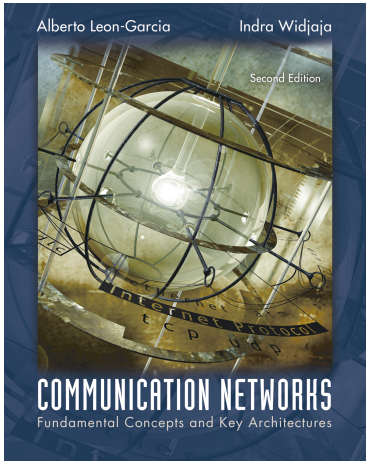
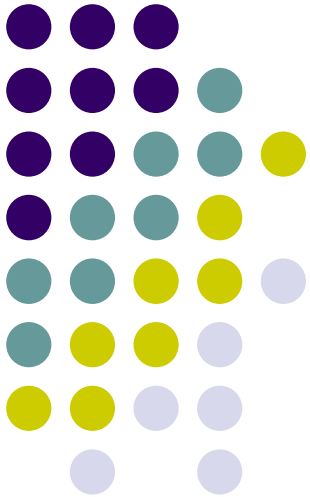
ARQ Protocols and Reliable Data Transfer

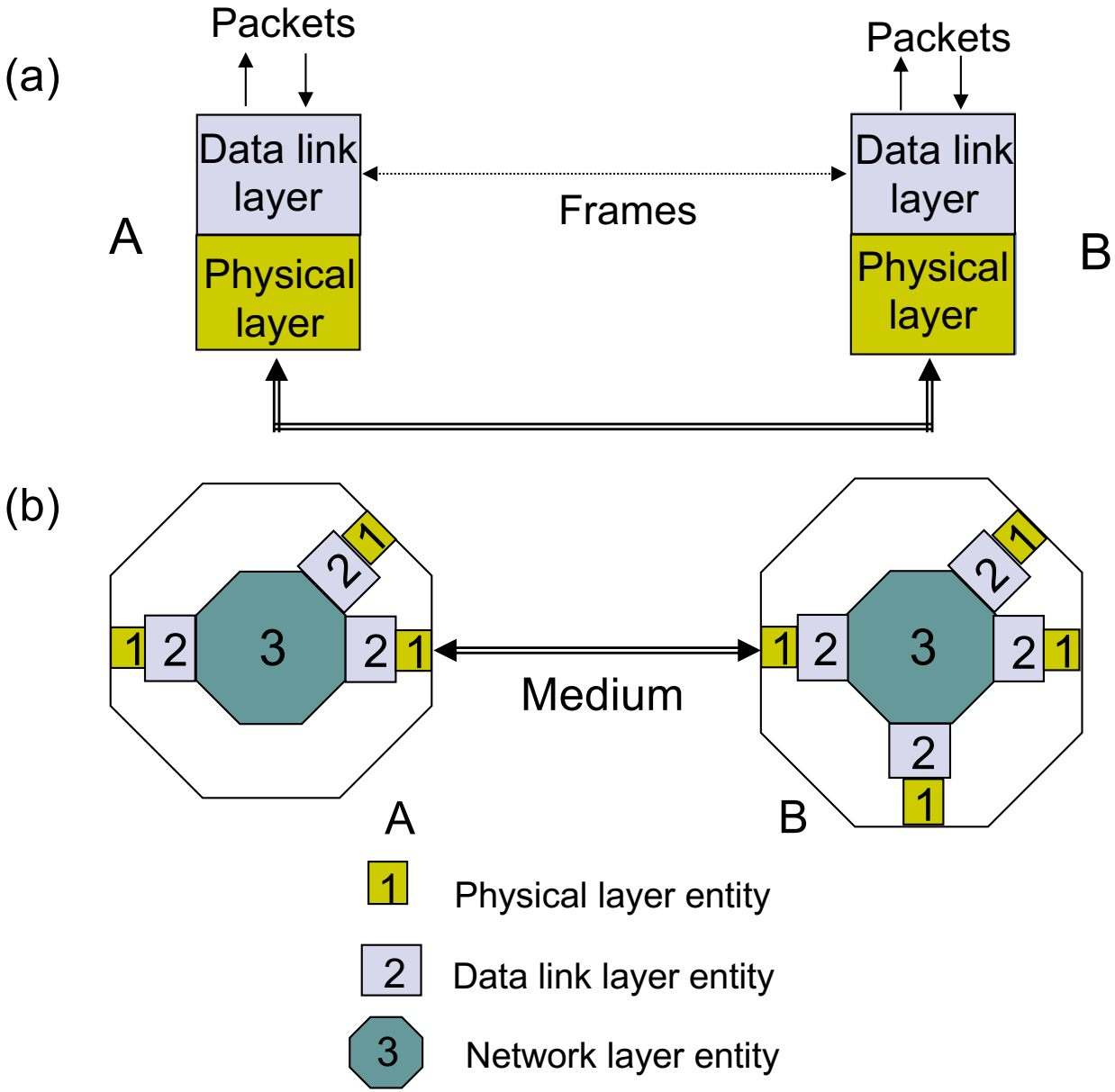


Stop-and-Wait *(pp 282-300)*

Selective Repeat *(pp 300-315)*



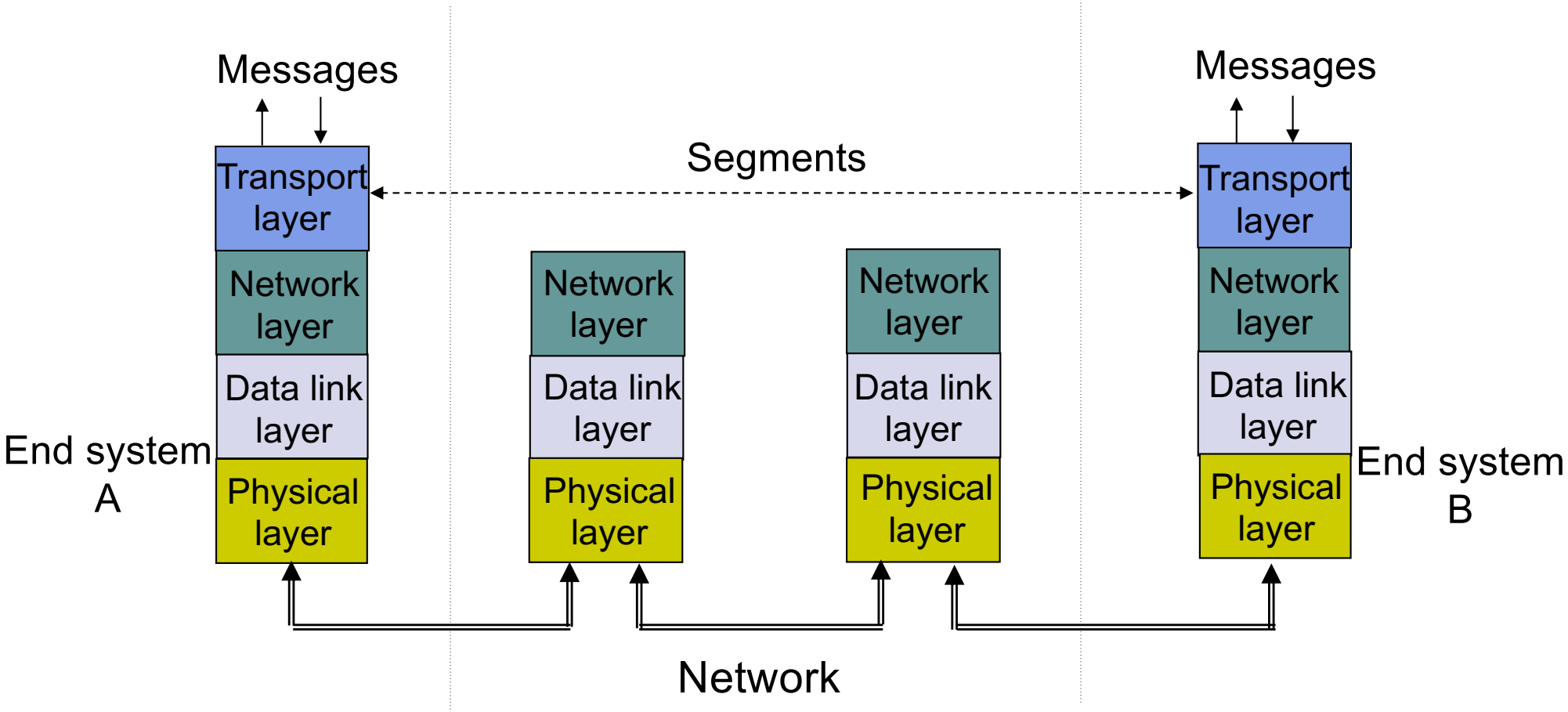
Error control in Data Link Layer



