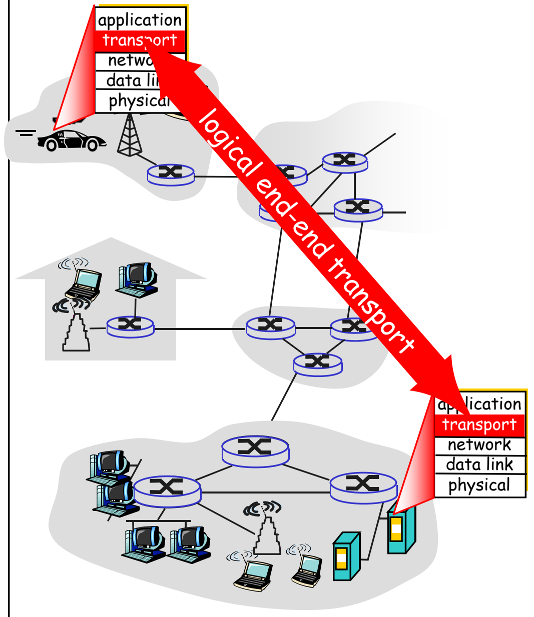
Chapter 3
Transport Layer

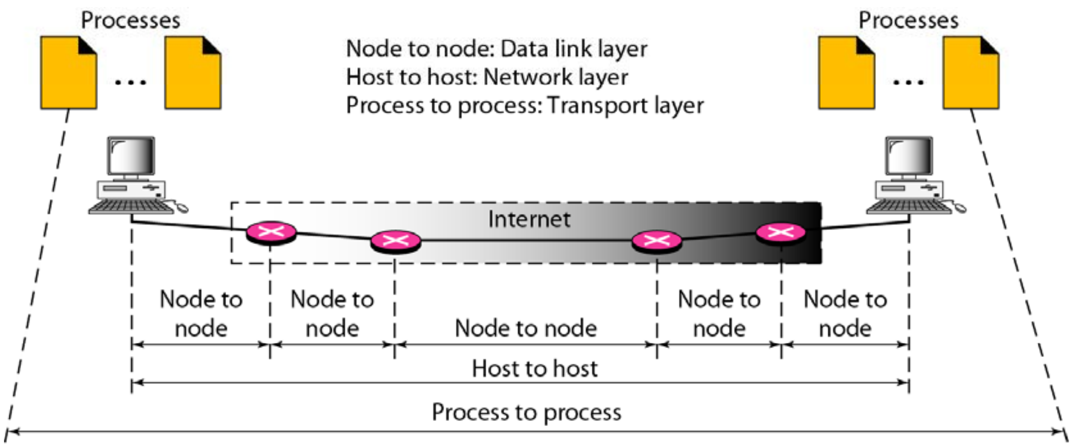
Transport Services and Protocols

- Provide *Logical Communication* between application **processes** running on different hosts
- Transport protocols run in end systems
  - Sender side: breaks application messages into Segments, passes to network layer
  - Receiver side: reassembles segments into Messages, passes to application layer
- more than one transport protocol available to applications
  - Internet: TCP and UDP



Type of Data Deliveries

- *Data link layer:* logical communication between **nodes**
- *Network layer:* logical communication between **hosts**
- *Transport layer:* logical communication between **processes**
  - relies on, enhances, network layer services



Process-to-Process Delivery

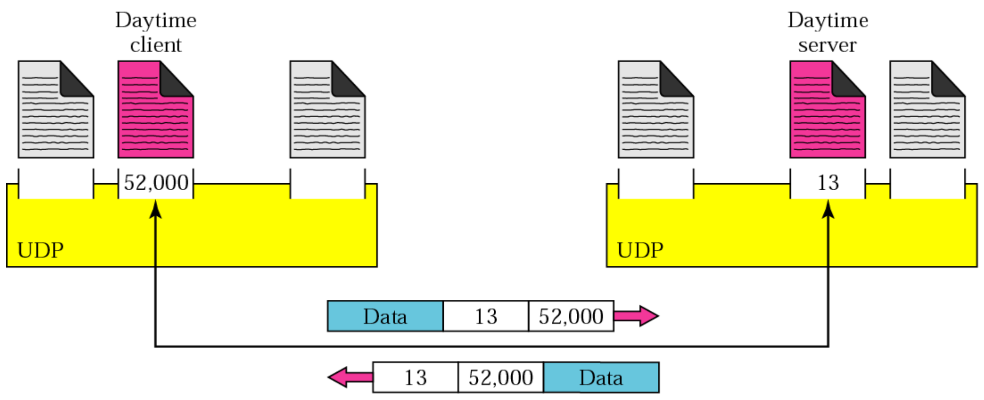
- OS today support both multiuser and multiprogramming environments
  - Computer can run several programs at the same time
- Client-Server Paradigm
- Process on local host (client) needs services from process on remote host (server)
- Both **processes** (client and server) have the same name, ex. ftp, DNS
- For communication, we must define
  - **Local** Host
  - **Local** Process
  - **Remote** Host
  - **Remote** Process

Addressing

- We need **address**, when we need to deliver something to one specific destination among many
  - Data link layer : MAC address
  - Network Layer : IP address
  - Transport Layer : Port Number
- Internet model, port number are 16-bit integer between 0 and 65,535
- **Client process** defines itself with port number, chosen randomly called *ephemeral (temporary) port number*
- **Server process** must also define itself with port number, which cannot chosen randomly

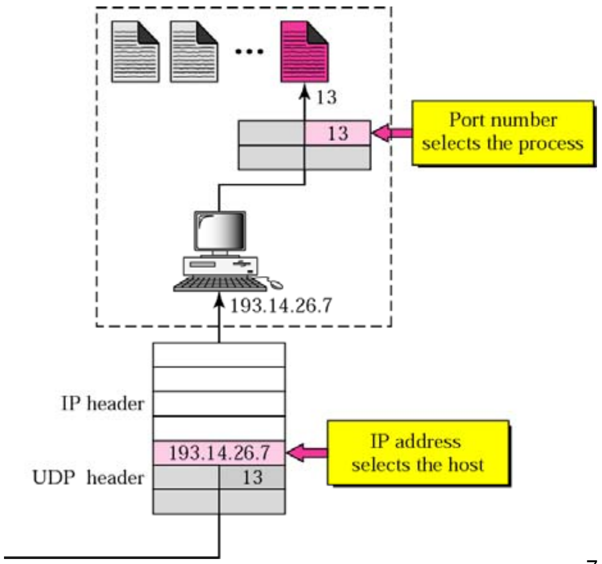
Addressing (cont.)

- For Server, Internet has decided to use universal port number called *well-known port numbers*
- Every client process knows well-known port number

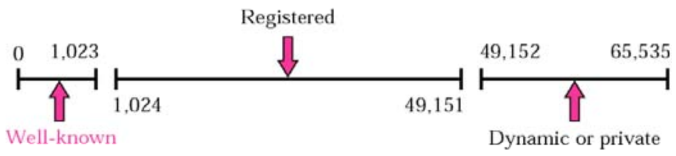


Addressing (cont.)

- It should be clear that *IP addresses* and *Port Numbers* play different roles in selecting final destination of data
- *Destination IP address* defines **host** among different hosts in the world
- *Port number* defines one of **process** on this particular host



IANA Ranges (cont.)

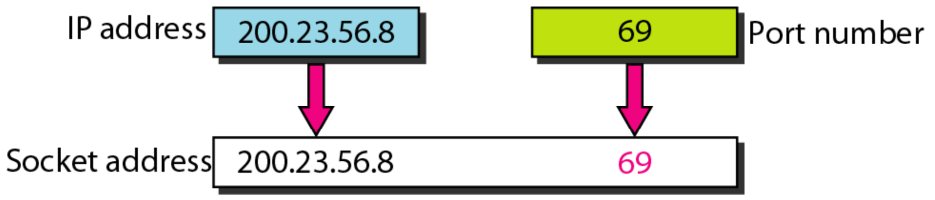


(http://www.iana.com)

- Well-known Ports
  - ranging from **0 to 1023**
  - Are assigned and controlled by IANA
- Registered Ports
  - Ranging from **1,024 to 49,151**
  - Not assigned and controlled by IANA
  - They can only be registered with IANA to prevent duplication
- Dynamic (or private) Ports
  - Ranging from **49,152 to 65,535**
  - Neither controlled nor registered
  - They can be used by any process
  - These are ***ephemeral ports** or **temporary ports***

Socket Address

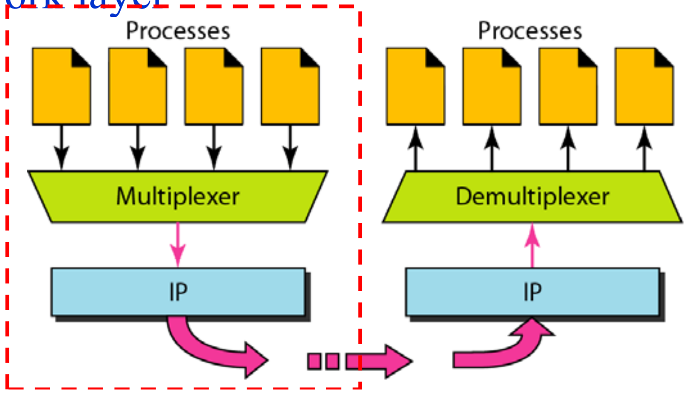
- *Socket Address* is combination of IP Address and Port Number



- Transport layer protocol needs a pair of socket addresses : *client socket address* and *server socket address*
- These four pieces of information are part of **IP header** and **Transport layer protocol header**
- IP header contains IP address; UDP or TCP header contains port number

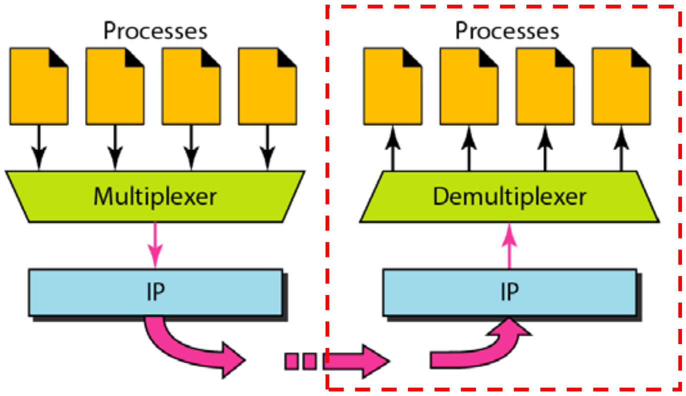
Multiplexing and Demultiplexing

- **Multiplexing**
  - There may be several processes that need to send packet
  - Protocol accepts messages from different processes, differentiated by their port number
  - After adding header, transport layer passes packet to network layer



Multiplexing and Demultiplexing (cont.)

- **Demultiplexing**
  - Transport layer receives datagrams from network layer
  - After error checking and dropping of header, transport layer delivers each message to appropriate process based on port number



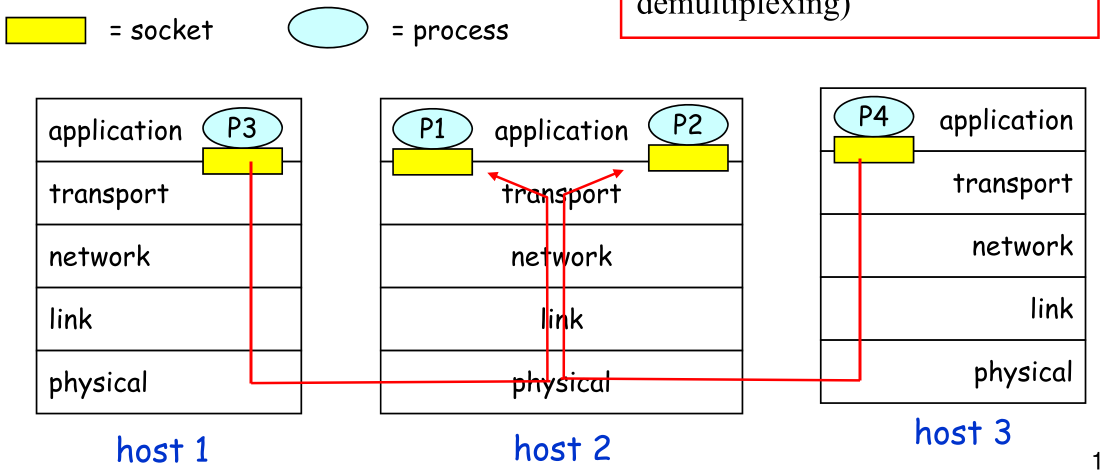
Multiplexing/Demultiplexing (cont.)

