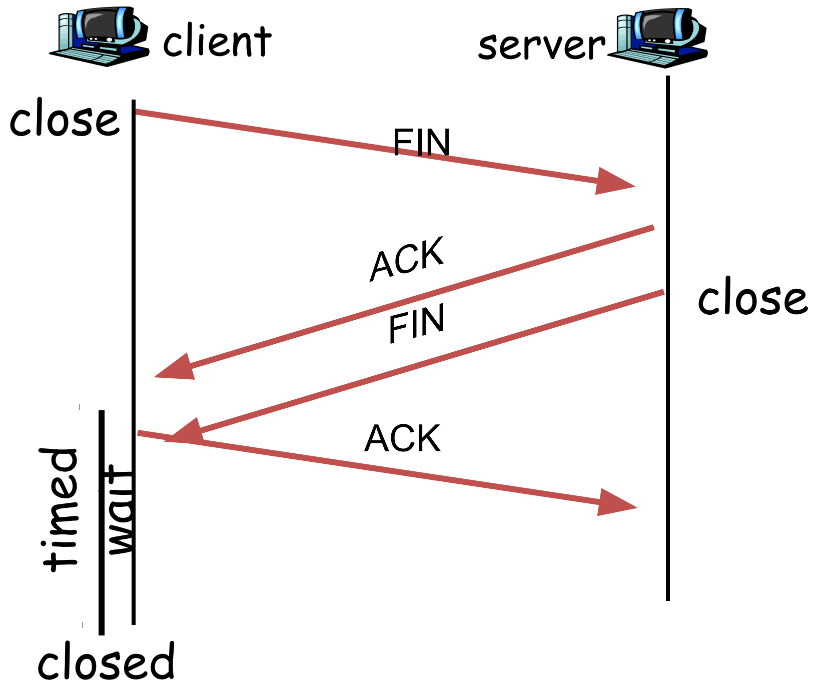
TCP Connection Management

Closing a connection:

close();

Step 1: client end system sends TCP/FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



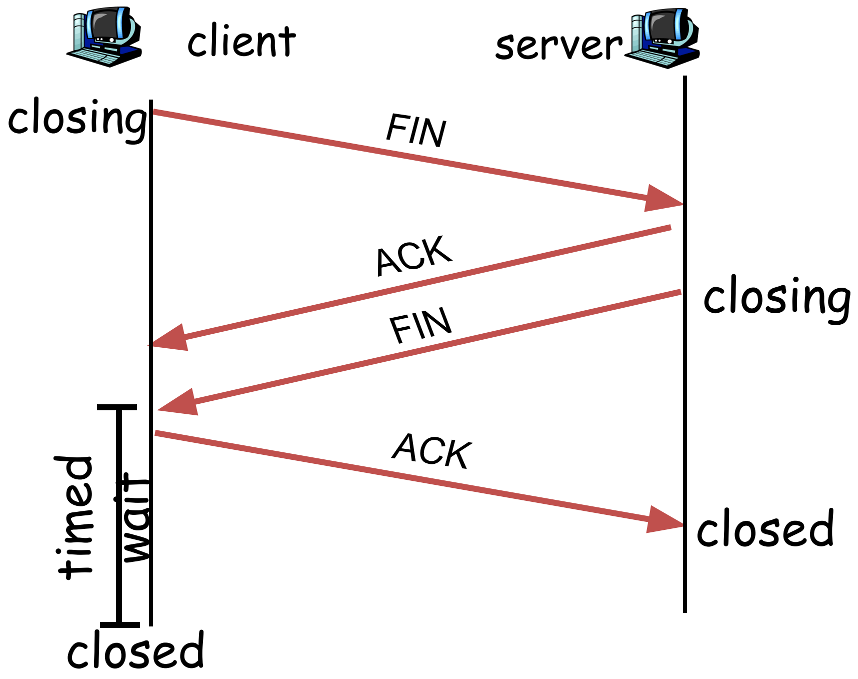
TCP Connection Management

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Some applications simply send RST to terminate TCP connections immediately