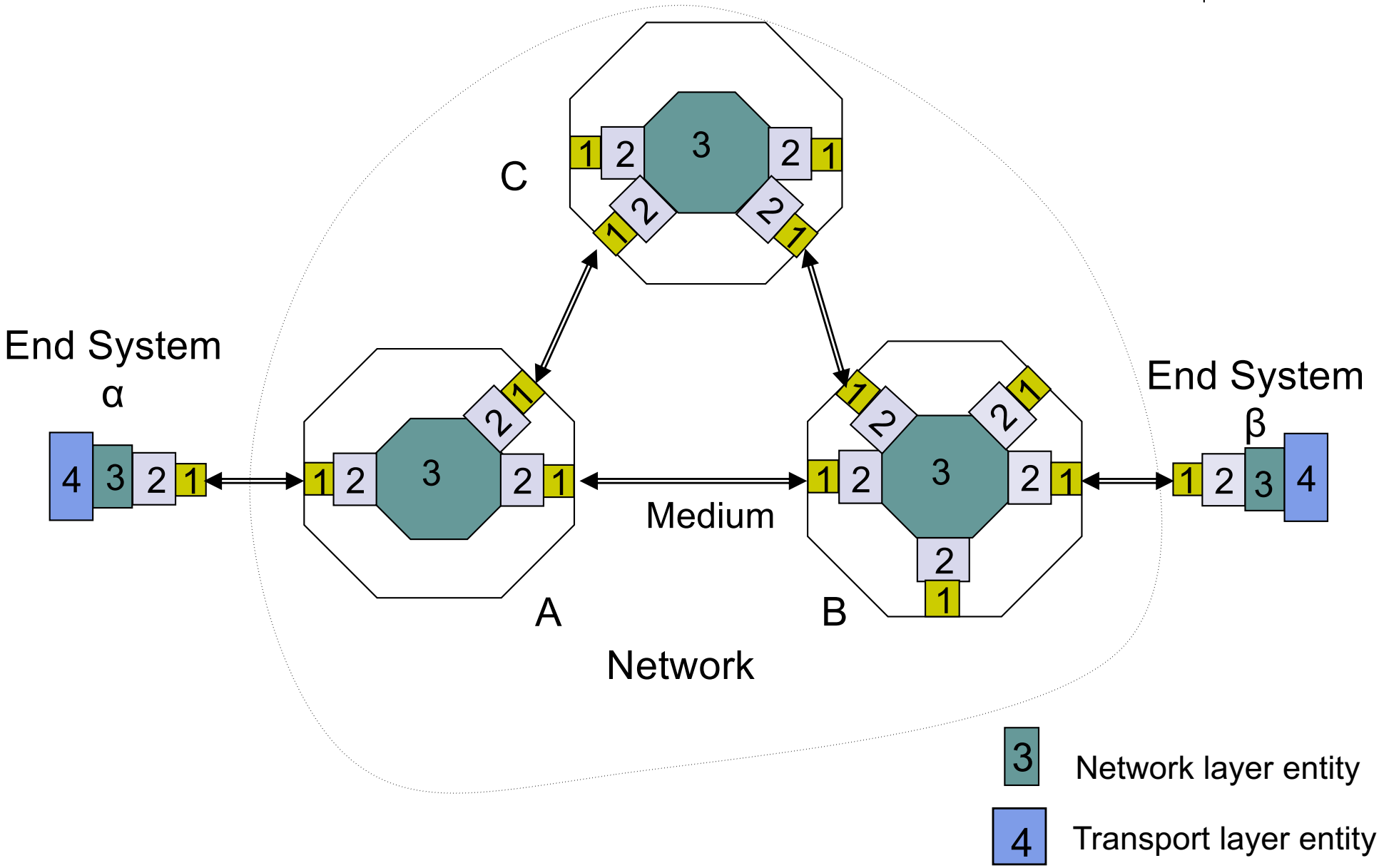
- Data Link operates over wire-like, directly-connected systems
- Frames can be corrupted or lost, but ***arrive in order***
- Data link performs error-checking & retransmission
- Ensures error-free packet transfer between two systems