Demultiplexing at receiving host:

delivering received segments to correct socket

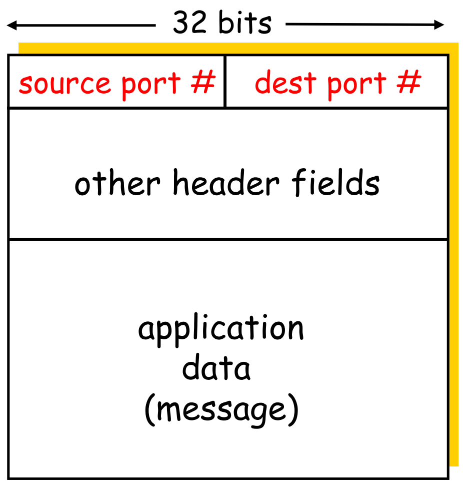
Multiplexing at sending host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How **demultiplexing** works

- Host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses* & *port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Connectionless **demultiplexing**

- Create sockets with port numbers:

DatagramSocket mySocket1 =
  new DatagramSocket(49152);
DatagramSocket mySocket2 =
  new DatagramSocket(49153);

- UDP socket identified by two-tuple:

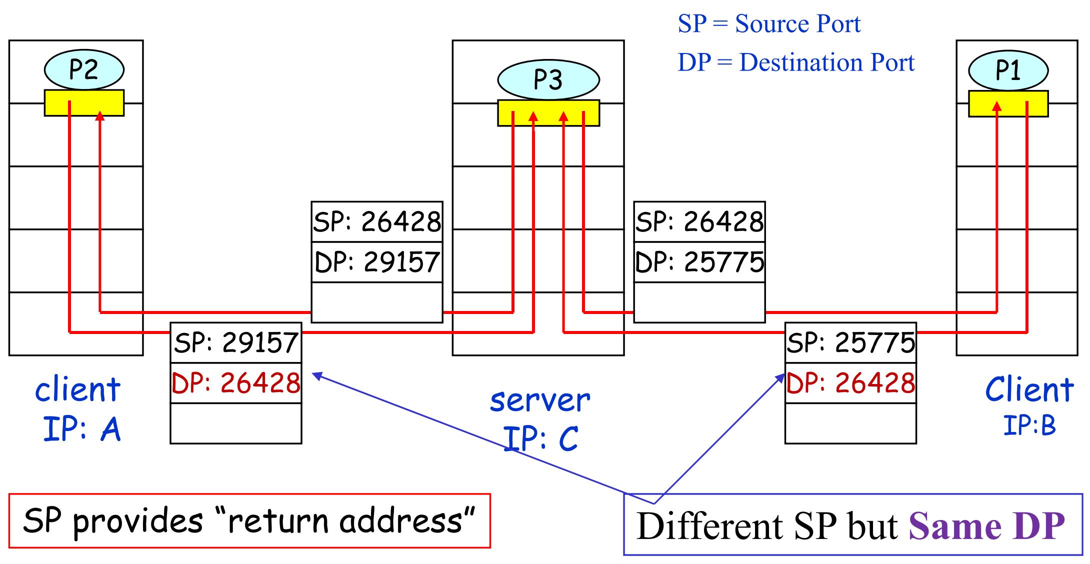
$\Downarrow$

(destination IP address, destination Port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- *IP datagrams with **different source IP addresses** and/or source port numbers directed to **same socket***

Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(26428);
```

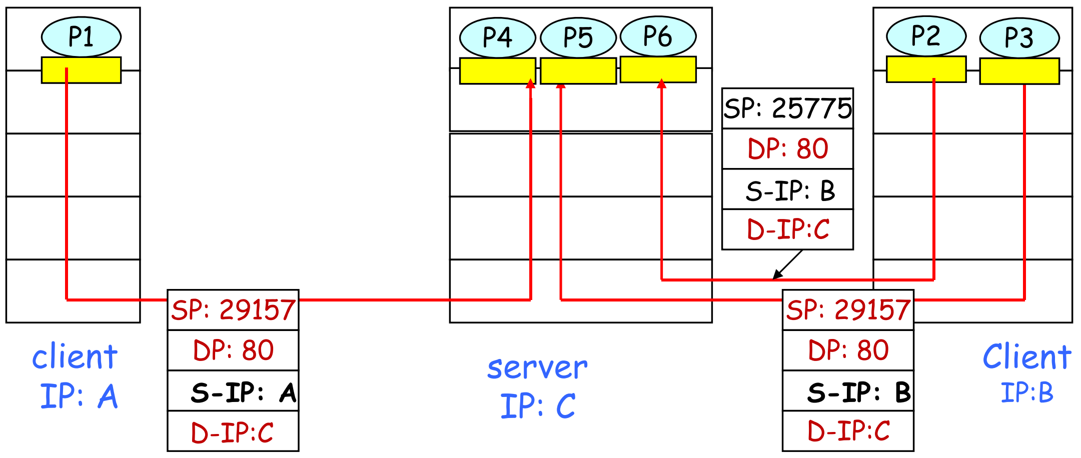


Connection-oriented demux

- TCP socket identified by 4-tuple:
  - Source IP address
  - Source Port number
  - Destination IP address
  - Destination Port number
- Receiver host uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux (cont)



Connectionless versus Connection-Oriented Service

- Connectionless Service
  - Packet are sent from one party to another with no need for connection establishment or connection release
  - Packet are not numbered
    - They may be delayed or lost or may arrive out of sequence
  - No acknowledgment
  - **UDP** is connectionless

Connectionless versus Connection-Oriented Service

- Connection-Oriented Service
  - Connection is first established between sender and receiver
  - Data are transferred; at the end, connection is released
  - Acknowledgement is needed
  - **TCP** is connection-oriented protocol

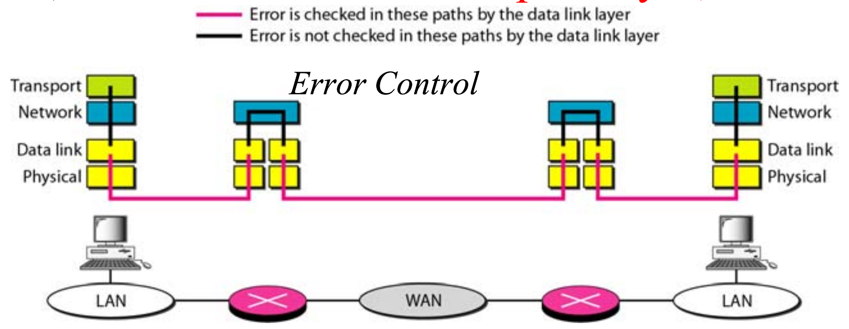
Reliable versus Unreliable

- Transport layer service can be reliable or unreliable
- Reliable transport layer protocol
  - implementing flow and error control at transport layer
  - Slower and more complex service
- Unreliable transport layer protocol
  - No flow and error control
  - Faster service

Reliable versus Unreliable (cont.)

Question?

- If data link layer is reliable and has flow and error control, do we need this at transport layer, too?

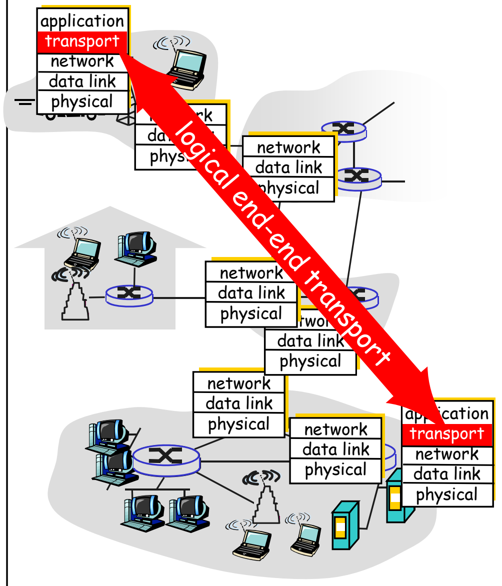

Error is checked in these paths by the data link layer
Error is not checked in these paths by the data link layer
Error Control
Transport
Network
Data link
Physical
Transport
Network
Data link
Physical
LAN
WAN
LAN

Answer

- Reliability at data link layer is between two nodes
- We need reliability between two ends
- Because network layer in Internet is unreliable (best-effort delivery), we need to implement reliability at transport layer

Internet Transport-layer Protocols

- **Reliable, in-order delivery : (Transmission Control Protocol, TCP)**
  - Connection setup
  - Flow control
  - Error control
  - Congestion control
- **Unreliable, unordered delivery: (User Datagram Protocol, UDP)**
  - no-frills extension of "best-effort" IP
  - No connection setup
  - No flow control
  - No error control
  - No congestion control
- services not available:
  - delay guarantees
  - bandwidth guarantees



UDP: User Datagram Protocol [RFC 768]

- UDP does not add anything to service of Internet Protocol except to provide process-to-process communication
- Unreliable transport Protocol (best effort service), UDP segments may be:
  - lost
  - delivered out of order to application
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

UDP: User Datagram Protocol

Why is there a UDP?

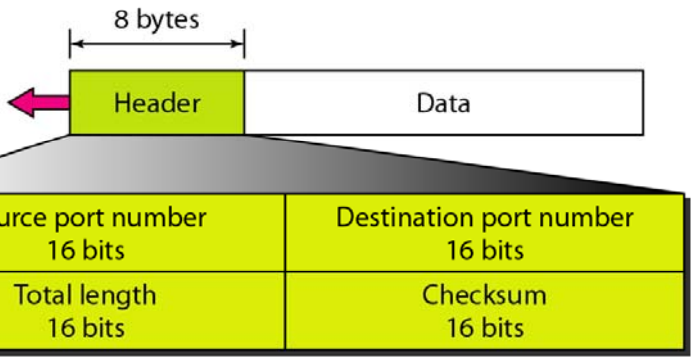
- *No connection establishment* (which can **add delay**)
- *Simple protocol using minumum of overhead* : no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

Well-Known Ports for UDP

Port	Protocol	Description
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Nameserver	Domain Name Service
67	BOOTPs	Server port to download bootstrap information
68	BOOTPc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
111	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

UDP Segment Format

- UDP segments have fixed-size header of **8 bytes**



- **Total length** defines total length of UDP segment, <u>*header plus data*</u>
- **Checksum** is used to detect error over the entire segment (header plus data)

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- Checksum: addition (1's complement sum) of segment contents
- Sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.

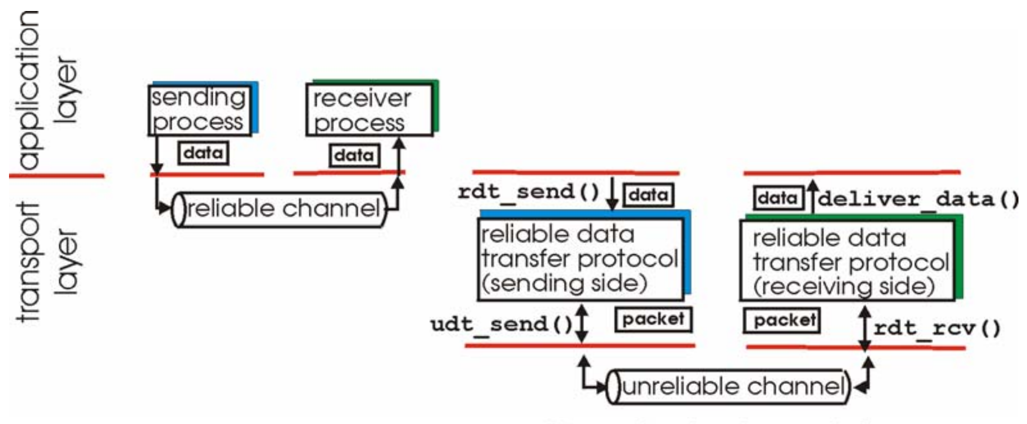
Internet Checksum Example

- Note
  - When adding numbers, a carry out from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