Chapter Outline

Flow Control

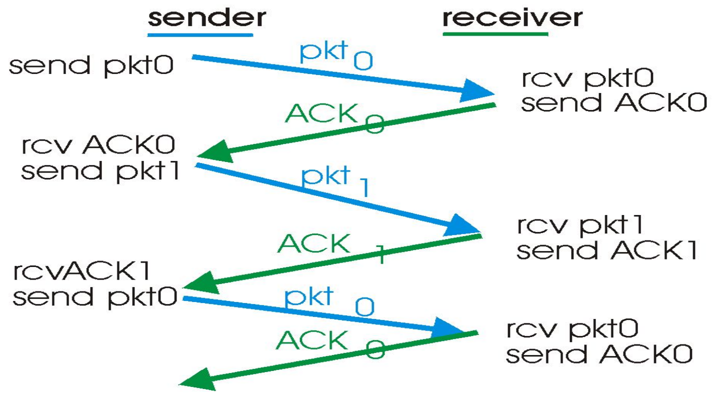
- End-to-end flow and Congestion control study is complicated by:
  - Heterogeneous resources (links, switches, applications)
  - Different delays due to network dynamics
  - Effects of background traffic
- We start with a simple case: hop-by-hop flow control

Hop-by-hop flow control

- Approaches/techniques for hop-by-hop flow control
  - Stop-and-wait
  - sliding window
    - Go back N
    - Selective reject

Stop-and-wait: reliable transfer over reliable channel

- underlying channel perfectly reliable
  - no bit errors, no loss of packets



(a) operation with no loss

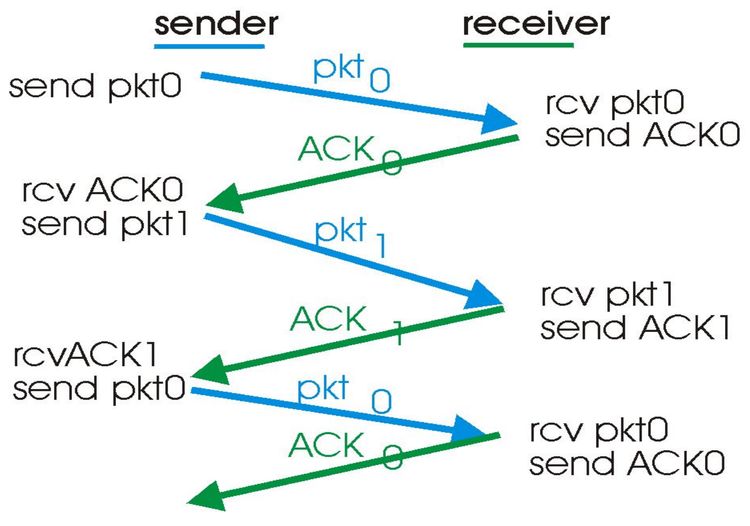
stop and wait

Sender sends one packet, then waits for receiver response

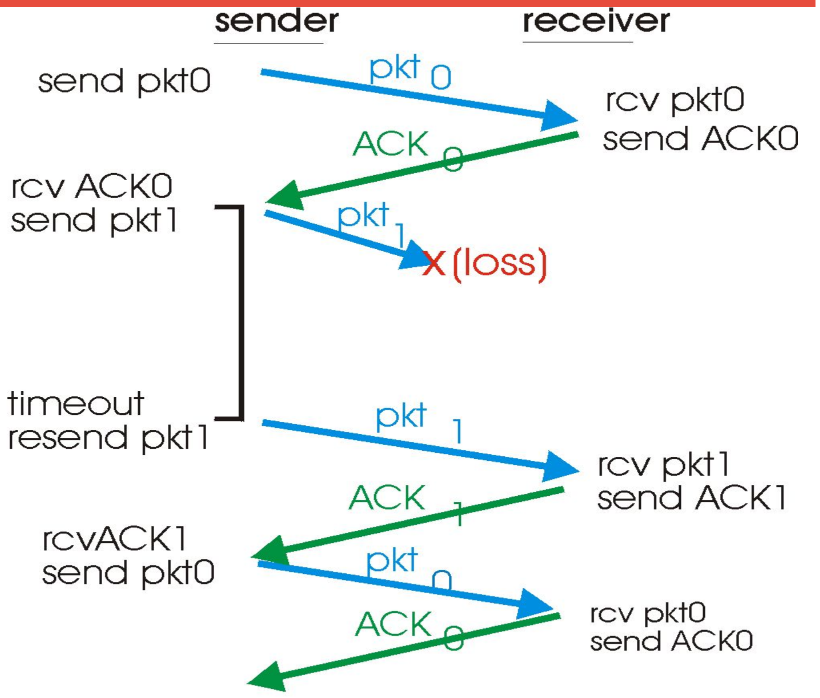
Channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms for:
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