Error Control in Transport Layer

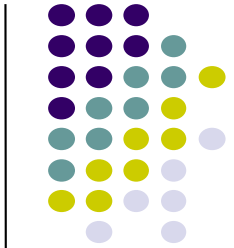
- Transport layer protocol (e.g. TCP) sends segments across network and performs end-to-end error checking & retransmission
- Underlying network is assumed to be unreliable



- Segments can experience long delays, be lost, or ***arrive out-of-order*** because packets can follow different paths across network
- End-to-end error control protocol more difficult



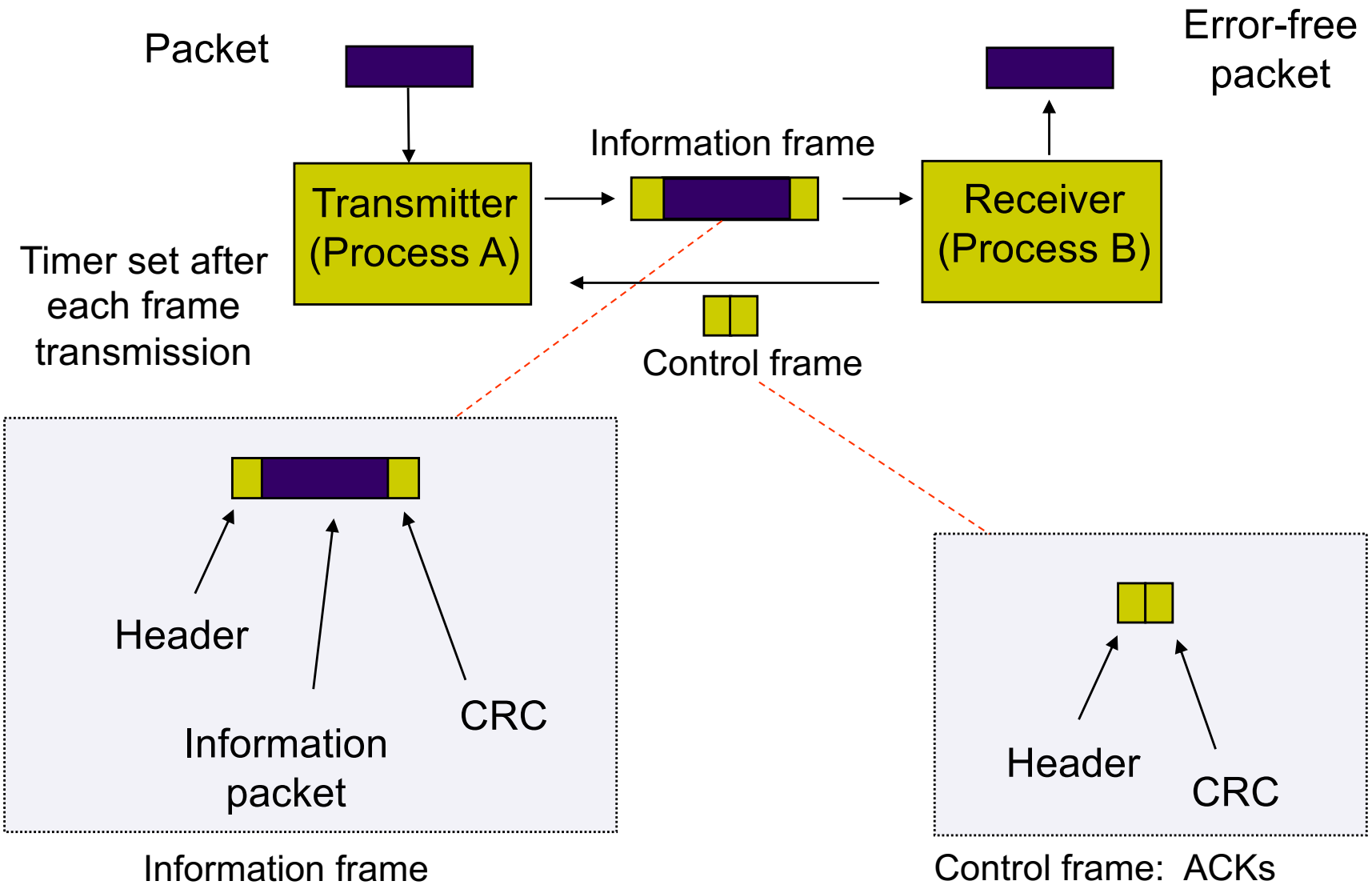
Automatic Repeat Request (ARQ)



