# Chapter 3 Transport Layer

Wednesday, March 24, 2021 2:10 PM

**3.1 Tranport Layer Services**
- provides logical communication processes running on diff hosts

**3.1.1 Relationship Transport and Network Layers**
- transport layer provides logical communication bw processes running on different hosts
- network layer protocol provides logical communication bw hosts

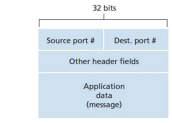
**3.1.2 Overview Transport layer in internet**
- UDP User Datagram Protocol
  - unreliable service
- TCP Transmission Control Protocol
  - reliable data transfer
  - flow control
  - sequence numbers
  - acknowledgements
  - timers
  - congestion control
- transport layer packet = segment
- IP service model is best-effort delivery service
- **Tranport layer multiplexing and demultiplexing**
  - extending host to host delivery to process to process delivery

**3.2 Multiplexing and Demultiplexing**

extending host to host delivery service provided by network layer to process to process delivery service for apps running on host
needed for all computer network

**demultiplexing**
- delivering data in transport layer segment to correct socket
  - ex. Bill receives mail from box, demultiplexing operation by observing whom letters addressed, and hand delivering to siblings
- each socket in host assigned port number
- when segment arrives at host, transport layer examines destination port number in segment and directs segment to correstponding socket

**multiplexing**
- gathering data chunks at source host from different sockets
- encapsulating each data chunk with header information to create segments
- passing segments to network layer
  - ex. Ann multiplexing operation when collecting letters from siblings and putting in mail box
- requires
  - that sockets have unique identifies
  - that each segments have special fields that indicate the socket to which segment is to be delivered
    - source port number
    - destination port number



Source and destination port-number fields in a transport-layer segment

**Connectionless Mutliplexing and Demultiplexing (UDP)**
- can associate specific port number to UDP socket bia socket .bind()
- server uses UDP recvfrom() to extract client side port number from segment received from client, sends new segment to client with extracted source port number
- UDP socket fully identified by **two tuple** consisting of dest IP and dest port number
  - if 2 UDP segments have diff source IP addresses/source port but same dest IP and dest port, 2 segments will be directed to same dest process

**Connection Oriented Multiplexing and Demultiplexing (TCP)**
- TCP socket vs UDP socket: TCP socket identified by **four-tuple**
  - source IP address
  - source port number
  - destination IP address
  - destination port number
- server may support many simultaneous TCP connection sockets with each socket attached to process, and each socket identified by its own four-tuple
- when a TCP segment arrives at host, all four fields are used to demultiplex segment to socket

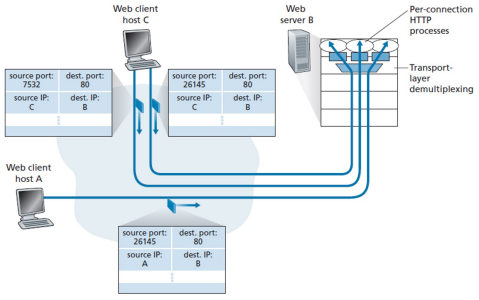


**Figure 3.5** • Two clients, using the same destination port number (80) to communicate with the same Web server application

- host C initiates 2 HTTP sessions to server B
- Host A initiates 1 HTTP session to B
- server B will be able to demultiplex 2 connections with same source port number since two connections have diff source IP addresses

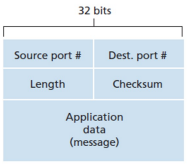
- **Web Servers and TCP**
  - web server spawns new process for each connection with own connection socket
  - not always one to one correspondence between connection sockets and processes
  - cand use persistent HTTP to exchange HTTP msgs via same server socket
  - if use non persistent HTTP then new TCP connection created and closed each request/response - busy web server

**3.3 Conncetionless Transport: UDP**
- adds nothing to IP aside from multiplexing/demultiplexing function and light error checking
- takes msg from app process, attaches src and dest port num fields for mux/demux service, adds two other small fields, passes segment to network layer
- no handshaking, connectionless
  - ex. DNS
- some applications are better suited for UDP
  - finer application level control over what data is sent and when
    - UDP packages data inside UDP segment and immediately passes to network layer
    - real time applications require min sending rate, do not want to delay segment transmission and can tolerate some data laoss
  - no connection establishment
    - DNS would be much slower if ran oer TCP
  - No connection state
    - UDP does not maintain connection state and does not track any of these parameters
    - can support many more active clients
  - small packet header overhead
    - 8 bytes of overhead vs TCP 20 bytes

| Application | Application-Layer Protocol | Underlying Transport Protocol |
|---|---|---|
| Electronic mail | SMTP | TCP |
| Remote terminal access | Telnet | TCP |
| Secure remote terminal access | SSH | TCP |
| Web | HTTP, HTTP/3 | TCP (for HTTP), UDP (for HTTP/3) |
| File transfer | FTP | TCP |
| Remote file server | NFS | Typically UDP |
| Streaming multimedia | DASH | TCP |
| Internet telephony | typically proprietary | UDP or TCP |
| Network management | SNMP | Typically UDP |
| Name translation | DNS | Typically UDP |

**Figure 3.6** ♦ Popular Internet applications and their underlying transport protocols

**3.3.1 UDP Segment structure**
- DNS datafield has query or response msg
- streaming audio app, audio samples fill datafield
- UDP header has 4 fields 2 bytes each
- length field specifies num bytes in UDP segment (header+data)
- checksum used by receiving host to check if errors in segment




**3.2.2 UDP Checksum**
- determine if bits in UDP seg altered from src to dest
- UDP sender side performs 1s complement of sum of all 16 bit words in segment with overflow from wrap around
- result put in checksum field of UDP seg
- ex. 3 16-bit words:

```
0110011001100000
0101010101010101
1000111100001100

The sum of first two of these 16-bit words is

0110011001100000
0101010101010101
1011101110110101

Adding the third word to the above sum gives

1011101110110101
1000111100001100
0100101011000010
```