```
                1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
                1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
              ----------------------------------
wraparound    1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
              ----------------------------------
       sum      1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
  checksum      0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

Principles of Reliable Data Transfer

- important in application, transport, link layers



(a) provided service (b) service implementation

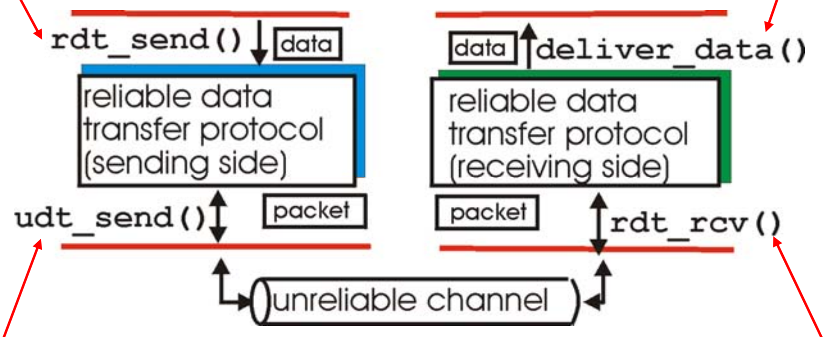
- characteristics of unreliable channel will determine complexity of **R**eliable **D**ata **T**ransfer protocol (**rdt**)

Reliable Data Transfer: getting started

`rdt_send()`: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

`deliver_data()`: called by **`rdt`** to deliver data to upper

Sender side


rdt_send()
data
reliable data transfer protocol (sending side)
udt_send()
packet
data
deliver_data()
reliable data transfer protocol (receiving side)
packet
rdt_rcv()
unreliable channel

Receiver side

`udt_send()`: called by rdt, to transfer packet over unreliable channel to receiver

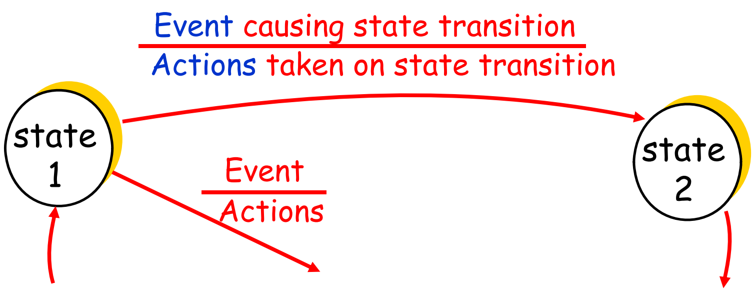
`rdt_rcv()`: called when packet arrives on rcv-side of channel

Reliable Data Transfer: getting started

We'll:

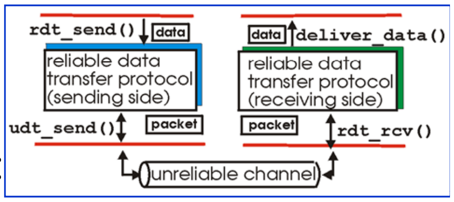
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use **F**inite **S**tate **M**achines (FSM) to specify sender, receiver

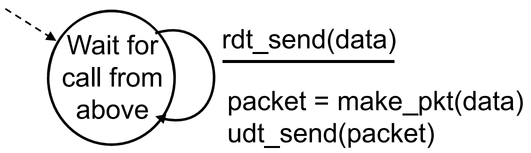
state: when in this "state" next state uniquely determined by next event


Event causing state transition
Actions taken on state transition
state 1
state 2
Event
Actions

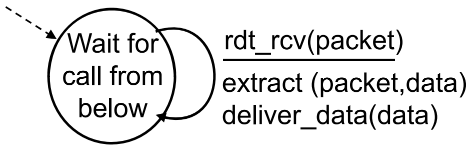
Rdt1.0: reliable transfer over a **reliable channel**

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel


rdt_send()
data
deliver_data()
reliable data transfer protocol (sending side)
reliable data transfer protocol (receiving side)
udt_send()
packet
rdt_rcv()
unreliable channel



sender

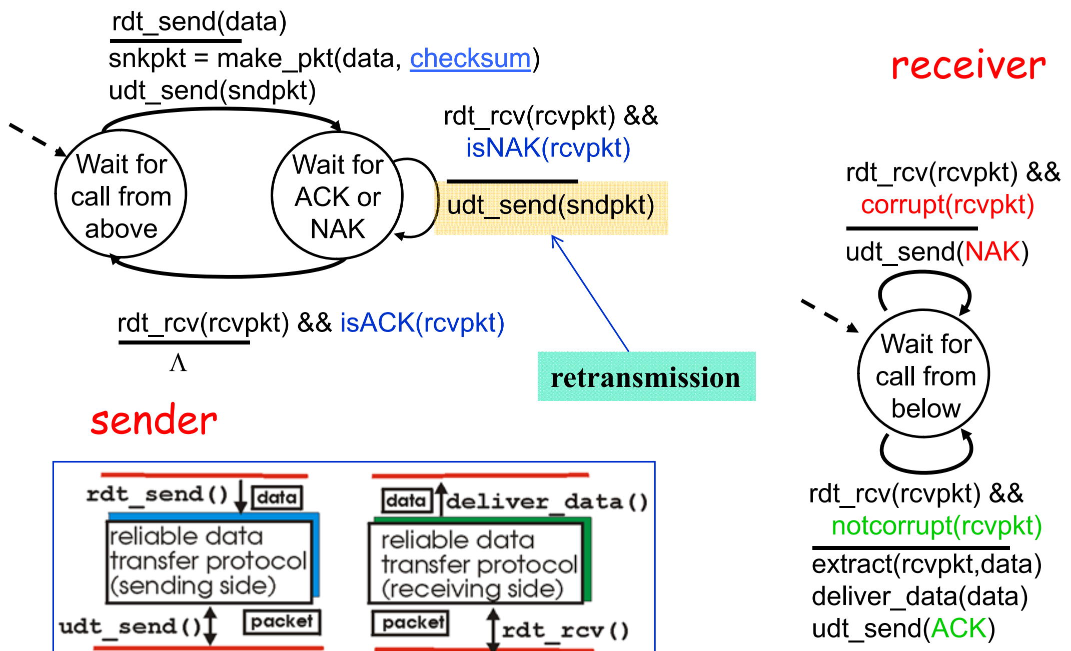


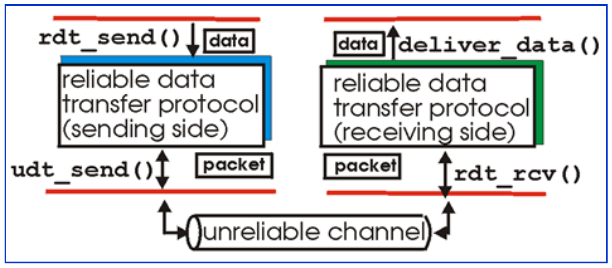
receiver

Rdt2.0: channel with **bit errors**

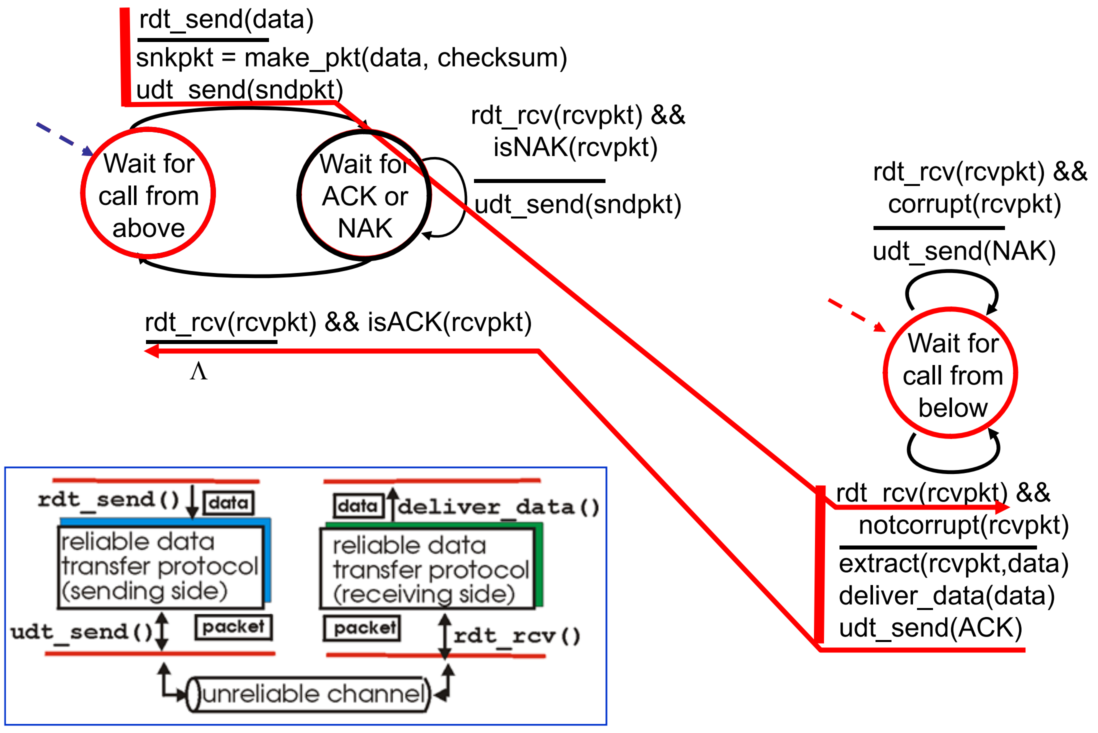
- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
- *Q*uestion: how to recover from errors:
  - *ACKnowledgements (ACKs):* receiver explicitly tells sender that packet received OK
  - *Negative ACKnowledgements (NAKs):* receiver explicitly tells sender that packet has errors
  - sender retransmits packet on receipt of NAK
- new mechanisms in **rdt2.0** (beyond **rdt1.0**):
  - Error Detection
  - Receiver Feedback:
    - control messages (ACK,NAK) receiver->sender

rdt2.0: FSM specification



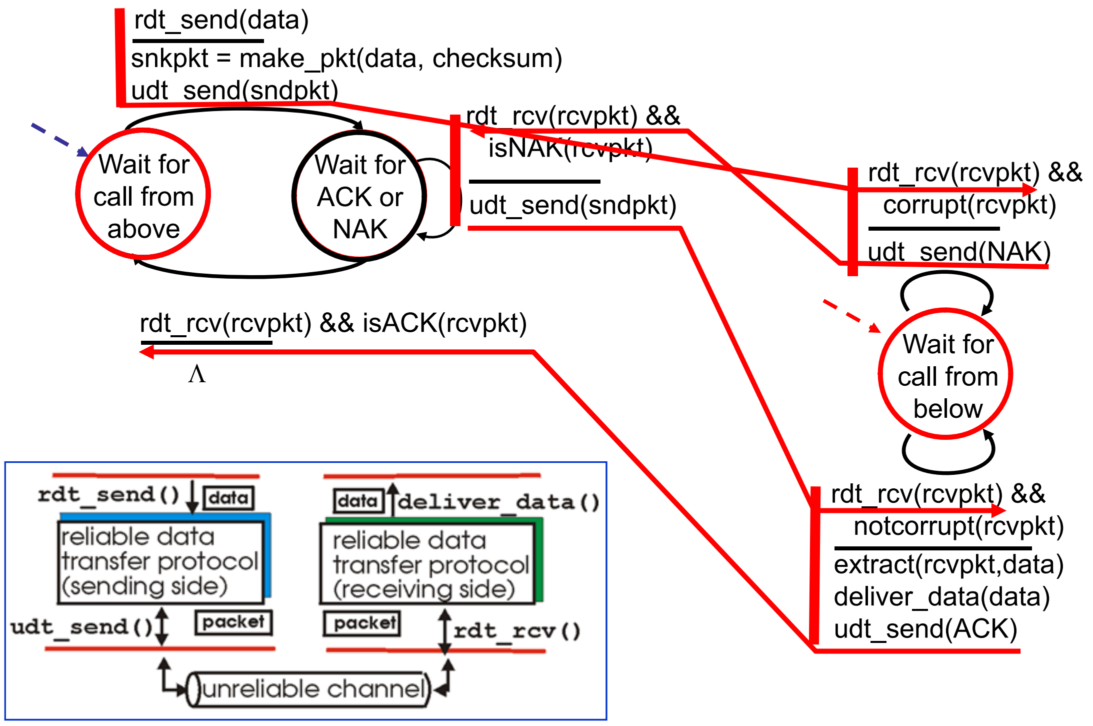

rdt_send()
data
reliable data transfer protocol (sending side)
udt_send()
packet
data
deliver_data()
reliable data transfer protocol (receiving side)
packet
rdt_rcv()
unreliable channel

rdt2.0: operation with no errors



rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
Wait for call from below
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)
rdt_send()
data
deliver_data()
reliable data transfer protocol (sending side)
reliable data transfer protocol (receiving side)
udt_send()
packet
rdt_rcv()
unreliable channel

rdt2.0: error scenario



rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
Wait for call from below
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)
rdt_send()
data
deliver_data()
reliable data transfer protocol (sending side)
reliable data transfer protocol (receiving side)
udt_send()
packet
rdt_rcv()
unreliable channel

rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- Sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