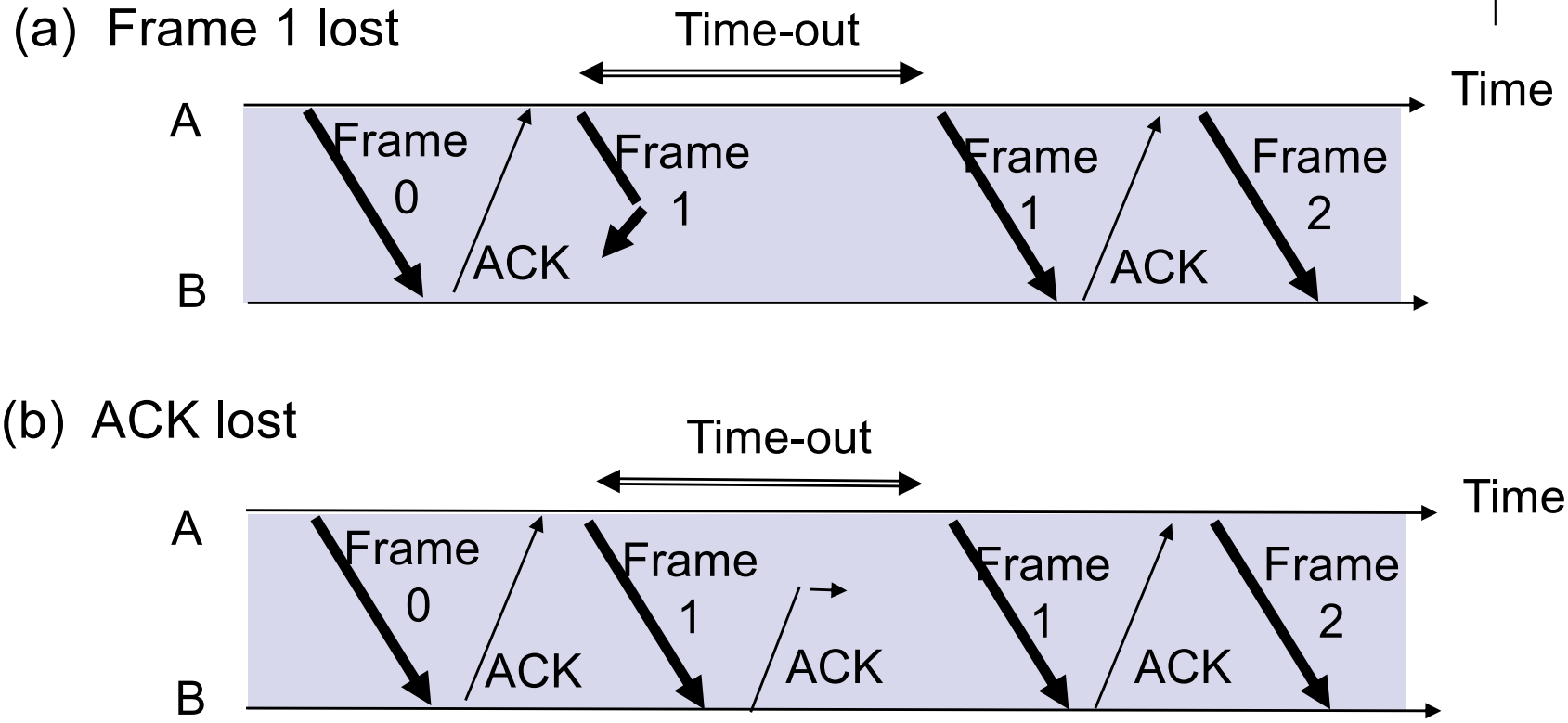
- *Purpose*: to ensure a sequence of information blocks (frames) is delivered in order and without errors or duplications despite transmission errors & losses
- We will look at:
  - Stop-and-Wait ARQ
  - Selective Repeat ARQ
- Basic elements of ARQ:
  - *Error-detecting code* with high error coverage
  - *ACKs* (positive acknowledgments) used extensively
  - *Timeout mechanism* used extensively

Stop-and-Wait ARQ

Transmit a frame, wait for ACK



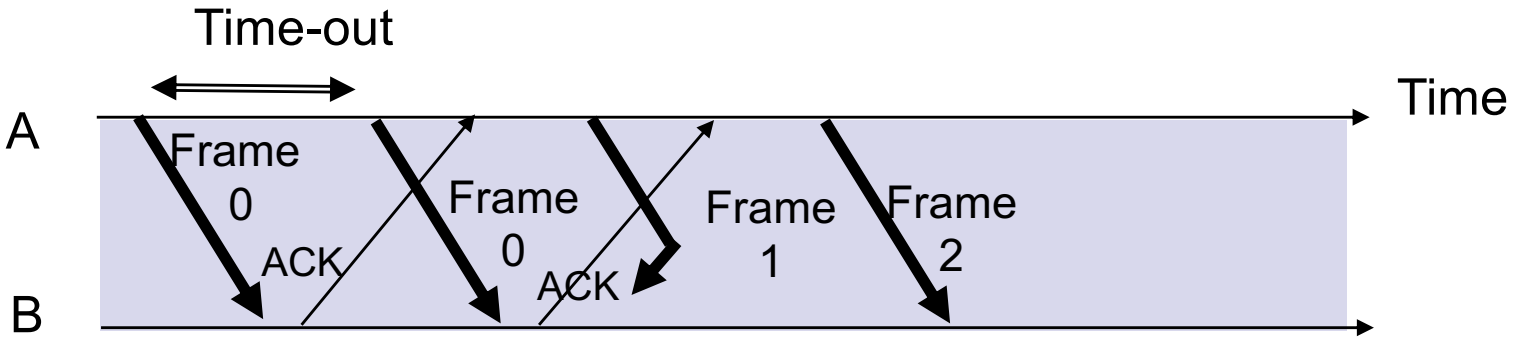
Need for Sequence Numbers



- In cases (a) & (b) the transmitting station A acts the same way
- But in case (b) the receiving station B accepts frame 1 twice
- Question: How is the receiver to know the second frame is also frame 1?
- Answer: ***Add frame sequence number in header***
- S_{last} is sequence number of most recent transmitted frame