  - 1s complement: reverse all 0 to 1 and 1 to 0
  - therefore checksum = 1011010100111101
- At receiver, all 4 16 bit words are added including checksum
  - if no errors, sum at receiver will be 1111111111111111
  - if one bit is 0, then error introduced in packet
- **end-end principle**
  - certain functionality (error detection) must be implemented on end-end basis
  - functions placed at lower level may be redundant or of little value when compared to cost of providing them at higher level
- UDP does not recover from an error, discard segment, or warning

**3.4 Principles of Reliable Data Transfer**
- reliable data transfer protocol
  - no transferred data bits are corrupted or lost, delivered in order

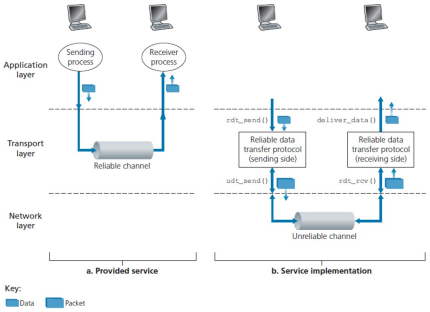
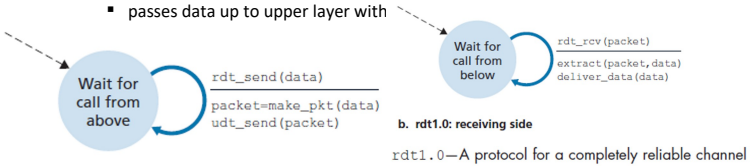



**Figure 3.8** ♦ Reliable data transfer: Service model and service implementation

- sending side of dtp invoked from above by call to **rdt_send()**
- pass data to be delivered toupper layer at receiving side
- receiving side **rdt_rcv()** called when packet arrives from receiving side of channel
- **deliver_data()** called when rdt protocol wants to deliver data to upper layer
- **unidirectional data transfer**
    - data transfer from sending to receiving side
- sending and receiving side of protocol will still need to transmit packets in both directions
- rdt needs to exchange control packets back and forth by calling **udt_send()**

### 3.4.1 Buidling Reliable Data Transfer Protocol

- **rdt 1.0: RDT over perfectly reliable channel**
    - channel completely reliable, no errors
    - FSM
        - separate FSMs for sender and receiver
        - sender and receiver FSMs have just one state
        - arrows = transition of protocol from 1 state to next
        - event causing transition shown above horizontal line
        - actions take when event occurs shown below horiontal line
        - no action taken = symbol ^ below/above horizontal
    - **Sender side**
        - sending side accepts data from upper layer with **rdt_send(data)** (from upper layer app)
        - creates packet containing data with **make_pkt(data)**
        - sends packet into channel
    - **Receiving side**
        - rdt receives packet from channel with **rdt_rcv(packet)** event (from lower layer protocol)
        - removes data from packet with **extract(packet, data)**
        - passes data up to upper layer with

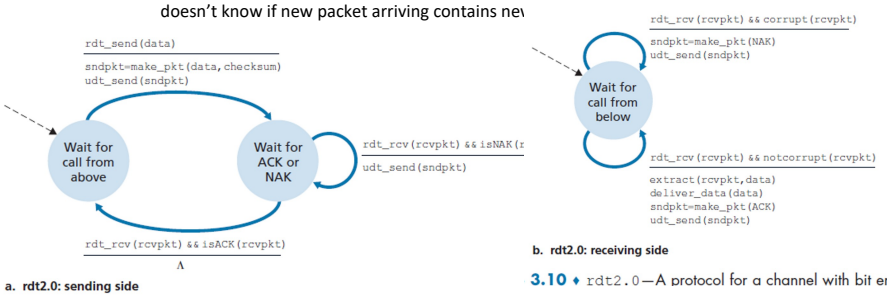



rdt1.0—A protocol for a completely reliable channel

**a. rdt1.0: sending side**

- **rdt 2.0: RDT over Channel with Bit Errors**
    - channel bits in packet may be corrupted
        - occur in physical components of network when packet transmitted/propagated/buffered
    - **Acknowledgement**
        - positive acknowledgement **ACK** (OK)
        - negative acknowledgments **NAK** (Repeat that)
        - lets receiver tell sender what has been received and what has errors/needs retransmit
            - retransmission protocols - ARQ Automatic Repeat reQuest
    - **ARQ Automatic Repeat reQuest**
        - **Error Detection**
            - receive detect when bit error occurred
            - extra bits gathered into packet checksum field of rdt 2.0 data packet
        - **Receiver feedback**
            - receiver provides explicit feedback to sender
            - pos ACK 1 and neg NAK 0
        - **Retransmission**
            - packet received in error is retransmitted by sender
    - **Sender side**
        - send side protocol waits for data passed down from upper layer
        - when **rdt_send(data)** event, sender creates packet **sndpkt** containing data along with packet checksum
        - packet is sent with **udt_send(sndpkt)**
        - waiting for ACK or NAK packet from receiver
        - if ACK (**rdt_rcv(packet) && isACK(rcvpkt)**), packet received correctly, return to wait for data
        - if NAK, protocol retransmits last packet and waits for ACK/NAK
        - when in wait for ACK/NAK state, sender is cannot get more data from upper layer, **rdt_send()** cannot occur, will not send new piece of data
            - **stop-and-wait** protocol
    - **Receiver side**
        - on packet arrival, receiver replies with ACK or NAK whether packet corrupted (**rdt_rcv(rcvpkt) && corrupt(rcvpkt)**
    - **Con:**
        - not accounted for ACK or NAK packet corrupted, sender has no way of knowing whether or not receiver has received last piece of transmitted data

- **Solutions**
  - new type of sender-to-receiver packet to protocol if not understood (OK, Repeat, What did you say?), but the What did you say? may be corrupted
  - add enough checksum bits to allow sender to detect and recover from bit errors
  - sender can resend current packet when garbled ACK/NAK packet received, but can duplicate packets. receiver doesn't know wheter ACK/NAK last sent received correctly by sender, so doesn't know if new packet arriving contains ne