Handling duplicates:

- sender retransmits current pkt if ACK/NAK garbled
- sender adds *Sequence Number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait
Sender sends one packet, then waits for receiver response

rdt3.0: channels with **errors** *and* **loss**

New assumption:
underlying channel can also lose packets (data or ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help

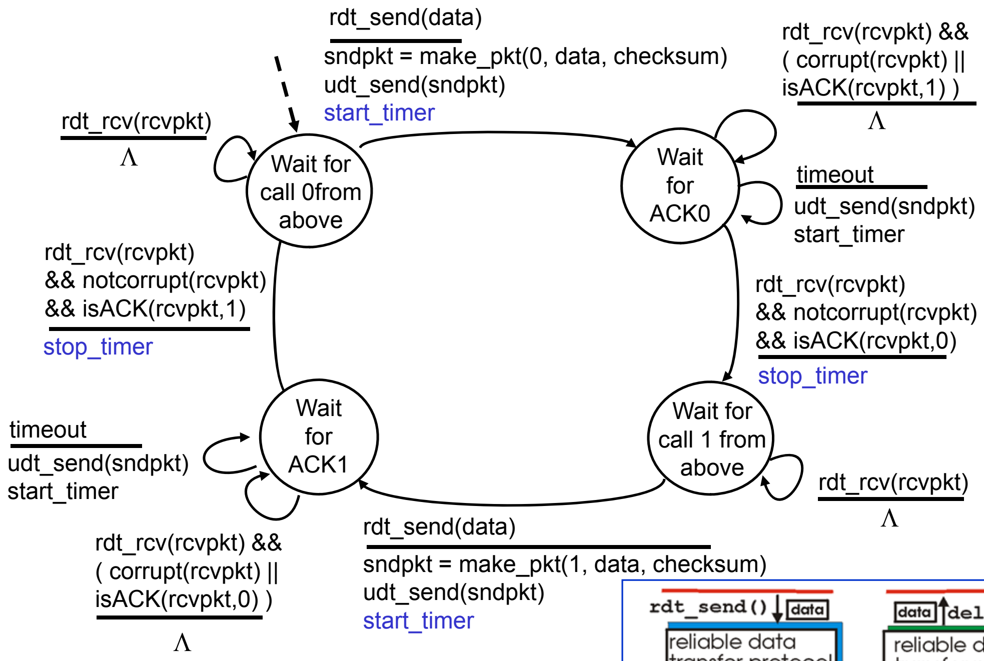
↓

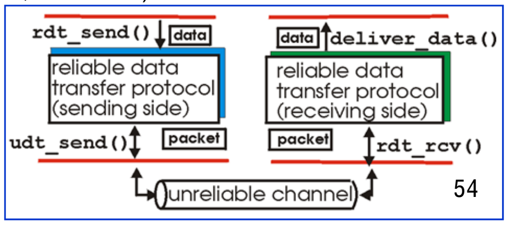
But not enough

Approach: sender waits "reasonable" amount of time for ACK

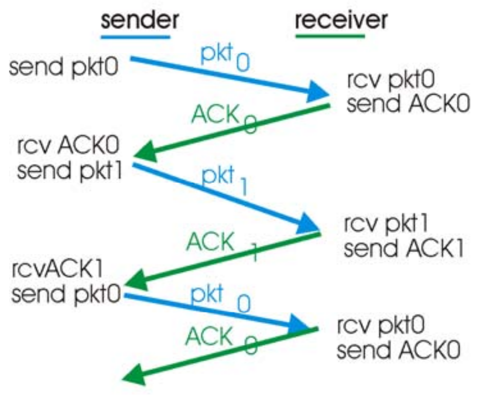
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0 sender

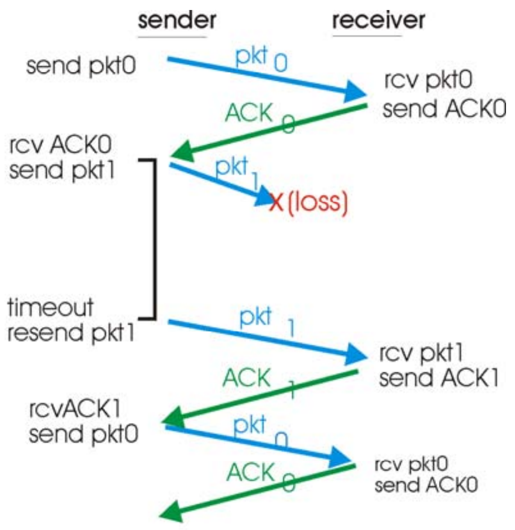

rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for call 0from above
Wait for ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
timeout
udt_send(sndpkt)
start_timer
Wait for ACK1
Wait for call 1 from above
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer


rdt_send()
data
reliable data transfer protocol (sending side)
udt_send()
packet
data
deliver_data()
reliable data transfer protocol (receiving side)
packet
rdt_rcv()
unreliable channel

rdt3.0 in action

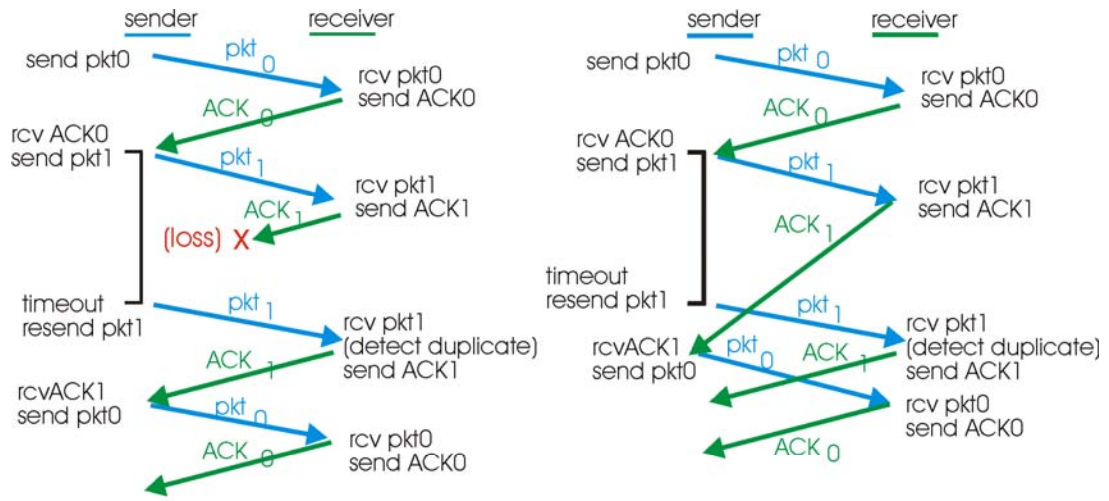


(a) operation with no loss



(b) lost packet

rdt3.0 in action

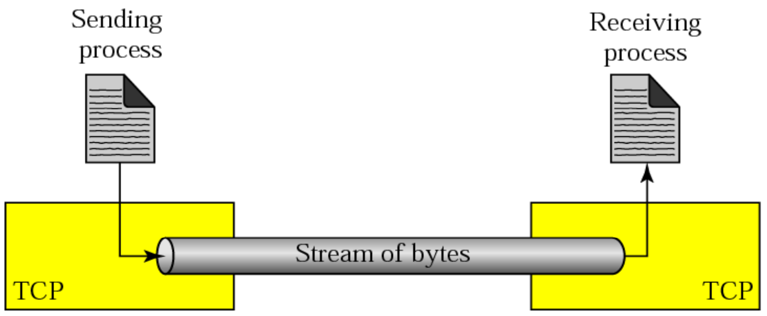


(c) lost ACK

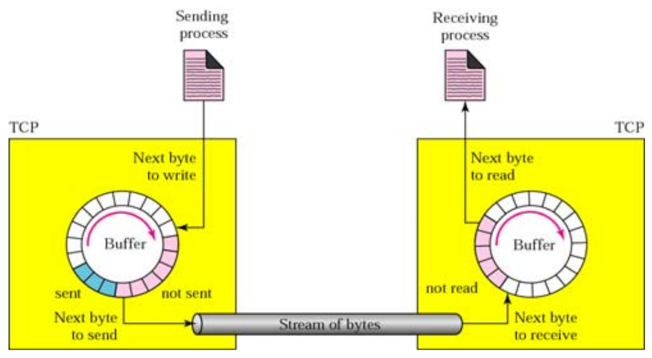
(d) premature timeout

TCP Services

- Process-to-Process Communication
- Stream Delivery Service
  - TCP allows sending process to delivery data as a stream of bytes and allows receiving process to obtain data as a stream of bytes
  - TCP creates an environment in which two process seem to be connected that carries data across Internet

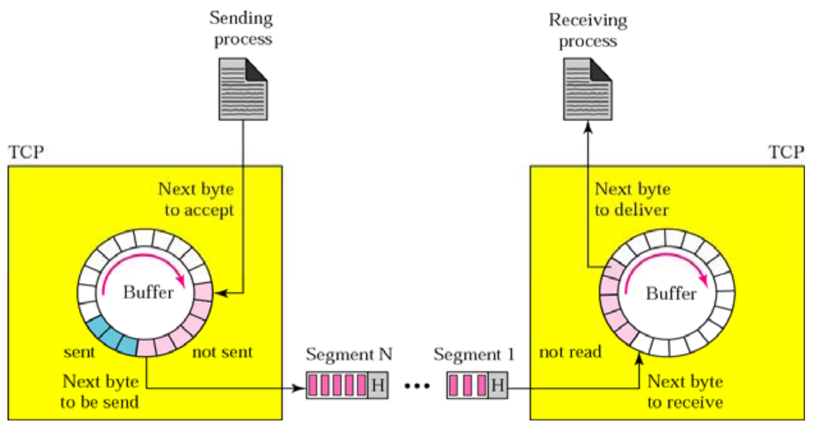


TCP Services :
Stream Delivery Service (continued)



- Sending and Receiving Buffers
  - Because sending and receiving processes may not write or read data at the same speed, TCP needs buffers for storage
  - There are two buffers, sending buffer and receiving buffer
  - Buffers are also necessary for *flow and error control* mechanisms used by TCP

TCP Services : Stream Delivery Service (continued)

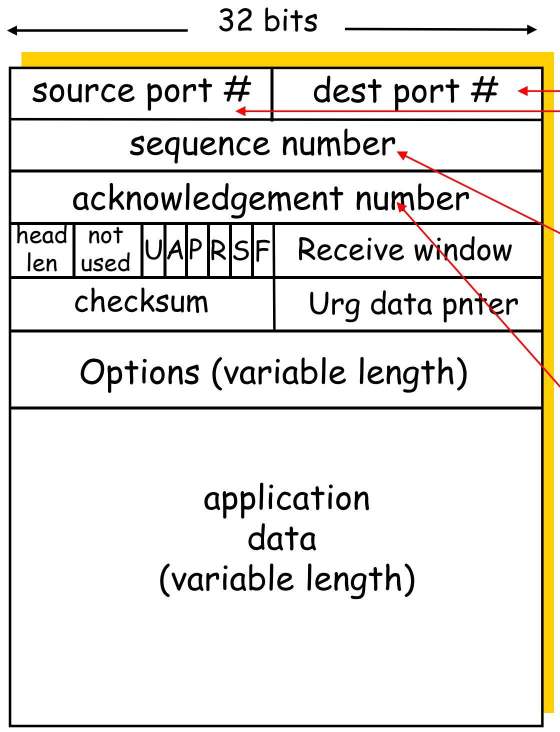


- Segment
  - TCP groups a number of bytes together into a packet called **Segment**
  - TCP adds header to each segment (for control purposes) and delivers segment to IP layer for transmission
  - Segment are encapsulated in IP datagram and transmitted
  - The entire operation is transparent to receiving process