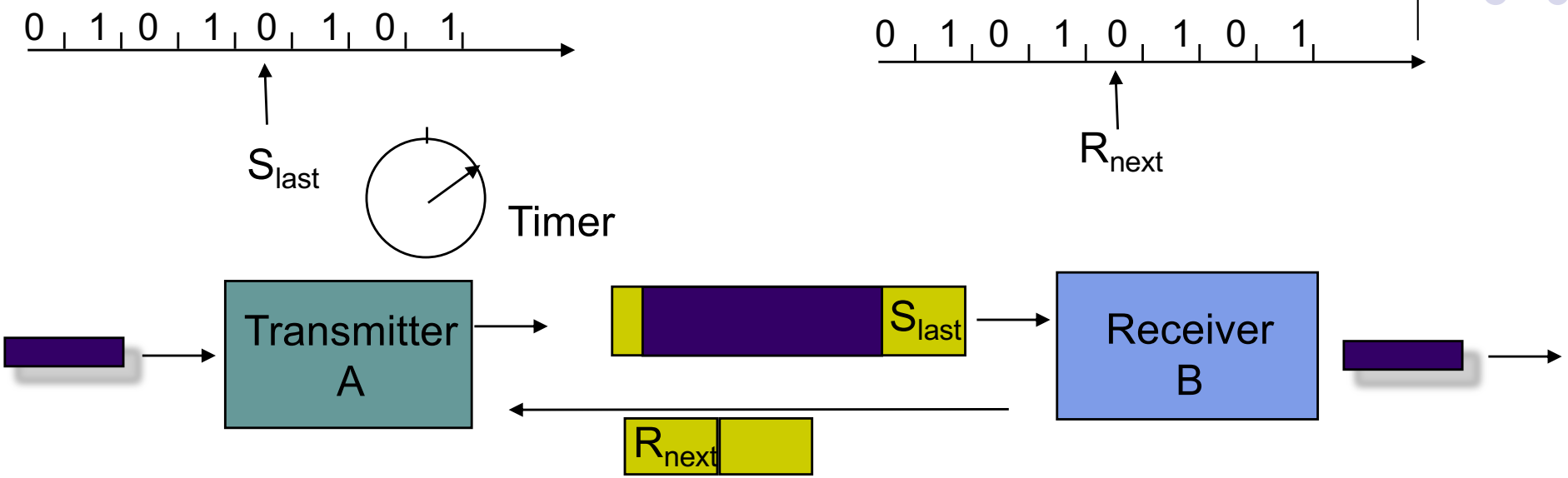
Sequence Numbers

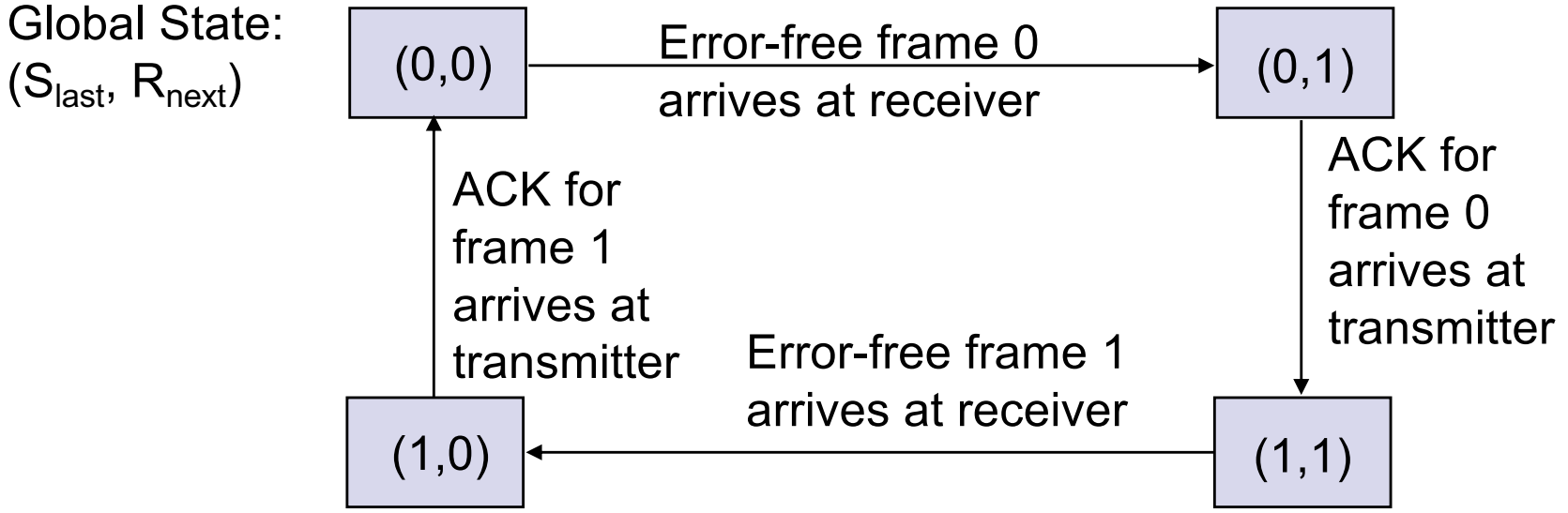
(c) Premature Time-out



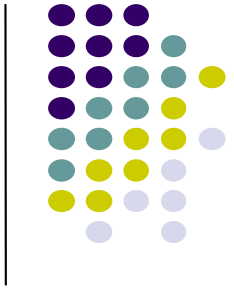
- The transmitting station A misinterprets duplicate ACKs
- Incorrectly assumes second ACK acknowledges Frame 1
- Question: How is the receiver to know second ACK is for frame 0?
- Answer: ***Add frame sequence number in ACK header***
- R_{next} is sequence number of next frame expected by the receiver
- Implicitly acknowledges receipt of all prior frames

1-Bit Sequence Numbering Suffices





Stop-and-Wait ARQ



Transmitter

Ready state

- Await request from higher layer for packet transfer
- When request arrives, transmit frame with updated S_{last} and CRC
- Go to Wait State

Wait state

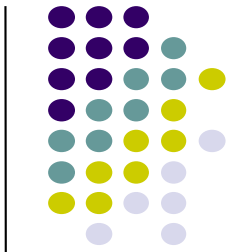
- Wait for ACK or timer to expire; block requests from higher layer
- If timeout expires
  - retransmit frame and reset timer
- If ACK received:
  - If sequence number is incorrect or if errors detected: ignore ACK
  - If sequence number is correct (R_{next} = S_{last} +1): accept frame, go to Ready state