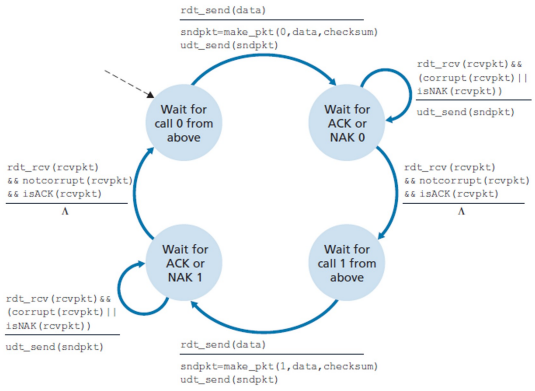



a. rdt2.0: sending side

b. rdt2.0: receiving side

3.10 ♦ rdt2.0—A protocol for a channel with bit errors

- **rdt 2.1: fixed version of 2.0 with sequence number**
  - add new field to data packet and have sender number its data packets with sequence number in the field
  - receiver checks seq num to determine if packet is retransmission
  - for this stop-an-wait protocol, 1-bit seq num will be enough
    - retransmission: seq num of received packet has same seq num as most recently received packet
    - new packet: seq num changes, moving fwd in %2 arithmetic
  - since channel does not lose packets, ACK NAK do not indicate seqnum of packets
  - **FSM**
    - state must reflect whether packet being sent by sender or expected packet at receiver should have seq num 0 or 1
    - When packet out of order received, receiver sends ACK
    - When corrupted packet received, receiver sends NAK




3.11 ♦ rdt2.1 sender

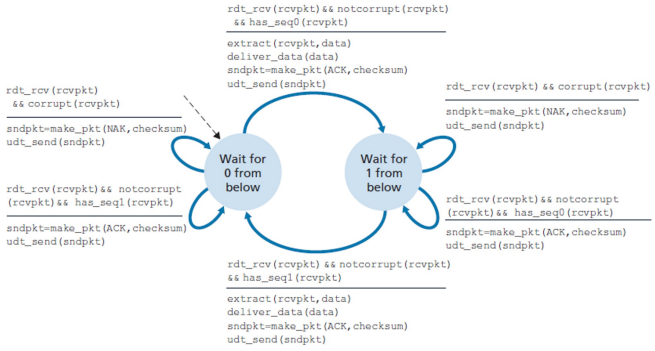
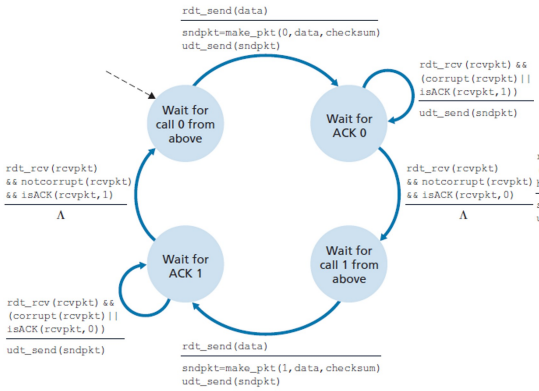



Figure 3.12 ♦ rdt2.1 receiver

- **rdt 2.2: receiver must include seq num of packet being acknowledged by ACK msg**
  - instead of NAK, can also send an ACK for the last correctly received packet, **duplicate ACK**
  - receiver did not correctly receive the corrupted packet since ACK twice
  - No need for NAK, use ACK 0 or ACK 1 in **make_pkt()**
  - sender checks seq num of packet being ACK by including 0 or 1 in **isACK()** in sender




3.13 ♦ rdt2.2 sender

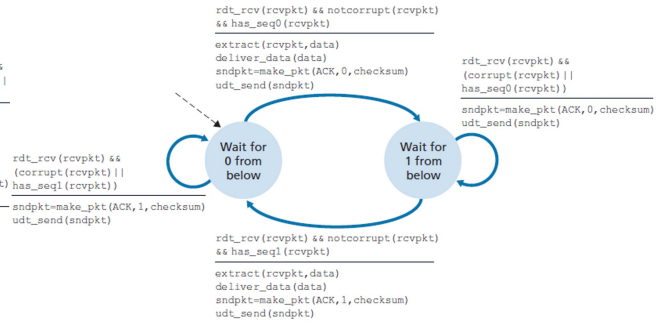
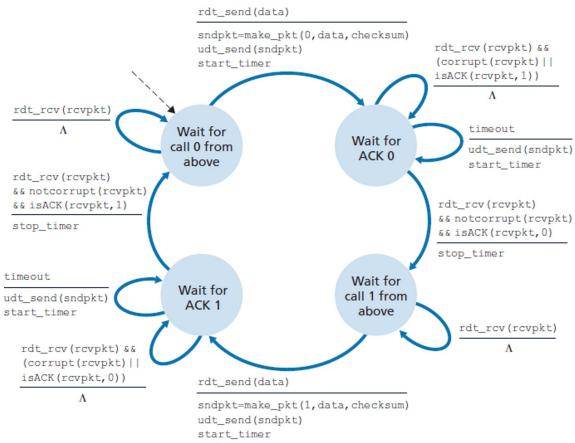


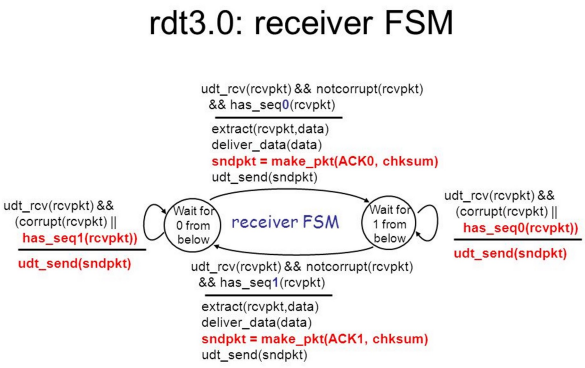

Figure 3.14 ♦ rdt2.2 receiver

- **rdt 3.0: rdt over lossy channel with bit errors (alternating bit protocol)**
  - channel can lose packets
    - need to detect packet loss and do something when packet loss occurs
    - use same techniques as **rdt 2.2** to detect packet loss
  - **Sender side**
    - sender detects and recovers from packet loss
    - sender transmits data packet
    - if sender transmits packet and data packet or receiver ACK gts lost, no reply
    - if sender waits long enough so that it is certain that packet is lost, can retransmit the data packet
      - needs to choose a time such that packet loss is likely to have happened