TCP Services (continued)

- TCP offers full-duplex service
  - Data can flow in both directions at the same time
  - Each TCP has sending and receiving buffer and segments move in both direction
- Connection-Oriented Service
  - Before sending and receiving data, following processes occur
    - Two TCP establish a connection between them
    - Data are exchanged in both directions
    - Connection is terminated
- Reliable Service
  - TCP is reliable transport protocol by using acknowledgement mechanism

TCP Segment Structure

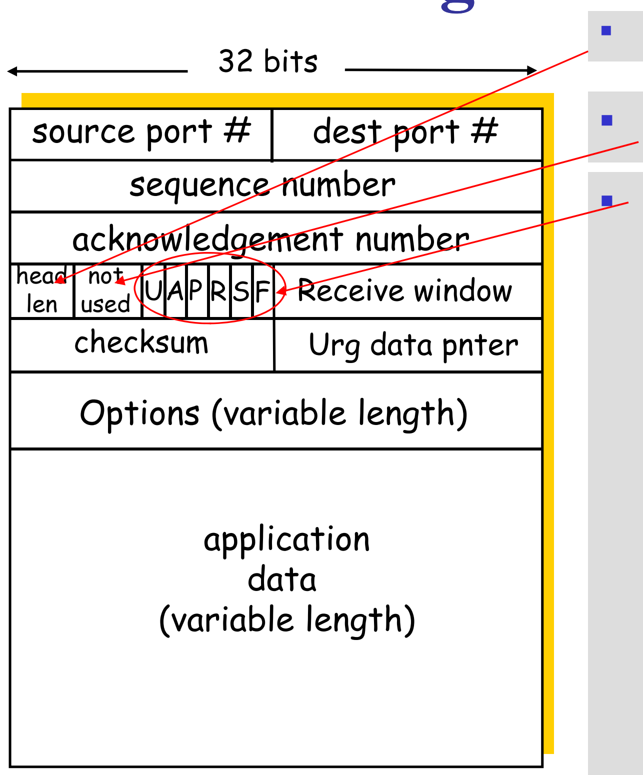


- **Source and Destination ports** - this identifies the upper layer applications using the connection

- **Sequence Number - this 32-bit number ensures that data is correctly sequenced.**
- **Each byte of data is assigned a sequence number.**

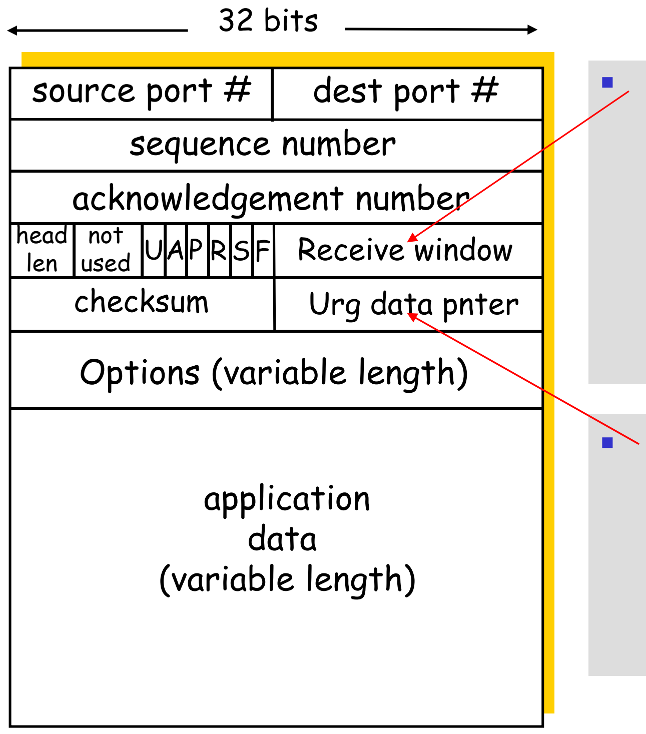
- **Acknowledgement Number - this 32-bit number indicates next sequence number that sending device is expecting from the other station.**

TCP Segment Structure



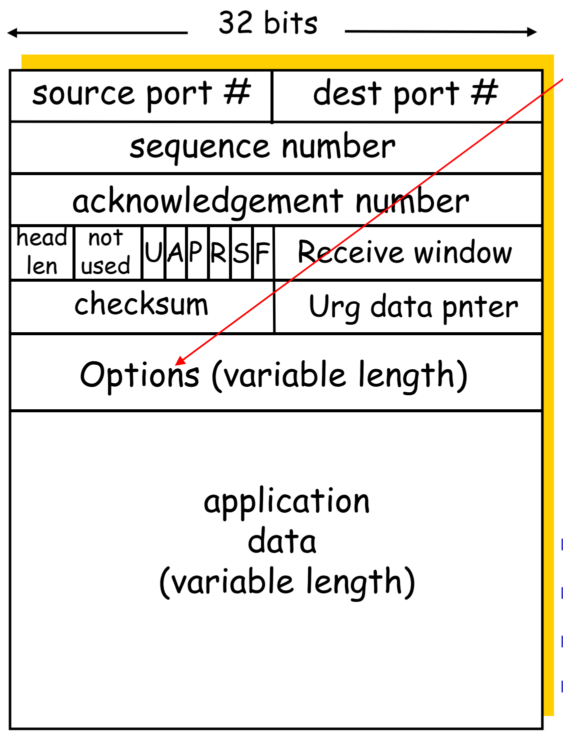
- **Header Length(4 bits)- No.of 4 byte words in TCP header length 20-60 bytes**
- **Reserved (6 bits) - always set to 0** (for future use)
- **Code bits (6 bits)**
  - URG-value of Urgent Pointer field is valid
  - ACK-value of acknowledgement field is valid
  - PSH-to let receiving TCP know that segment includes data that must be delivered to receiving application program as soon as possible and not to wait for more data to come (generally not used)
  - RST-Reset connection
  - SYN-Synchronize sequence number during connection
  - FIN-Terminate connection

TCP Segment Structure

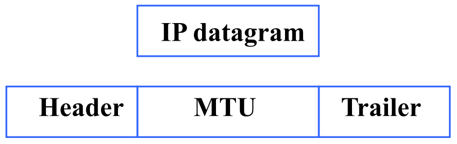


- **Receive window (16 bits)**
  - **Number of bytes receiver willing to accept**
  - **indicates range of acceptable sequence numbers beyond last segment that was successfully received.**
  - **It is allowed number of octets that *sender* of ACK is willing to accept before acknowledgement.**
- **Urgent Data Pointer (16 bits)-**
  - **shows end of urgent data so that interrupted data streams can continue**
  - **When URG bit is set, data is given priority over other data streams**

TCP Segment Structure



- **Option (0-32 bits)-** mainly only TCP Maximum Segment Size (MSS) sometimes called Maximum Window Size or Send Maximum Segment Size (SMSS).
  - Segment is series of data bytes within TCP header.



Maximum Transfer Unit : Max. length of data that can be encapsulated in a frame

- **Token Ring (16 Mbps) – 17,914 bytes**
- **Token Ring (4 Mbps) - 4,464 bytes**
- **Ethernet - 1,500 bytes**
- **Point-to-Point Protocol (PPP)–296 bytes**

TCP Features

- ***To provide services mentioned in previous section, TCP has several features as follows :***
  - ***Numbering System***
  - ***Flow Control***
  - ***Error Control***
  - ***Congestion Control***

TCP Features : Numbering System

- Although TCP keeps track of segments being transmitted or received
- **No field for segment number value** in segment header
- Instead, there are two field called
  - Sequence Number
  - Acknowledgement Number

  $\Big\}$ refer to byte number and not segment number
- Bytes of data being transferred in each connection are numbered by TCP
- Numbering starts with a randomly generated number. $(0 – 2^{32}-1)$
  - Ex. Random generate No. = 1,057, for 6,000 bytes data
  - Range of data is start from 1,057 to 7,056

TCP Features : Numbering System

- What is Sequence Number
  - After bytes have been numbered, TCP assigns a sequence number to each segment that is being sent
- Sequence number for each segment is number of the first byte carried in that segment

TCP Features : Numbering System

- Example :
  - Suppose TCP connection is transferring file of 5,000 bytes.
  - The first byte is numbered 10001.
  - What are sequence numbers for each segment if data is sent in five segments, each carrying 1000 bytes?

10,001-11,000	11,001-12,000	12,001-13,000	13,001-14,000	14,001-15,000

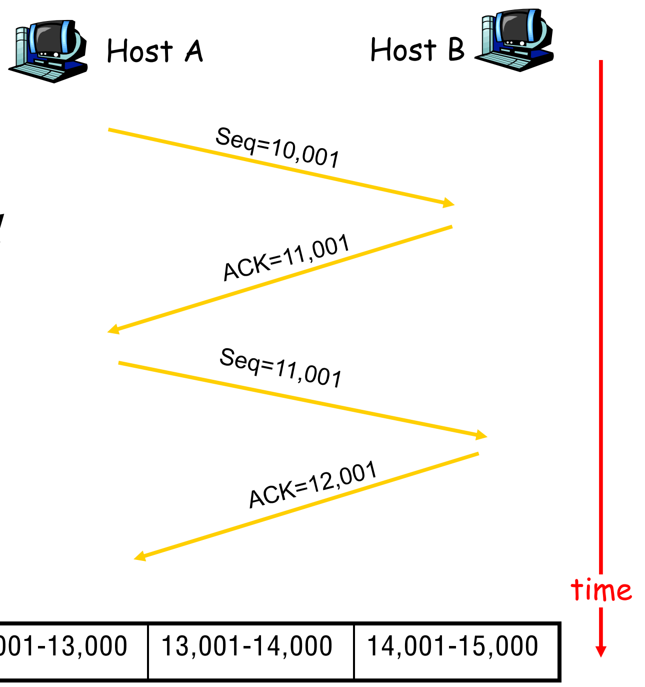
|---1,000 bytes----|

Segment 1 ➡	***Sequence Number: 10,001 (range: 10,001 to 11,000)***
Segment 2 ➡	***Sequence Number: 11,001 (range: 11,001 to 12,000)***
Segment 3 ➡	***Sequence Number: 12,001 (range: 12,001 to 13,000)***
Segment 4 ➡	***Sequence Number: 13,001 (range: 13,001 to 14,000)***
Segment 5 ➡	***Sequence Number: 14,001 (range: 14,001 to 15,000)***

Value of sequence number field of segment defines number of the first data byte contained in that segment

TCP Features : Numbering System

- *What is **Acknowledgment Number**?*
  - *Value of acknowledgment field in segment defines number of next byte a party expects to receive*
  - *Acknowledgment number is cumulative*



10,001-11,000	11,001-12,000	12,001-13,000	13,001-14,000	14,001-15,000

|---1,000 bytes----|

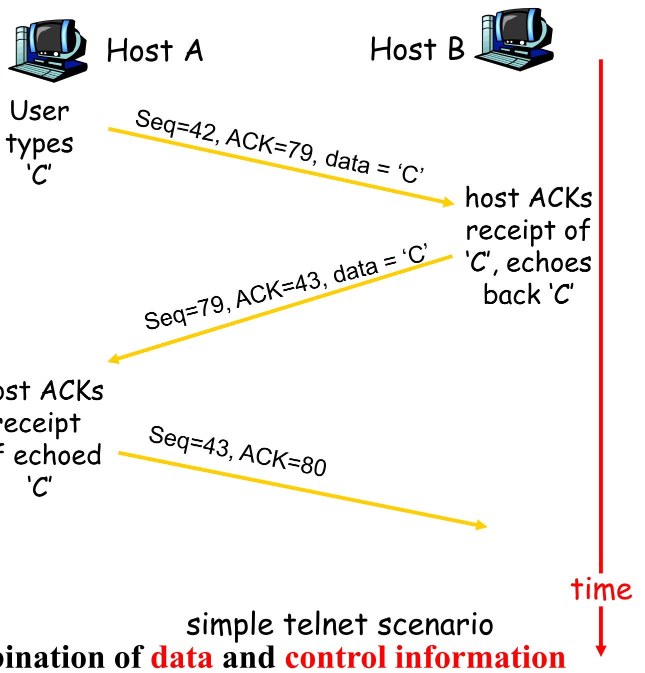
TCP Sequence Number and Acknowledgement Number (ACK)