Receiver

Always in Ready State

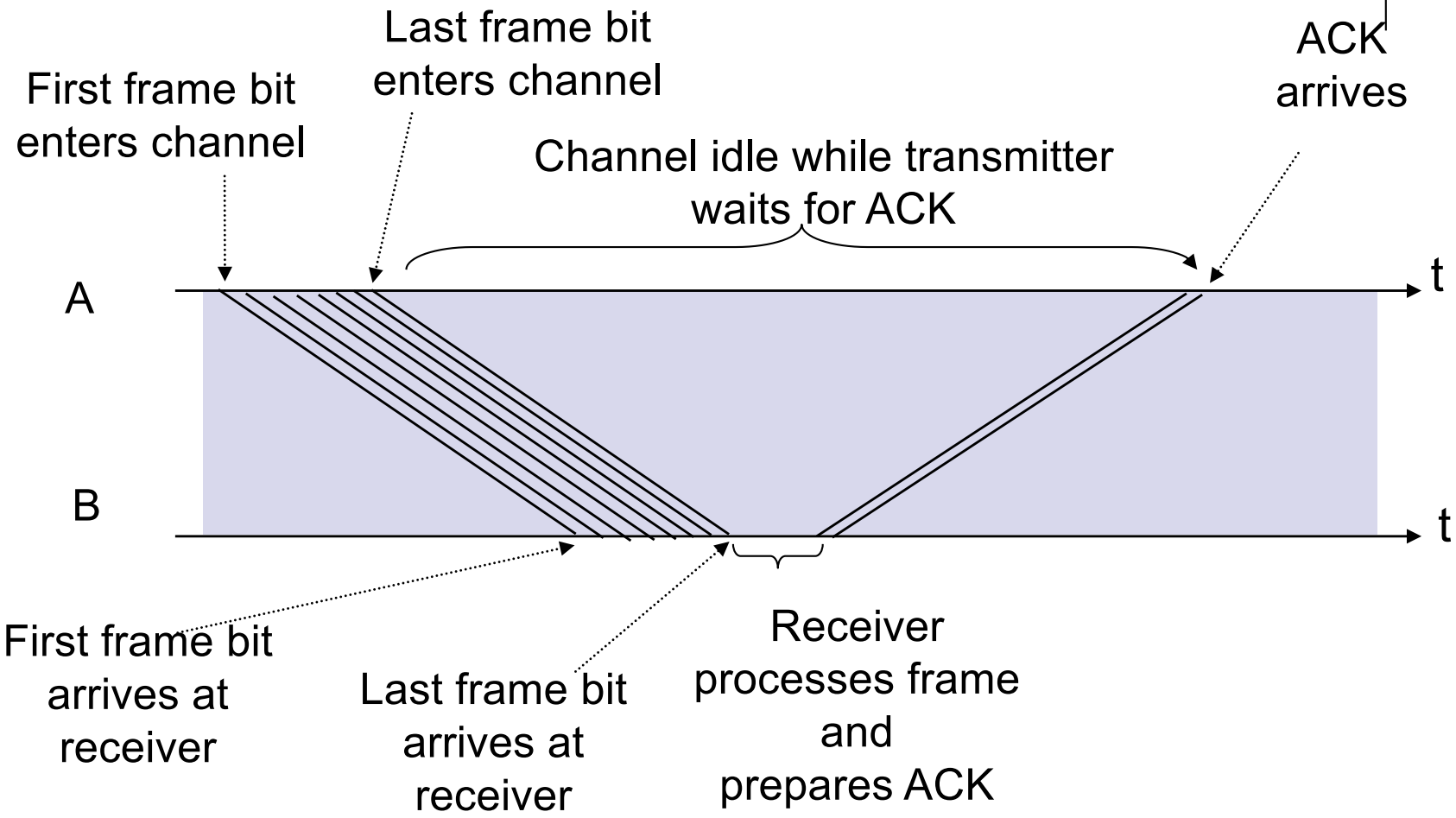
- Wait for arrival of new frame
- When frame arrives, check for errors
- If no errors detected and sequence number is correct (S_{last}=R_{next}), then
  - accept frame,
  - update R_{next},
  - send ACK frame with R_{next},
  - deliver packet to higher layer
- If no errors detected and wrong sequence number
  - discard frame
  - send ACK frame with R_{next}
- If errors detected
  - discard frame

Applications of Stop-and-Wait ARQ



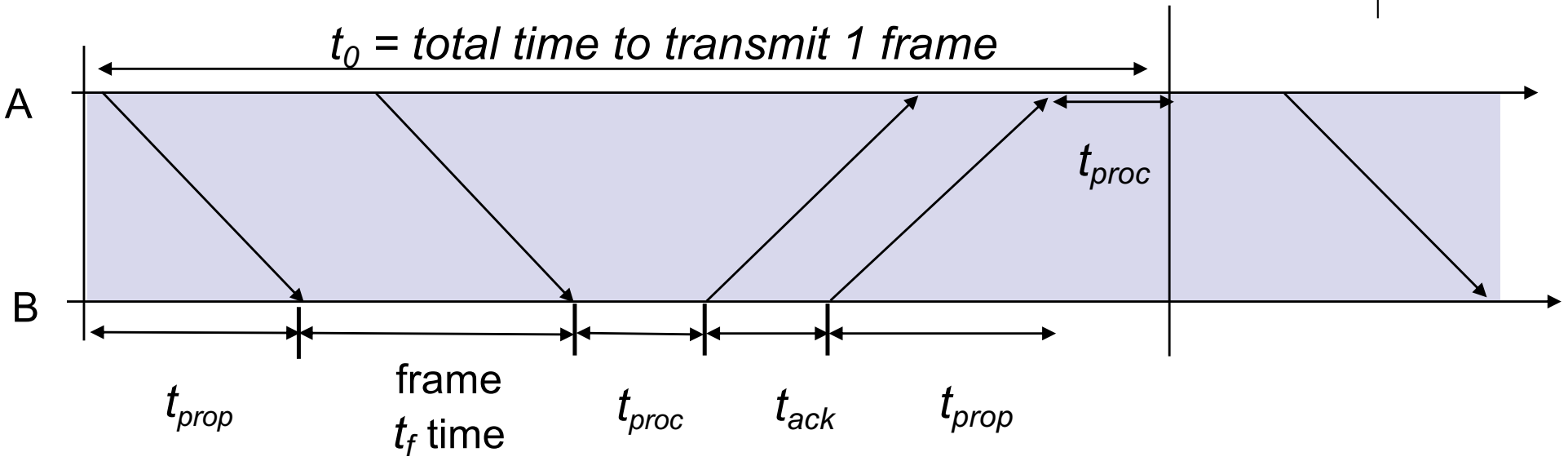
- IBM *Binary Synchronous Communications protocol* (Bisync): character-oriented data link control
- *Xmodem*: modem file transfer protocol
- *Trivial File Transfer Protocol* (RFC 1350): simple protocol for file transfer over UDP