Stop-and-wait with lost packet/frame

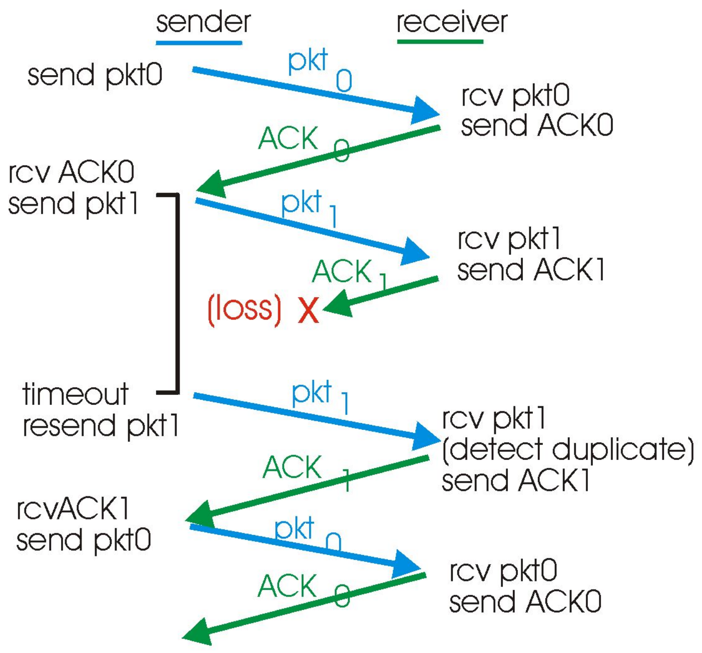


(a) operation with no loss

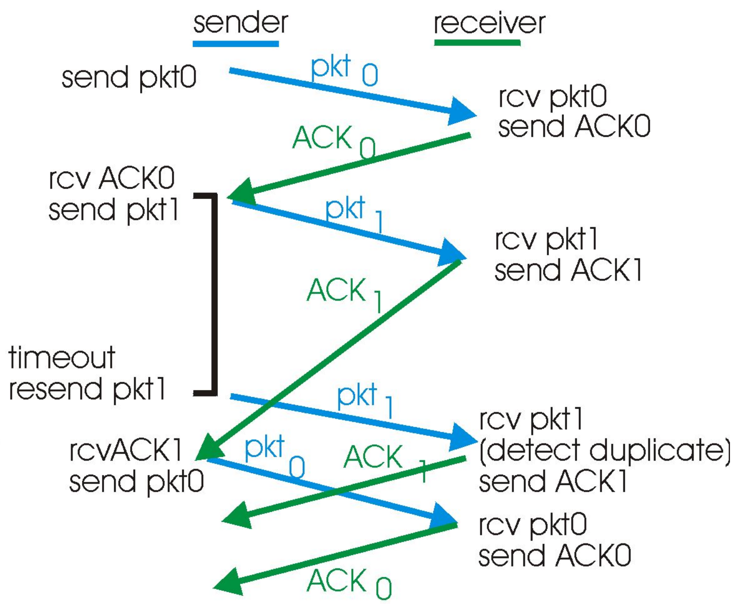


(b) lost packet

Stop-and-wait with lost packet/frame



(c) lost ACK



(d) premature timeout

Sliding window technique

- In all sliding window protocols, each outbound frame contains a sequence number, ranging from 0 up to some maximum.
- The maximum is usually $2^n - 1$ so the sequence number fits exactly in an *n-bit* field.
- The stop-and-wait sliding window protocol uses n = 1, restricting the sequence numbers to 0 and 1, but more sophisticated versions can use arbitrary n.