Seq. #'s:

- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK


Host A
Host B
User types 'C'
Seq=42, ACK=79, data = 'C'
host ACKs receipt of 'C', echoes back 'C'
Seq=79, ACK=43, data = 'C'
host ACKs receipt of echoed 'C'
Seq=43, ACK=80
time
simple telnet scenario

- **When Segment carries a combination of data and control information called "Piggybacking"**

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- **longer than RTT**
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

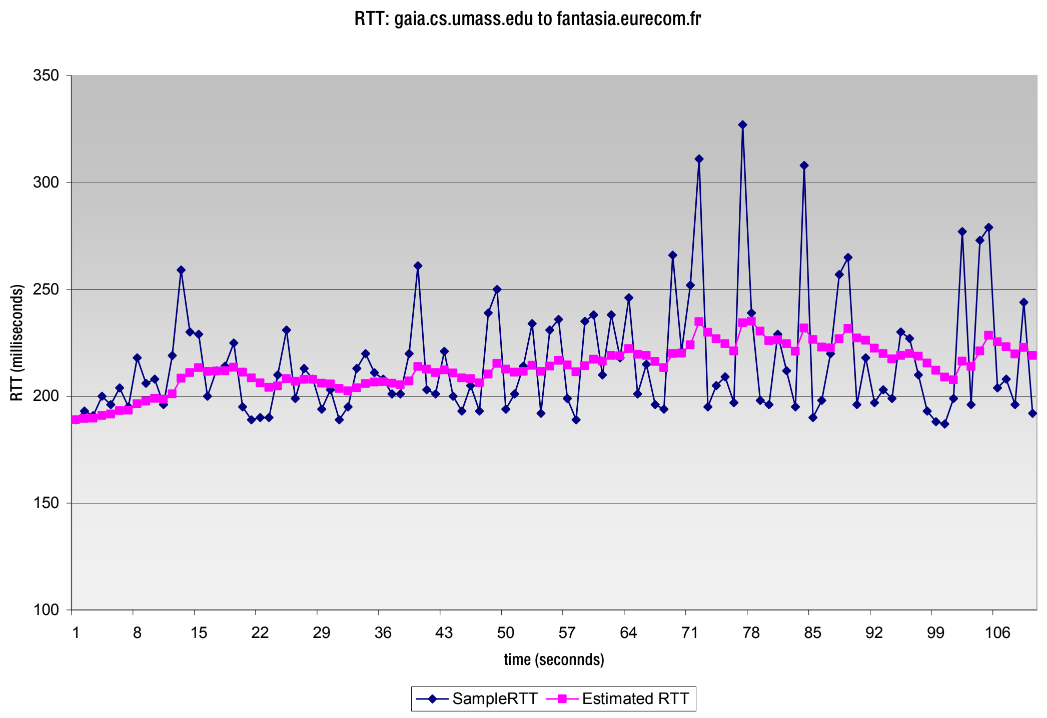
- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want **estimated RTT** "smoother"
  - average several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$ $[(1-\alpha) = 0.875]$

Example RTT estimation:



TCP Round Trip Time and Timeout

- Setting the timeout
- **EstimtedRTT** plus "safety margin"
  - large variation in **EstimatedRTT ->** larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

TCP reliable data transfer

- TCP creates reliable data transfer (rdt) service on top of IP's unreliable service
- Pipelined segments
- Cumulative acknowledges
- TCP uses single retransmission timer

- Retransmissions are triggered by:
  - **Timeout events**
  - Duplicate acknowledges
- Initially consider simplified TCP sender:
  - ignore duplicate acknowledges
  - ignore flow control, congestion control

TCP sender events:

Data received from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: TimeOutInterval

Timeout:

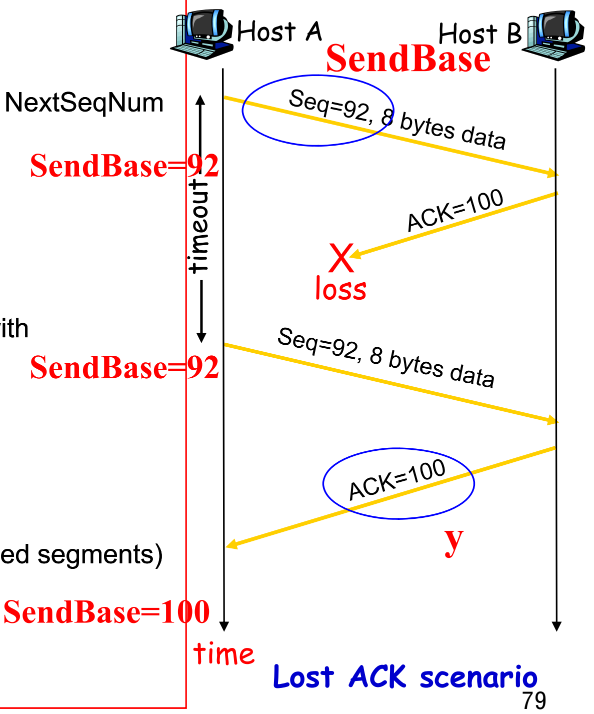
- retransmit segment that caused timeout
- restart timer

Ack received:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

TCP sender (simplified)

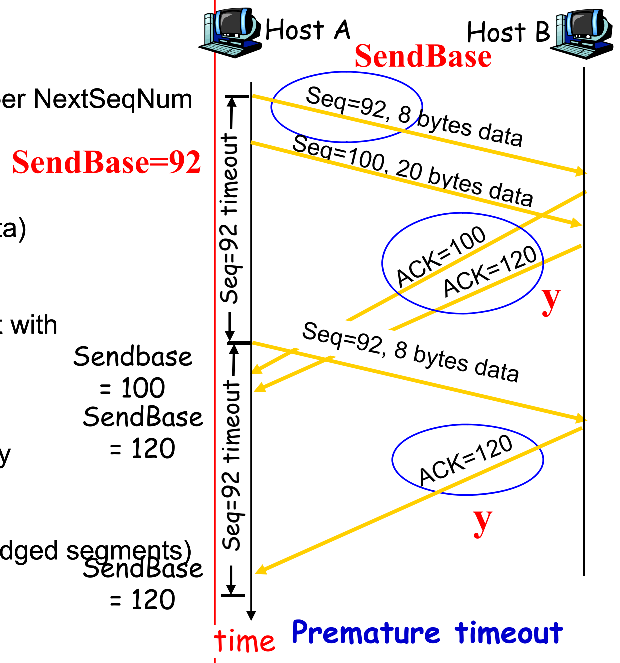
```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
            smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
                start timer
          }

} /* end of loop forever */
```



Lost ACK scenario

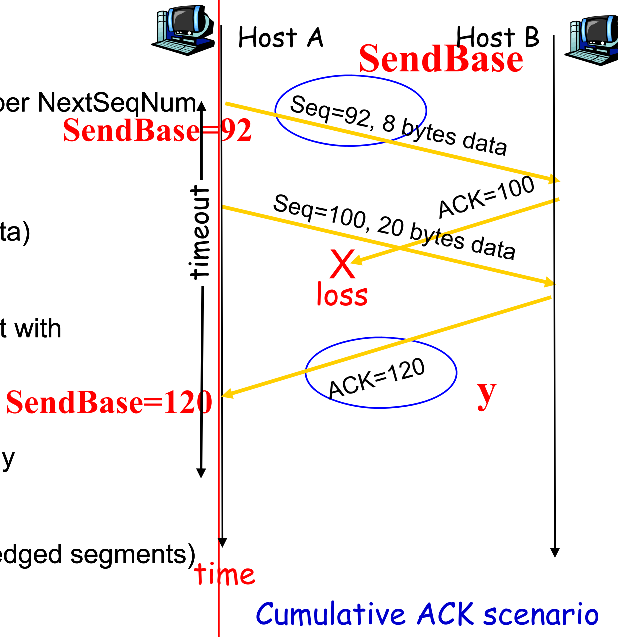
TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
            smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
                start timer
          }

} /* end of loop forever */
```



TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
            smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
                start timer
          }

} /* end of loop forever */
```



Fast Retransmit

- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.

- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - Fast Retransmit: resend segment before timer expires