Stop-and-Wait Efficiency



- 10000 bit frame @ 1 Mbps takes 10 ms to transmit
- If wait for ACK = 1 ms, then efficiency = 10/11= 91%
- If wait for ACK = 20 ms, then efficiency =10/30 = 33%

Stop-and-Wait Model



$$t_0 = 2t_{prop} + 2t_{proc} + t_f + t_{ack}$$

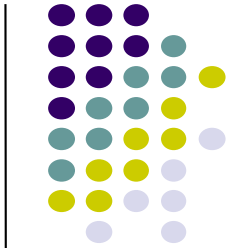
$$= 2t_{prop} + 2t_{proc} + \frac{n_f}{R} + \frac{n_a}{R}$$

bits/info frame

bits/ACK frame

channel transmission rate

S&W Efficiency on Error-free channel



Effective transmission rate:

bits for header & CRC

$$R_{eff}^{0} = \frac{\text{number of information bits delivered to destination}}{\text{total time required to deliver the information bits}} = \frac{n_f - n_o}{t_0},$$

Transmission efficiency:

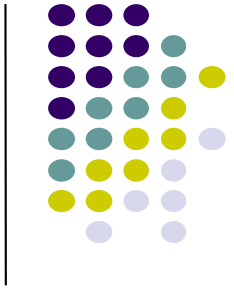
$$\eta_0 = \frac{R_{eff}}{R} = \frac{\dfrac{n_f - n_o}{t_0}}{R} = \frac{1 - \dfrac{n_o}{n_f}}{1 + \dfrac{n_a}{n_f} + \dfrac{2(t_{prop} + t_{proc})R}{n_f}}.$$

Effect of frame overhead

Effect of ACK frame

Effect of ***Delay-Bandwidth Product***

Example: Impact of Delay-Bandwidth Product

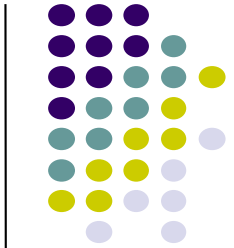


n_f=1250 bytes = 10000 bits, n_a=n_o=25 bytes = 200 bits

2xDelayxBW Efficiency	1 ms 200 km	10 ms 2000 km	100 ms 20000 km	1 sec 200000 km
1 Mbps	10^3 88%	10^4 49%	10^5 9%	10^6 1%
1 Gbps	10^6 1%	10^7 0.1%	10^8 0.01%	10^9 0.001%

Stop-and-Wait does not work well for very high speeds or long propagation delays

Geometric RV



X=# transmissions till first success

- X=1, $(1-P_f)$
- X=2, $(1-P_f)P_f$
- X=3, $(1-P_f)P_f^2$
- …
- X=k, $(1-P_f)P_f^{k-1}$
- …

Mean of X

E[X]

$= (1-P_f) + 2(1-P_f)P_f +$

$+\ldots+ k(1-P_f)P_f^{k-1} +\ldots$

$E[X]/ (1-P_f)$

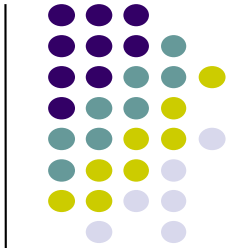
$= 1 + 2P_f +\ldots+ kP_f^{k-1} +\ldots$

$=d/dx \{P_f + P_f^2+\ldots+ P_f^k +\ldots\}$

$=1/(1-P_f)^2$

$E[X] = 1/(1-P_f)$

S&W Efficiency in Channel with Errors

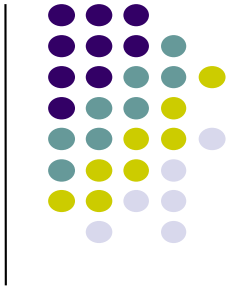


- Let 1 – P_f = probability frame arrives w/o errors
- Avg. # of transmissions to first correct arrival is then 1/ (1–P_f)
- "If 1-in-10 get through without error, then avg. 10 tries to success"
- Avg. Total Time per frame is then $t_0/(1 - P_f)$

$$\eta_{SW} = \frac{R_{eff}}{R} = \frac{\frac{n_f - n_o}{t_0}}{\frac{1 - P_f}{R}} = \frac{1 - \frac{n_o}{n_f}}{1 + \frac{n_a}{n_f} + \frac{2(t_{prop} + t_{proc})R}{n_f}} (1 - P_f)$$

Effect of frame loss

Example: Impact Bit Error Rate



n_f=1250 bytes = 10000 bits, n_a=n_o=25 bytes = 200 bits

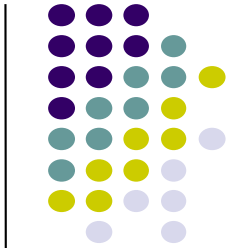
Find efficiency for random bit errors with p=0, 10^{-6}, 10^{-5}, 10^{-4}

$$1 - P_f = (1-p)^{n_f} \approx e^{-n_f p} \quad \text{for large } n_f \text{ and small } p$$

$1 - P_f$ **Efficiency**	0	10^{-6}	10^{-5}	10^{-4}
1 Mbps	1	0.99	0.905	0.368
& 1 ms	88%	86.6%	79.2%	32.2%

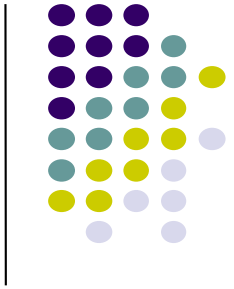
Bit errors impact performance as $n_f p$ *approach 1*

Improvement: Don't Wait!