- **Problem:**
  - packet may just be delayed, sender may retransmit packet even though data packet and ACK not lost, **duplicate data packets**
  - solved with rdt 2.2 with seq num duplicate packets
- Actions for whether apcket is lost, ACK lost, or packet/ACK delayed is to retransmit
- time based retramsission **countdown timer**
  - interrupt sender after amount of time
  - sender starts timer each time a packet is sent
  - responds to timer interrupt (taking appropriate actions)
  - stops timer

- **alternating bit protocol**
  - seq num alternate bw 0 and 1




### 3.4.2 Pipelined Reliable Data Transfer Protocols

- rdt 3.0 stop-and-wait protocol not optimal performance

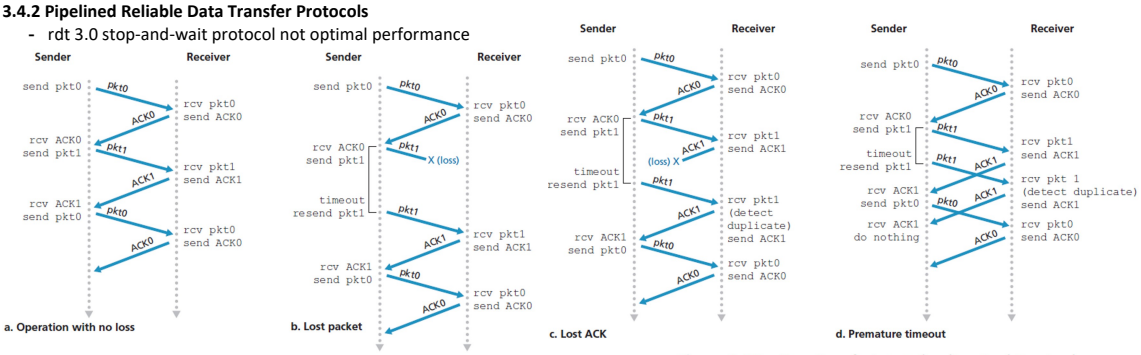



**Figure 3.16** ♦ Operation of `rdt3.0`, the alternating-bit protocol

- network protocols can limit the capabilities provided by network
- **Utilization of sender**
  - fraction of time sender is actually busy sending bits into channel - lots of wasted time

$$d_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microseconds}$$

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

- **Solution - Pipelining**
  - sender is allowed to send multiple packets without waiting for ACK
    - if sender allowed transmit 3 packets before having to wait for

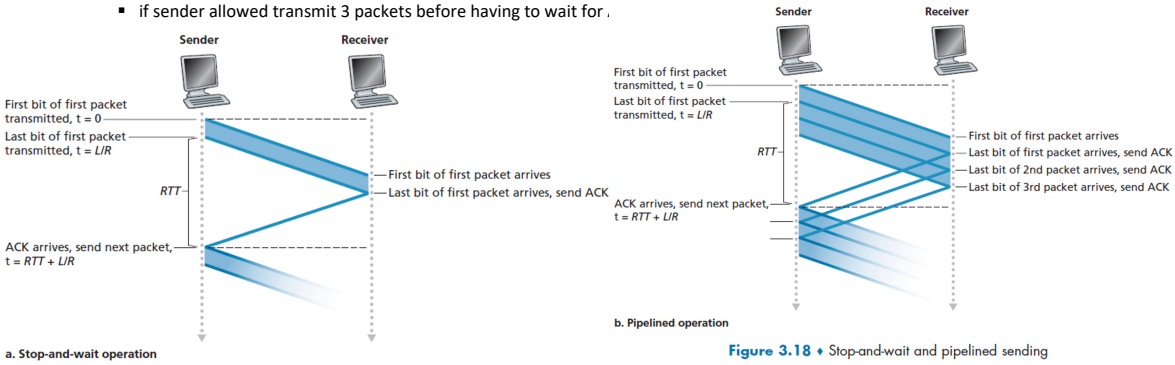



**Figure 3.18** ♦ Stop-and-wait and pipelined sending

  - **Consequences**
    - range of seq num increased since each in-transit packet must have unique seq num, may be multiple unACKed packets
    - sender and receiver sides may have to buffer packets transmitted but not ACKed. Buffering of correctly received packets may also be needed
    - Range of seq num needed and buffering req will depend on manner in which data transfer protocol responds to lost corrupted and delayed packets
  - **Solutions - Pipelining Error Recovery**
    - Go-Back-N
    - Selective Repeat

### 3.4.3 Go-Back-N (GBN)

- sender is allowed to transmit multiple packets wihtout waiting for ACK
- but constrained to have no more than max number N of unACKed packets in pipeline

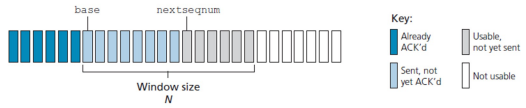
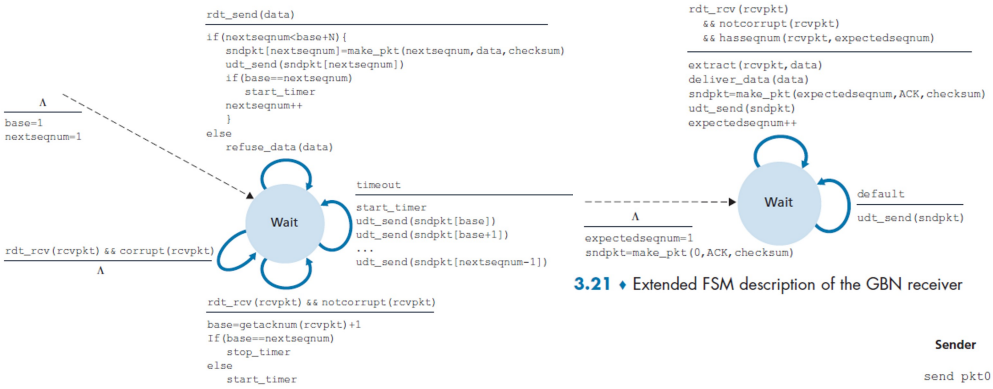