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
             }
          else {
               increment count of dup ACKs received for y
               if (count of dup ACKs received for y = 3) {
                   resend segment with sequence number y
                }
```

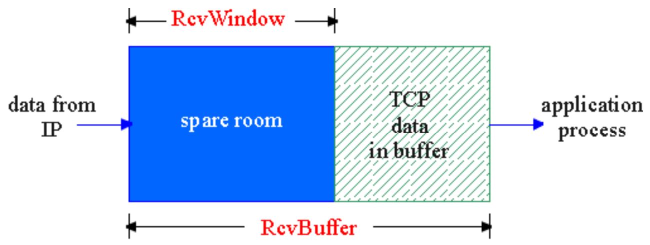
a duplicate ACK for already ACKed segment

fast retransmit

TCP Flow Control

- receive side of TCP connection has a receive buffer:

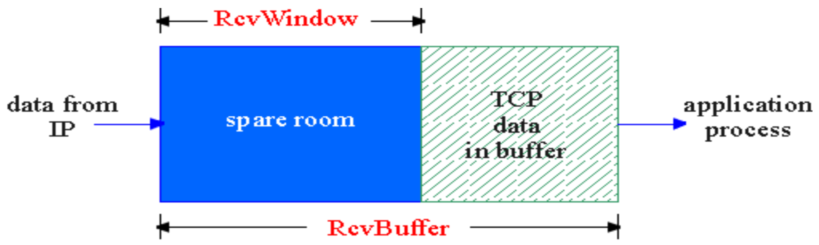
Flow Control
sender won't overflow receiver's buffer by transmitting too much, too fast



- speed-matching service: matching the send rate to the receiving app's drain rate

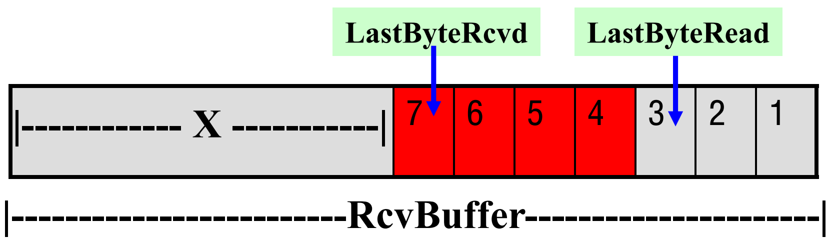
- application process may be slow at reading from buffer

TCP Flow control: how it works

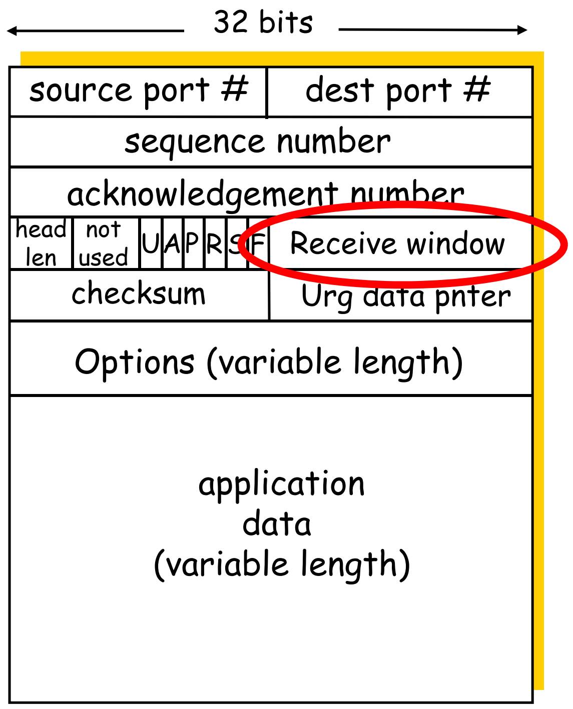


(Suppose TCP receiver discards out-of-order segments)

Spare Room in Buffer = RcvWindow
= RcvBuffer-[LastByteRcvd - LastByteRead]


LastByteRcvd
LastByteRead
|----------- X -----------|
7
6
5
4
3
2
1
|------------------------RcvBuffer---------------------|

TCP Flow control: how it works

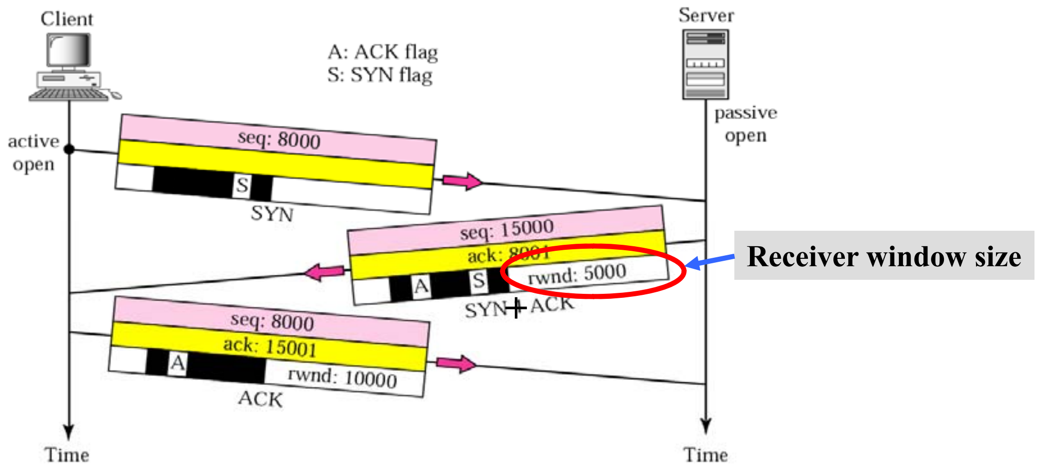


- Receiver advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - guarantees receive buffer doesn't overflow

TCP Connection

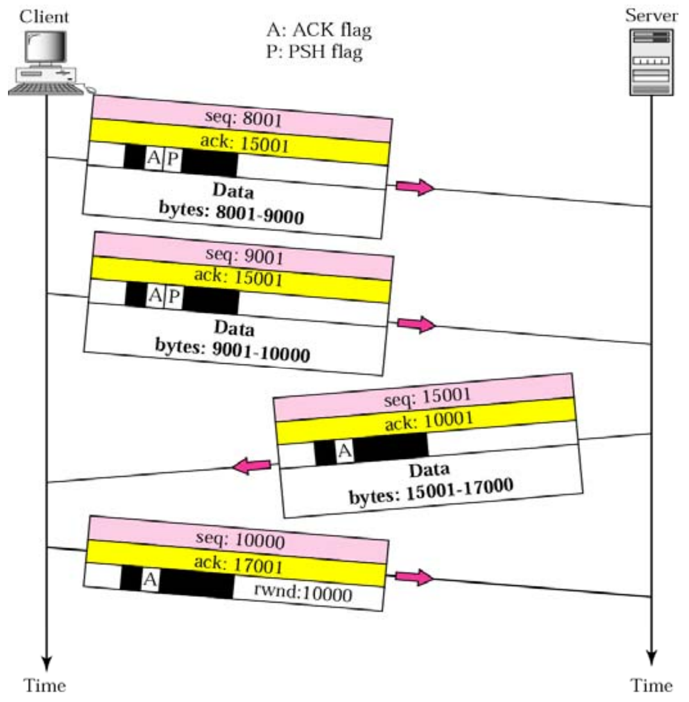
- TCP is connection-oriented.
- Connection-oriented transport protocol establishes virtual path between source and destination.
- All of segments belonging to message are then sent over this virtual path.
- Connection-oriented transmission requires three phases:
  - Connection Establishment,
  - Data transfer, and
  - Connection Termination.

Connection Establishment using Three-Way Handshaking

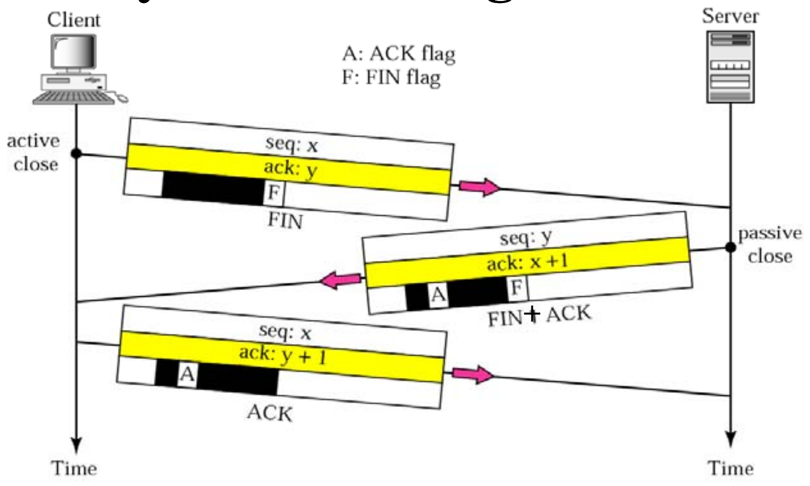


- ***SYN segment cannot carry data*, but it consumes one sequence number.**
- ***SYN + ACK segment cannot carry data*, but does consume one sequence number**
- ***ACK segment*, *if carrying no data*, consumes *no sequence number*.**

Data Transfer

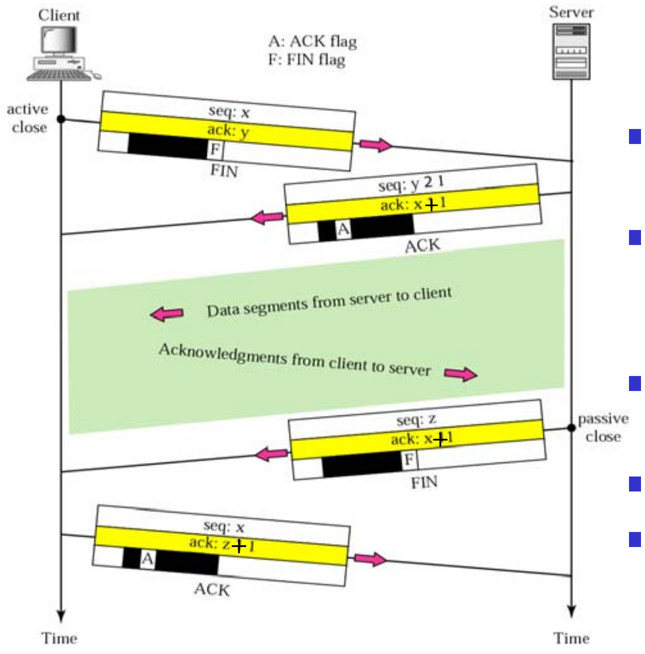


Connection termination using three-way handshaking



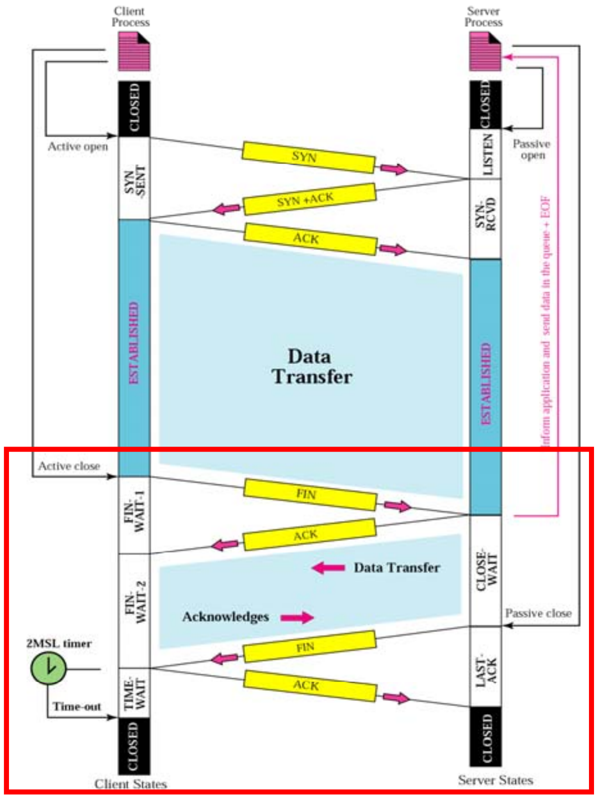
- **FIN segment consumes one sequence number if it does not carry data.**
- **FIN + ACK segment consumes one sequence number if it does not carry data**
- **ACK segment, if carrying no data, consumes no sequence number**

Half-Close

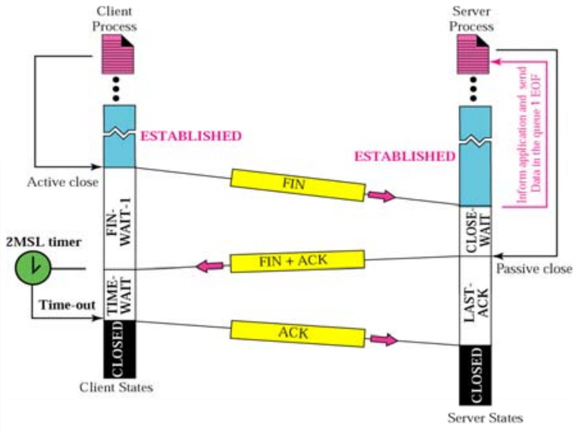


- In TCP, one end can stop sending data while still receiving data called **<u>Half-close</u>**
- Client half-close connection by sending FIN segment
- Server accepts this half-close by sending ACK segment
- Data transfer from client to server stops
- Server can still send data
- After sending all data, server sends FIN segment, which is acknowledged by ACK from client

Connection Establishment and Termination



- The common value of MSL (Maximum Segment Lifetime is between 30 seconds and 1 minutes



- **Three-way Handshake**

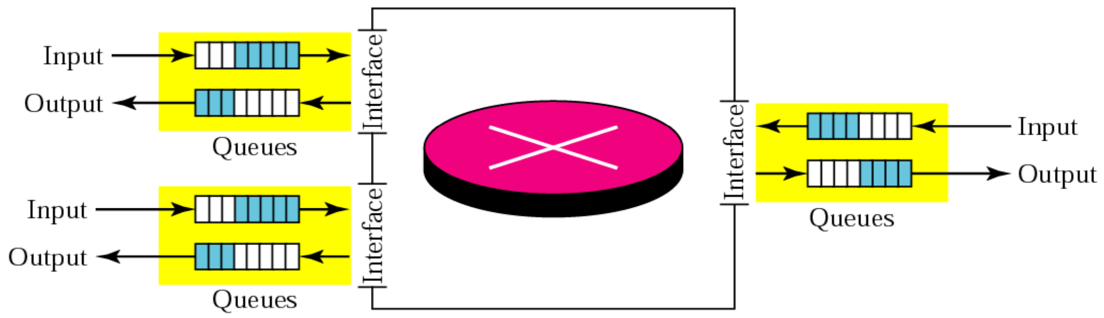
Principles of Congestion Control

Congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

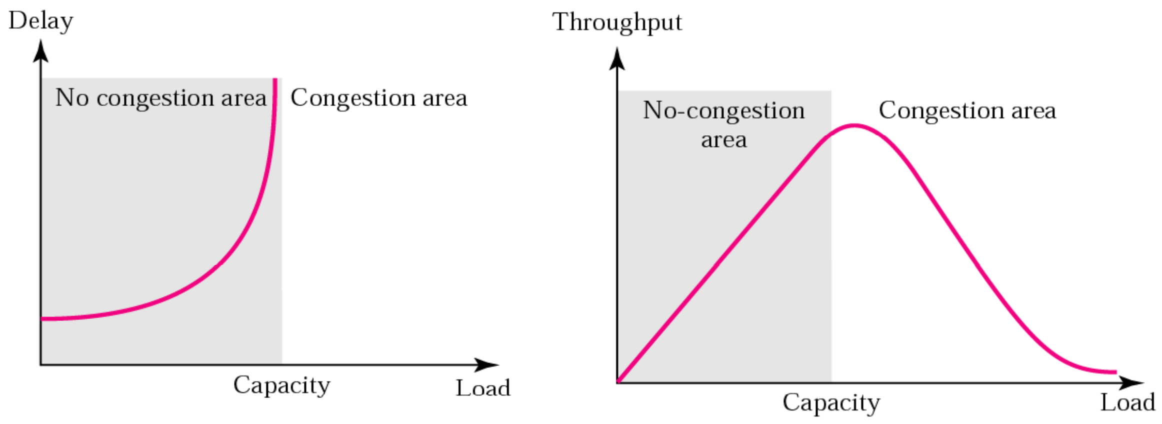
- ***Congestion control* refers to *mechanisms* and *techniques* to keep *load below capacity*.**

Router Queues



- Packet is placed at end of input queue while waiting to be checked