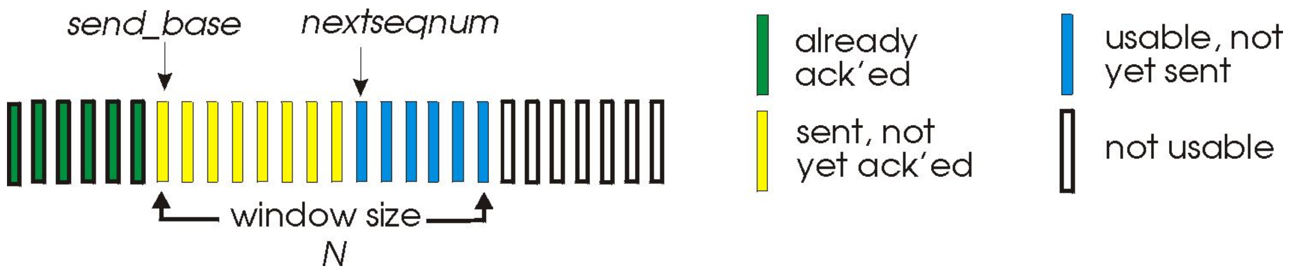
Sliding window technique

- The essence of all sliding window protocols is that at any instant of time, the sender maintains a set of sequence numbers corresponding to frames it is permitted to send.
- These frames are said to fall within the sending window.
- Similarly, the receiver also maintains a receiving window corresponding to the set of frames it is permitted to accept.
- The sender's window and the receiver's window need not have the same lower and upper limits or even have the same size.
- In some protocols they are fixed in size, but in others they can grow or shrink over the course of time as frames are sent and received.