**Figure 3.19** ♦ Sender's view of sequence numbers in Go-Back-N

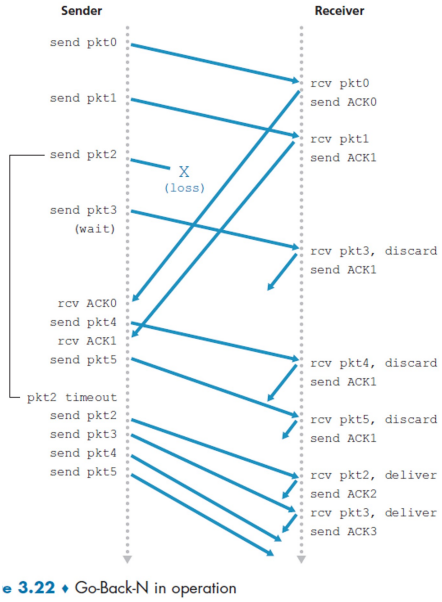
  - **base** is seq num of oldest unACKed packet
  - **nextseqnum** is smallest unused seq num, next packet to be sent
  - seq num in interval **[0, base-1]** correspond to packets already transmitted and ACKed
  - interval **[base, nextseqnum-1]** corresponds to packets have been sent but not ACKed
  - seq num in interval **[nextseqnum, base+N-1]** can be used for packets that can be sent immediately if data arrives from upper layer
  - seq num >= **base+N** cannot be used until unACKed packet in pipeline with seqnum **base** is ACKed
- **N** is **window size,** GBN is **sliding-window protocol**




**3.21** ♦ Extended FSM description of the GBN receiver

**re 3.20** ♦ Extended FSM description of the GBN sender

- **extended FSM**
  - have added variables for base and nextseqnum
  - added operations on these variables and conditional actions involving these variables
- **Sender Side GBN Events**
  - **Invocation from above**
    - when **rdt_send()** called from above, sender checks to see if window full and if unACKed packets
    - if not full, packet created and sent, vars are updated
    - if full, sender returns data back to upper layer or buffer, indicates window full
  - **Receipt of an ACK**
    - ACK for packet with **seqnum n** taken to be **cumulative acknowledgement**
    - all packets with seqnum up to including **n** correctly received at receive
  - **Timeout event**
    - when lost or delayed packets, timer is used to recover lost data or ACK packets
    - if timeout, sender resends all packets previously sent but not yet ACKed
    - if ACK received but still additional transmitted but unACKed packets, timer is restarted
    - if no outstanding unACKed packets, timer stopped
- **Receiver Side**
  - if packet with seq num **n** received and in order, receiver sends ACK for packet n and delivers data to upper layer
  - else receiver discatds packet and resends ACK for most recently received in-order packet
  - cumulative acknowledgement
    - if packet k received and delivered, then all packets seq num < k also delivered
  - discards out-of-order packets even if correctly received
    - receiver does not need to buffer any out of order packets
  - receiver only maintains seq num of next in order packet while sender maintains upper and lower bounds of window and position of **nextseqnum**
- **event-based programming**
  - actions taken in response to events that can occur
  - procedures called by other procs or stack or interrupt
  - Sender
    - call from upper layer entity **rdt_send()**
    - timer interrupt
    - call from lower layer to invoke **rdt_rcv()** when packet arrives
- techniques: seq num, cumulative acknowledgements, checksums, timeout/retransmit operation

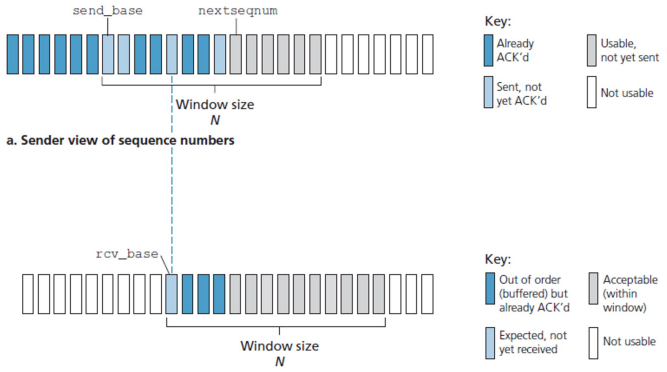



**e 3.22** ♦ Go-Back-N in operation

### 3.4.4 Selective Repeat (SR)

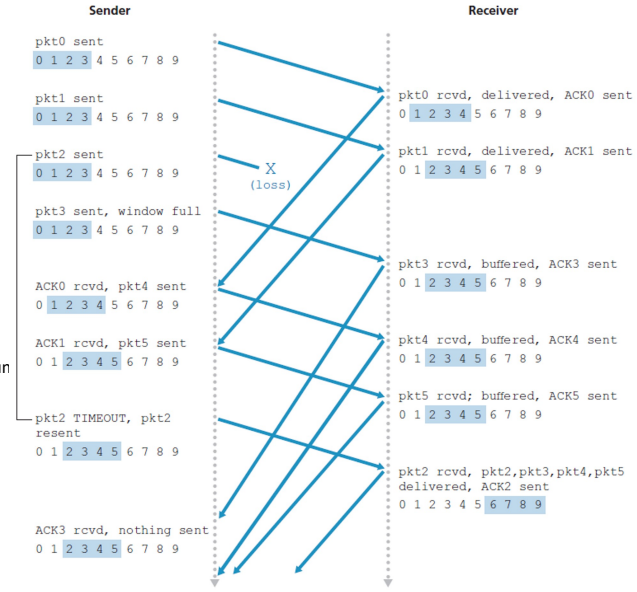
- **Problem GBN**
  - GBN can have performance problems, window size and bandwidth delay products are large, many packets in pipeline
  - single packet error can cause GBN to retransmit large number of packets
- **Solution SR**
  - avoids unecesary retransmissions by having sender retransmit only packets that were received in error (lost or corrupted) at receiver
  - receiver individually acknowledges correctly received packets
  - window size N limits nubmer of outstanding unACKed packets in pipeline
  - sender will have already received ACKs for some of the packets in window
- **Receiver**
  - ACK correcly received packet whether or not in order