- Processing module of router removes packet from input queue one it reaches the front of queue and uses its routing table and destination address to find route
- Packet is put in appropriate output queue and waits its turn to be sent

Network Performance



- Congestion control involves two factors that measure performance of network
  - Delay
  - Throughput – number of packets passing through network in a unit of time

Approaches towards congestion control

Two broad approaches towards congestion control:

1. Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN (Explicit Congestion Notification), ATM)

2. End-end congestion control:

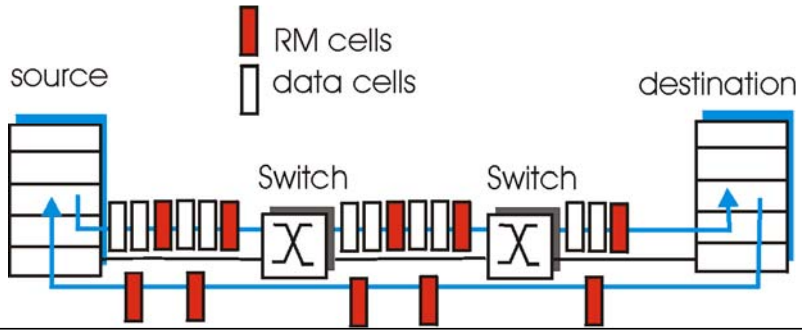
- no explicit feedback from network
- congestion inferred from end-system observed **loss**, **delay**
- approach taken by TCP

Case study: ATM ABR congestion control

ABR: Available Bit Rate:

- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

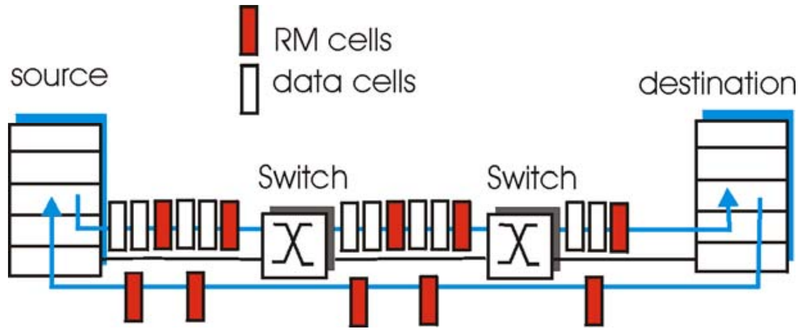
Case study: ATM ABR congestion control



RM (Resource Management) cells:

- Sent by sender, interspersed with data cells (one RM cell every 32 data cell)
- bits in RM cell set by switches (“*network-assisted”*)
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- two-byte ER (Explicit Rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender’ send rate thus maximum supportable rate on path
- RM cells returned to sender by receiver, with bits intact

Case study: ATM ABR congestion control

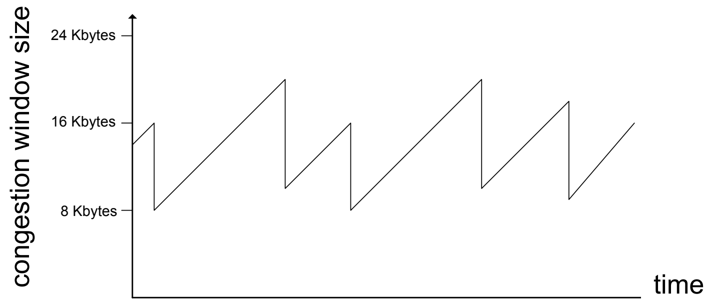


- EFCI bit in data cells:
  - Each data cell contains an Explicit Forward Congestion Indication (EFCI) bit
  - EFCI bit may be set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

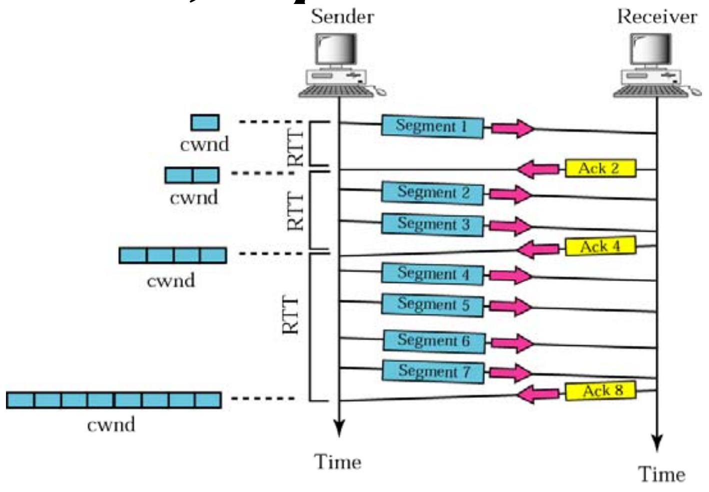
TCP Congestion Control: Additive Increase (AI), Multiplicative Decrease (MD)

- *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **CongWin** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **CongWin** in half after loss

Saw tooth behavior: probing for bandwidth

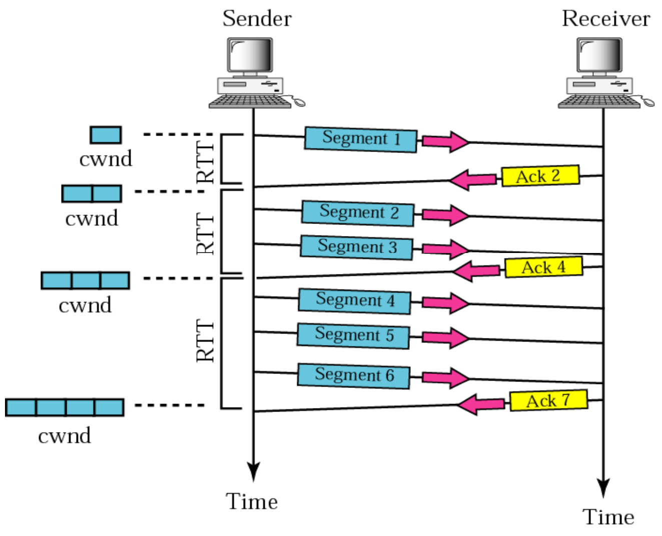


Slow Start*, *exponential increase



- When connection begins, increase rate exponentially until first loss event:
  - double **Congestion Windows** (cwnd) every RTT
  - done by incrementing **cwnd** for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast

Congestion Avoidance, additive increase

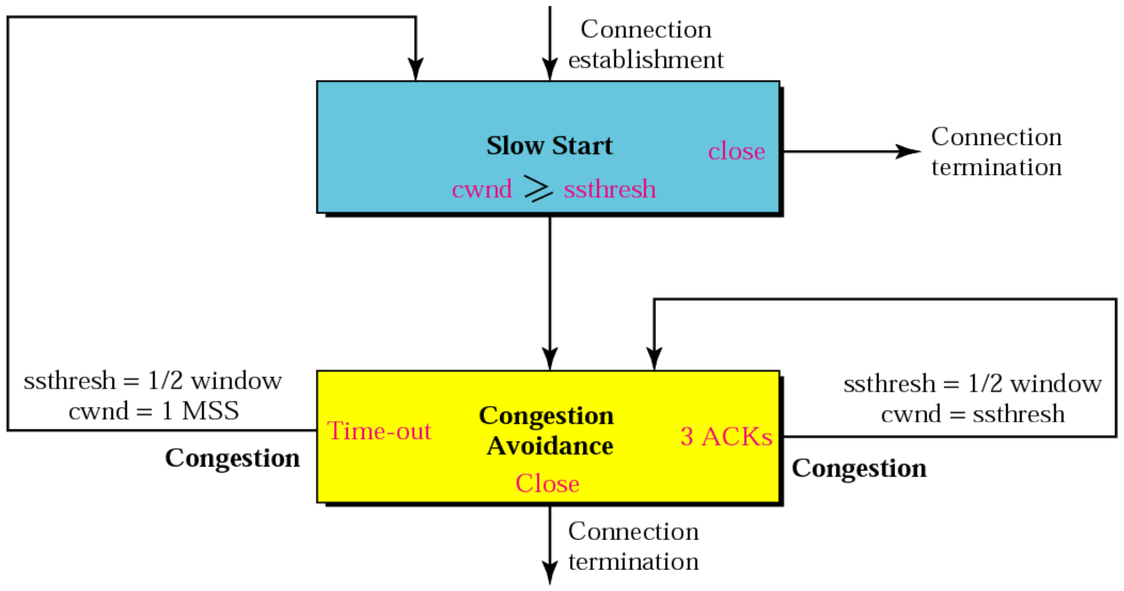


- ***In congestion avoidance algorithm size of congestion window (cwnd) increases additively until congestion is detected***

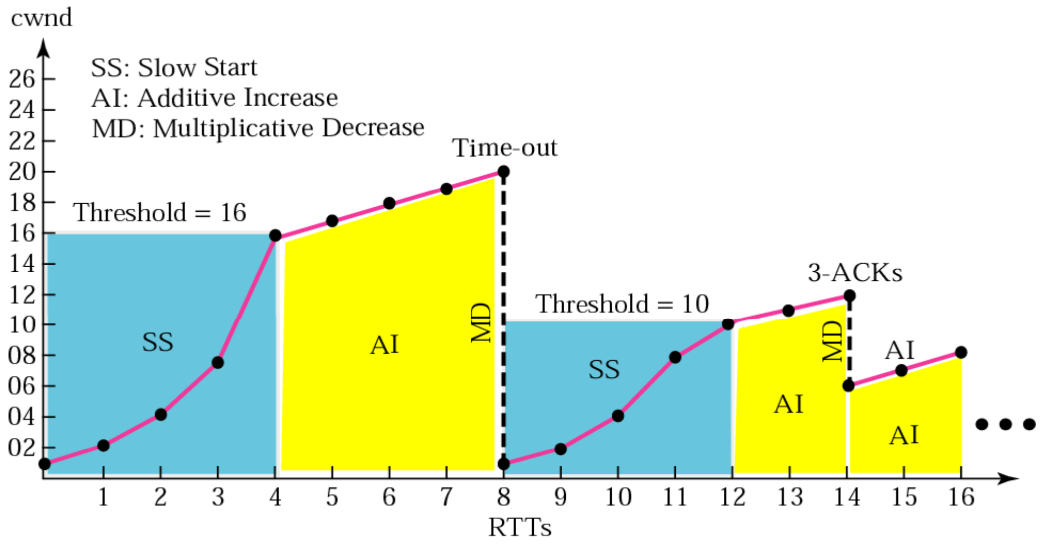
Most implementations react differently to congestion detection:

- ***If detection is by time-out, a new slow start phase starts***
- ***If detection is by three ACKs, a new congestion avoidance phase starts***
- ***Mechanism***
  - ***AIMD (Additive Increase, Multiplicative Decrease)***
  - ***Slow Start (SS)***
  - ***Congestion Avoidance***

TCP congestion policy summary



Congestion example



Summary: TCP Congestion Control

- When **CongestionWindow(cwnd)** is below **Threshold**, sender in slow-start phase, window grows exponentially
- When **cwnd** is above **Threshold**, sender is in congestion-avoidance phase, window grows linearly
- When triple duplicate ACK occurs, **Threshold** set to **cwnd/2** and **cwnd** set to **Threshold**
- When timeout occurs, **Threshold** set to **cwnd/2** and **cwnd** is set to 1 MSS