Go-Back-N

Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

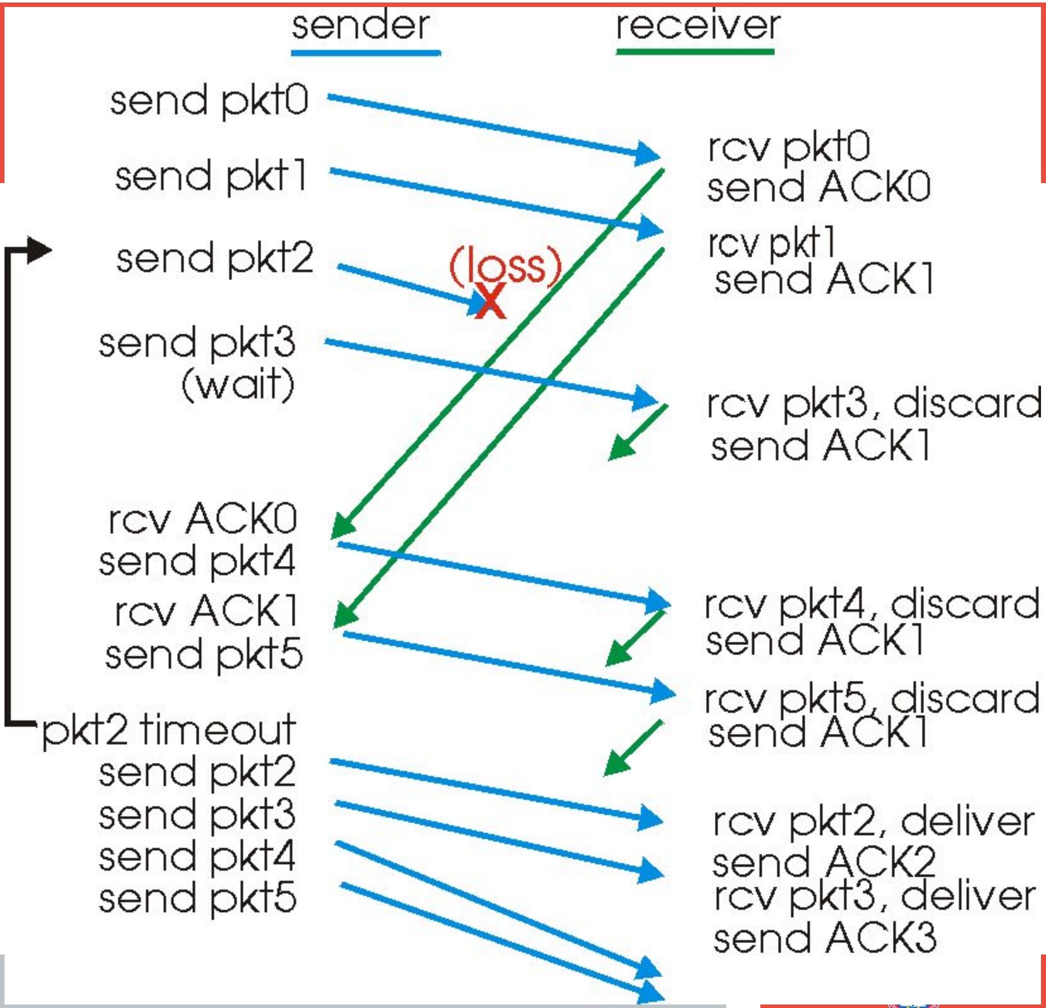
GBN: receiver side

ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **`expected seq num`**

- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #

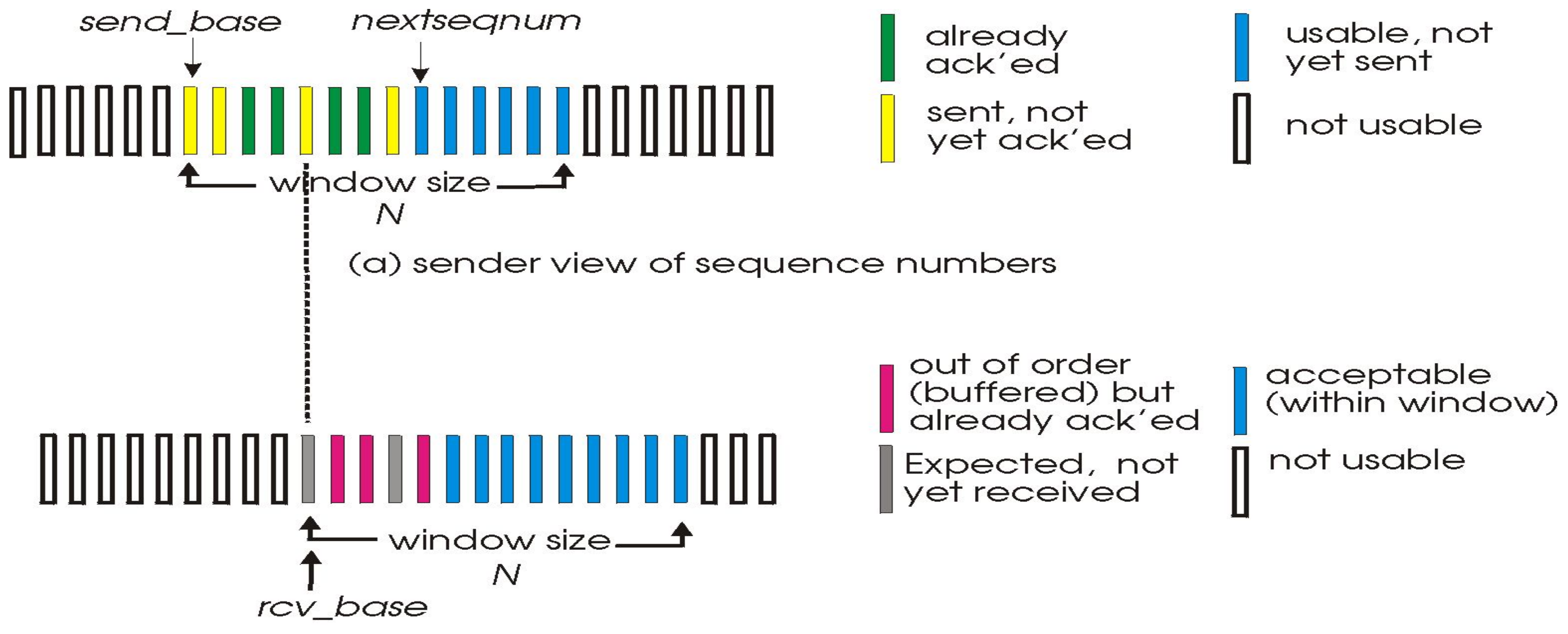
GBN in action



Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  – buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  – sender timer for each unACKed pkt
- sender window
  – N consecutive seq #'s
  – limits seq #s of sent, unACKed pkts

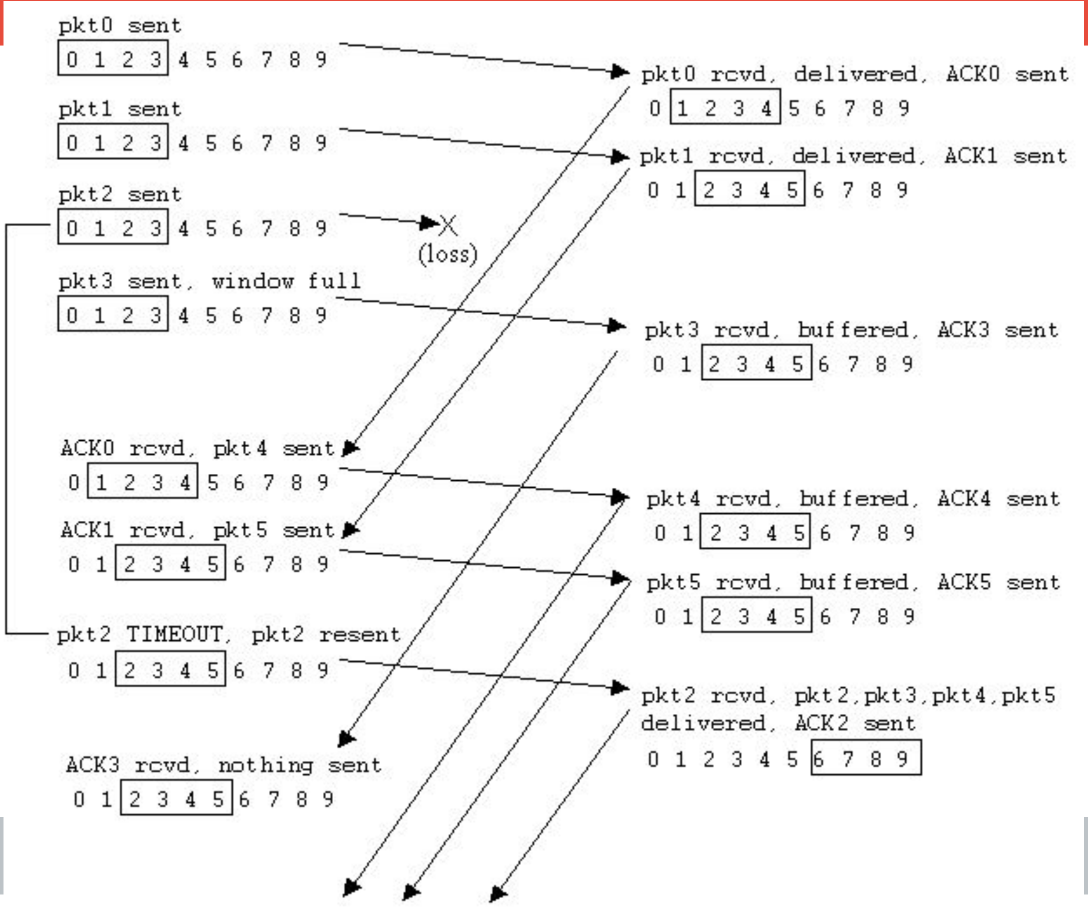
Selective Repeat: sender, receiver, windows



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Selective repeat in action



END.