- Out of order packets are buffered until any missing packets (lower seq num) received, then batch of packets can be delivered to upper layer

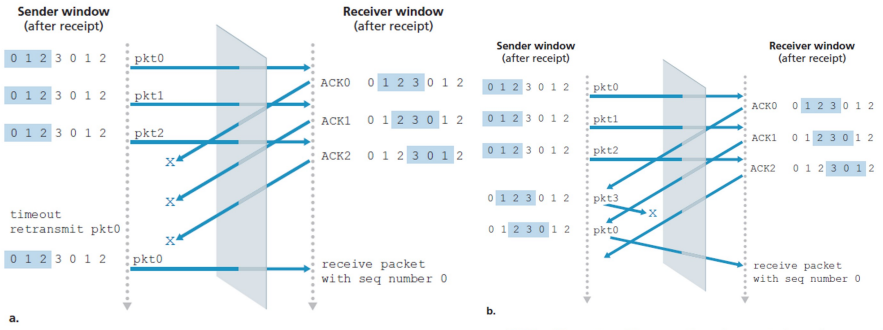



ure 3.23 ♦ Selective-repeat (SR) sender and receiver views of sequence-number space

- **SR Sender Events and Actions**
  - **Data received from above**
    - sender checks next available seq num for packet
    - if seq num within window, data sent, else buffered or returned
  - **Timeout**
    - protect against lost packets
    - each packet has own logical timer since only single packet transmitted on timeout
  - **ACK received**
    - if ACK received, sender marks packet as received if in window
    - if packet seq num = **send_base**, window base moved fwd to unACKed packet with smallest seq nun
    - untransmitted packets with seq num within window are now transmitted
- **SR Receiver Events and Actions**
  - **Packet with seq num in [rcv_base, rcv_base+N-1] correctly received**
    - packet falls within receiver window, selective ACK packet returned to sender
    - if packet not prevously received, is buffered
    - if this packet has seq num = base of receive window **rcv_base**, packet and previously buffered conecutive packets are delivered to upper layer
    - receive window moved forward by num packets delivered
  - **Packet with seq num in [rcv_base-N, rcv_base-1] is correctly received**
    - ACK is generated even if packet previously acknowledged
  - **Otherwise**
    - ignore packet
- **Lack of synchronization bw sender and receiver**
  - finite range of seq num 4 packet seq nums and window size 3
  - packets 0-2 transmitted correctly received and ACKed at receiver
  - receiver window moves over 4,5,6 packets with seq num 3,0,1
  - receiver receives packet with seq num 0
  - no way of distinguishing retransmission of first packet from original transmission of fight packet
  - window size 1 less than size of seq num space won't work
  - **Window size** needs to be **<=** half size of seq num space




ure 3.26 ♦ SR operation




ure 3.27 ♦ SR receiver dilemma with too-large windows: A new packet or a retransmission?

| Mechanism | Use, Comments |
|---|---|
| Checksum | Used to detect bit errors in a transmitted packet. |
| Timer | Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within the channel. Because timeouts can occur when a packet is delayed but not lost (premature timeout), or when a packet has been received by the receiver but the receiver-to-sender ACK has been lost, duplicate copies of a packet may be received by a receiver. |
| Sequence number | Used for sequential numbering of packets of data flowing from sender to receiver. Gaps in the sequence numbers of received packets allow the receiver to detect a lost packet. Packets with duplicate sequence numbers allow the receiver to detect duplicate copies of a packet. |
| Acknowledgment | Used by the receiver to tell the sender that a packet or set of packets has been received correctly. Acknowledgments will typically carry the sequence number of the packet or packets being acknowledged. Acknowledgments may be individual or cumulative, depending on the protocol. |
| Negative acknowledgment | Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgments will typically carry the sequence number of the packet that was not received correctly. |
| Window, pipelining | The sender may be restricted to sending only packets with sequence numbers that fall within a given range. By allowing multiple packets to be transmitted but not |

| Mechanism | Use, Comments |
|---|---|
| Checksum | Used to detect bit errors in a transmitted packet. |
| Timer | Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within the channel. Because timeouts can occur when a packet is delayed but not lost (premature timeout), or when a packet has been received by the receiver but the receiver-to-sender ACK has been lost, duplicate copies of a packet may be received by a receiver. |
| Sequence number | Used for sequential numbering of packets of data flowing from sender to receiver. Gaps in the sequence numbers of received packets allow the receiver to detect a lost packet. Packets with duplicate sequence numbers allow the receiver to detect duplicate copies of a packet. |
| Acknowledgment | Used by the receiver to tell the sender that a packet or set of packets has been received correctly. Acknowledgments will typically carry the sequence number of the packet or packets being acknowledged. Acknowledgments may be individual or cumulative, depending on the protocol. |
| Negative acknowledgment | Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgments will typically carry the sequence number of the packet that was not received correctly. |
| Window, pipelining | The sender may be restricted to sending only packets with sequence numbers that fall within a given range. By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation. We'll see shortly that the window size may be set on the basis of the receiver's ability to receive and buffer messages, or the level of congestion in the network, or both. |

or a retransmission?

### 3.5 Connection-Oriented Transport: TCP

#### 3.5.1 TCP Connection

- connection-oriented, handshake
- **full duplex service**
  - if TCP connection bw process A on host and Process B on other host, app layer data can flow from A to B at same time as app layer data flows from B to A
- **point to point**
  - bw single sender and receiver
- max amount of data that can be grabbed and placed in segment limited by **maximum segment size (MSS)**
  - set by determining length of largest link-layer frame sent by local sending host **maximum transmission unit MTU**
  - then set MSS to ensure TCP segment datagram + header 40 bytes will fit

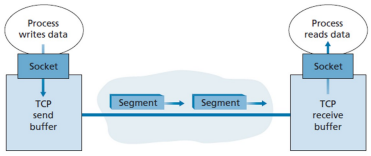
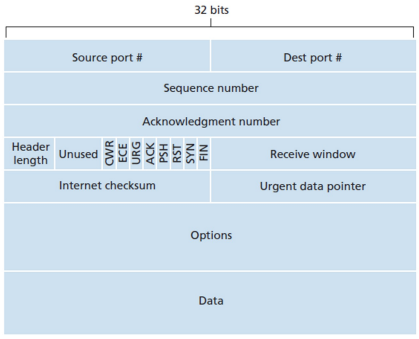
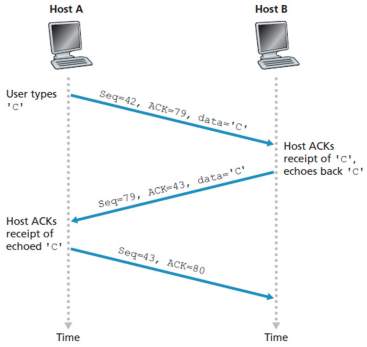



Figure 3.28 ♦ TCP send and receive buffers

#### 3.5.2 TCP Segment Structure




- header fields and data field
- MSS limits max size of segment data field
- **header**
  - source port
  - dest port
  - checksum
  - 32 bit seq num field
  - 32 bit ACK number field for rdt protocol
  - 16 bit receive window field for flow control
  - 4 bit header length field for length of TCP header in 32-bit words
  - optional and variable length option field for sender/receiver negotiating MSS or as window scaling factor for high speed networks
  - 6 bit flag field
    - ACK bit used to indicate value carried is valid
    - RST, SYN, FIN bits used for connection setup and teardown
    - CWR and ECE bits used in explicit congestion notif
    - PSH bit indicates receiver should pass data to upper layer immediately
    - URG bit indicates there is data in this segment that sending side upper layer marked urgent
- **Sequence Numbers and Acknowledgment Numbers**
  - Seq num for segment is the byte stream num of first byte in segment
  - ACK num that Host A puts in its segment is the seq num of next byte Host A is expecting from Host B
    - A receives bytes 0-535, and segment with bytes 900-1000, not received bytes 536-899
    - A's next seg to B will contain 536 in ACK number field
- **Telnet**




re 3.31 ♦ Sequence and acknowledgment numbers for a simple Telnet application over TCP

### 3.5.3 Round Trip Time Estimation and Timeout

- TCP uses timeout/retransmit mechanism to recover lost segments
- timeout needs to be larger than connection RTT, time segment sent until it is ACKed
- **Estimating RTT**
  - **SampleRTT:** time bw when segment sent and ACKed for when received
    - new value once every RTT
  - Upon obtaining new SampleRTT, RCP updates **EstimatedRTT**

$$\text{EstimatedRTT} = (1-\alpha)\cdot\text{EstimatedRTT} + \alpha\cdot\text{SampleRTT}$$

$$\text{EstimatedRTT} = 0.875\cdot\text{EstimatedRTT} + 0.125\cdot\text{SampleRTT}$$

  - weighted average
- measure variability of RTT **DevRTT** estimate how much **SampleRTT** deviates from **EstimatedRTT**

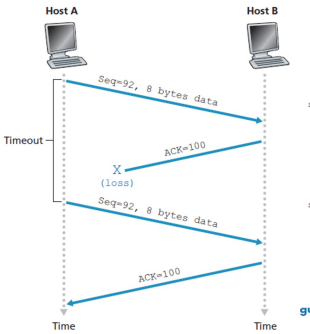
$$\text{DevRTT} = (1-\beta)\cdot\text{DevRTT} + \beta\cdot|\,\text{SampleRTT} - \text{EstimatedRTT}\,|$$

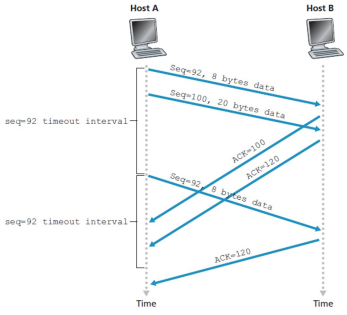
  - Beta = 0.25
- Determining retransmission timeout interval

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4\cdot\text{DevRTT}$$

### 3.5.4 Reliable Data Transfer

- TCP timer management proc use a

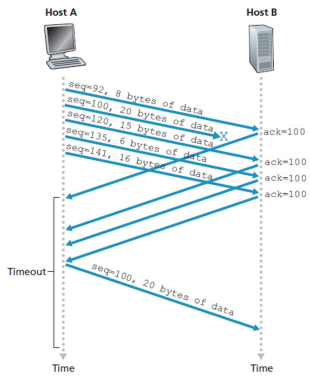



gure 3.35 ♦ Segment 100 not retransmitted

34 ♦ Retransmission due to a lost acknowledgment

- **Doubling Timeout interval**
  - each time TCP retransmits, sets next timeout interval to twice previous value
- **Fast Retransmit**
  - timeout triggered retransmission - timeout period can be long
  - sender can detect packet loss before timeout event by noting duplicate ACKs
  - when TCP receiver receives segment with seq num larger than next expected in order seq num, detects gap in data stream - missing segment
  - since multiple segments sent back to back, will be many back to back duplicate ACKs
    - indication that segment following segment that has been ACKed 3 times is lost
    - TCP sender performs fast retransmit of missing segment before segment's timer expires




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

      else {/* a duplicate ACK for already ACKed
              segment */
          increment number of duplicate ACKs
              received for y
          if (number of duplicate ACKS received
              for y--3)
              /* TCP fast retransmit */
              resend segment with sequence number y
      }
break;
```

3.37 ♦ Fast retransmit: retransmitting the missing segment before the segment's timer expires

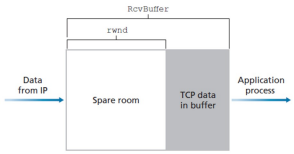
### 3.5.5 Flow Control

- eliminate possibility of sender overflowing receiver's buffer
- speed matching service - matches rate at which sender sending agains rate at which receiving app is reading
- sender maintains variable called **receive window**
  - how much free buffer space is available at receiver
  - **LastByteRead:** num of last byte in data stream read from buffer by B
  - **LastByteRcvd:** last byte in data stream arrived from network and
  - TCP cant overflow buffer

    `LastByteRcvd - LastByteRead ≤ RcvBuffer`
  - receive window **rwnd** set to amount of spare room in buffer

    `rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]`
    - rwd is dynamic, spare room changes with time
    - initially **rwnd = RcvBuffer**

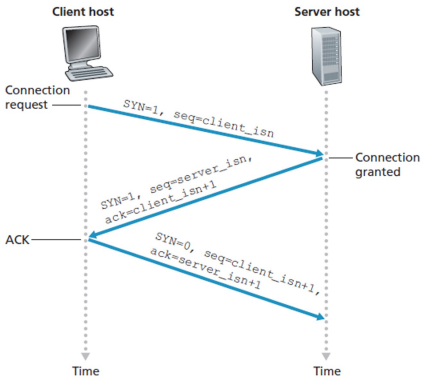
    `LastByteSent - LastByteAcked ≤ rwnd`
  - tcp receiver advertises free buffer space in rwnd field



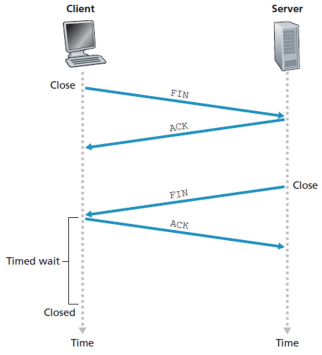

3.38 ♦ The receive window (rwnd) and the receive buffer (RcvBuffer)

### 3.5.6 TCP Connection Management

- **3 way handshake**
  - **Step 1**
    - client sends TCP segment to server
    - no application layer data, but flag SYN bit set 1 (SYN segment)
    - chooses random init seq num **client_isn** and puts in seq num field
  - **Step 2**
    - SYN segment arrives at server host
    - server extracts TCP SYN segment from datagram, allocates buffers and vars to connection
    - sends connection granted segment
    - no application layer data
    - SYN set to 1
    - ACK field set to **client_isn+1**
    - server chooses own init seq num **server_isn** puts value in seq num field of TCP seg header
    - connection granted segment is **SYNACK segment**
  - **Step 3**
    - after SYNACK seg received, client allocates buffers and vars to connection
    - client sends server another segment, acknowledges server connection
    - value **sever_isn+1** in acknowledgment field
    - SYN bit set to 0

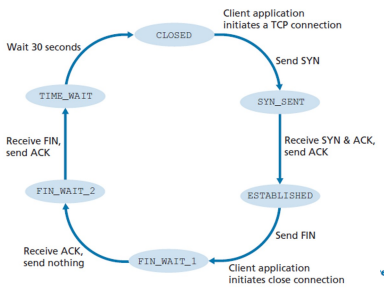



ure 3.39 ♦ TCP three-way handshake: segment exchange

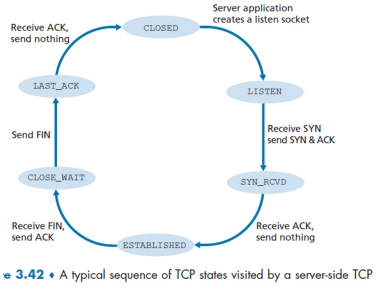



40 ♦ Closing a TCP connection

- close connection
  - client close command
  - send TCP segment to server process with **FIN** bit
  - server sends client acknowledgment segment
  - server sends own shutdown segment with FIN bit
  - client acknowledges server shutdown segment

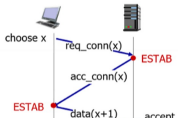



igure 3.41 ♦ A typical sequence of TCP states visited by a client TCP

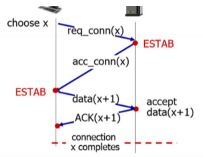



e 3.42 ♦ A typical sequence of TCP states visited by a server-side TCP

  - ESTABLISHED, client can send and receive segments with payload
  - FIN WAIT 1 client waits for segment from server with ACK
  - client goes to FIN WAIT 2 waits for server FIN segment
  - client ACKs server and enters TIME W
  - connection closed
- **2 way handshake**
  - variable delays
  - retransmitted msgs due to msg loss



    - connection closed
- **2 way handshake**
    - variable delays
    - retransmitted msgs due to msg loss
    - msg reordering
    - cant see other side




**3.6 Principles of Congestion Control**

- **Scenario 1: Two senders, router with infinite buffers**
- **Scenario 2: Two senders and router with finite buffers**
- **Scenario 3: 4 senders, routers w finite buffers, multihop paths**
- **End to end gongestion control**
- **Network assisted coongestion control**

**3.7 TCP Congestion Control**

- Lost segmetn implies congestion, TCP sender's rate should be decreased when segment lost
- ACKed segment indicates network is delivering senders segments to receiver, sender's rate can be increased when ACK arrives for previously unACKed segment
- Bandwidth probing
- **TCP congestion control algorithm**
    - **Slow Start**
    - **Congestion Avoidance**
    - **Fast Recovery**
    - **Retrospective**
    - **TCP Cubic**
    - **Reno Throughput**

**3.7.2 Network Assisted Explicit Congestion Notification and Delayed based Congestion Control**

- **Explicit Congestion Notification**
- **Delay based Congestion control**

**3.7.3 Fairness**

- **Fairness and UDP**
- **Fairness and Parallel TCP Connections**

**3.8 Evolution of Transport Layer Functionality**

- **QUIC: Quick UDP Internet Connections**