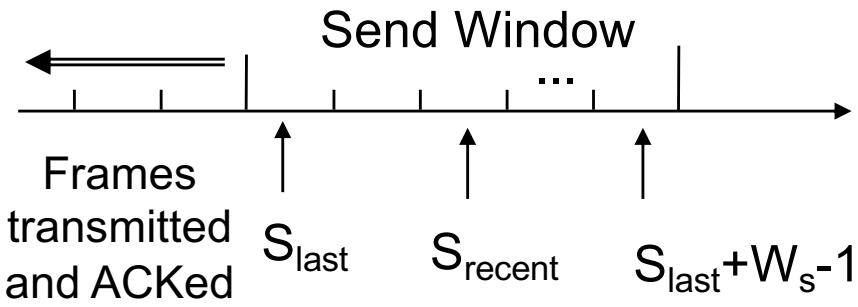


- Keep channel busy by continuing to send frames
- Allow a window of up to W_s outstanding frames
- Use *m*-bit sequence numbering
- If ACK for oldest frame arrives before window is exhausted, we can continue transmitting
- Many design options:
  - Use of timeout mechanism
  - What to ACK?
  - How to deal with out-of-order frames?
  - When and What to Retransmit?

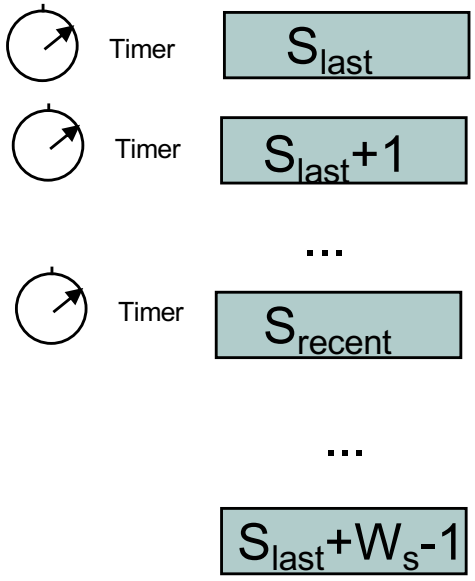
Send Many, Receive One



Transmitter



Buffers



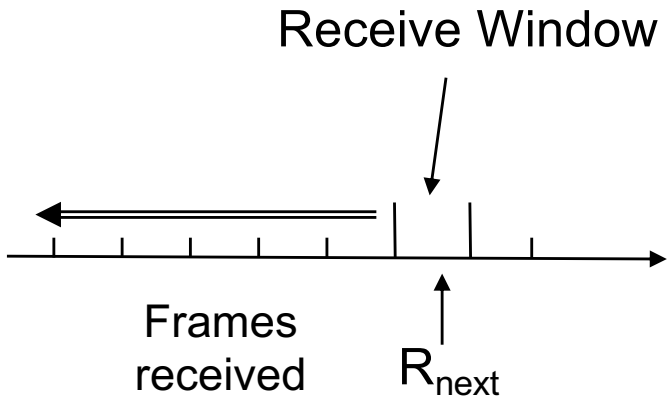
oldest un-ACKed frame

most recent transmission

max Seq # allowed

Maximum Send Window Size is $W_s = 2^m-1$

Receiver

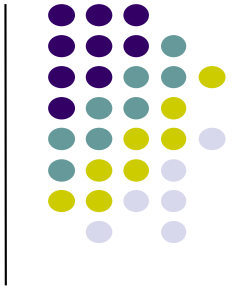


Receiver will only accept a frame that is error-free and that has sequence number R_{next}

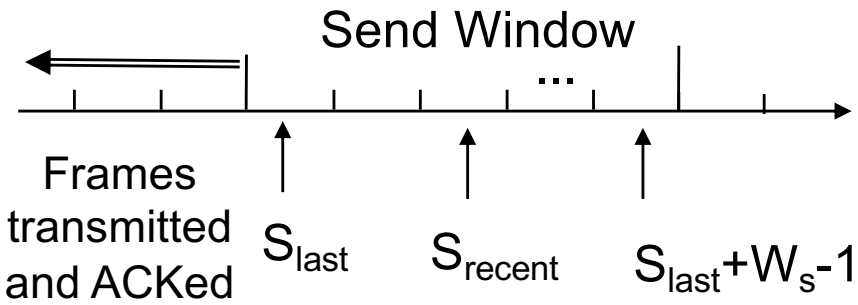
When such frame arrives R_{next} is incremented by one, so the *receive window slides forward* by one

If error-free, but out-of-sequence frame arrives, ACK with R_{next} is sent

Sliding Window Operation



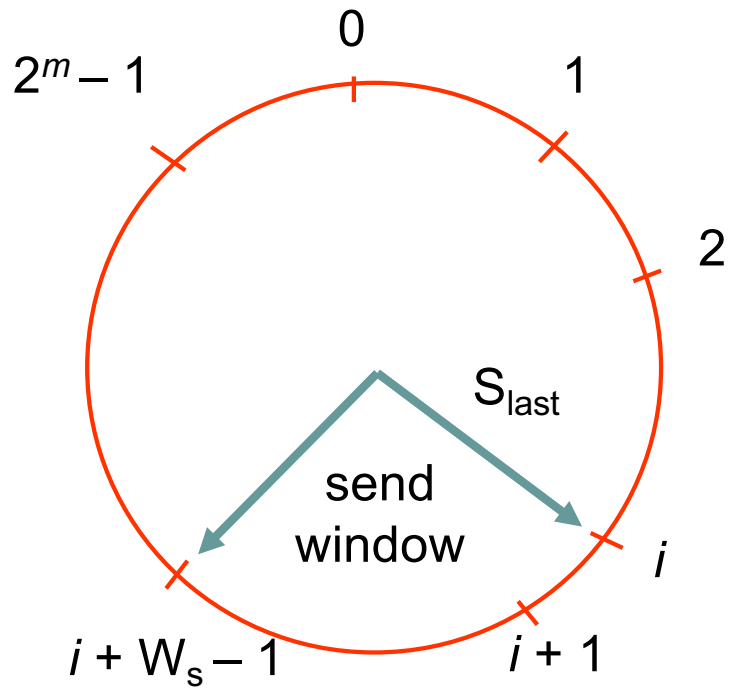
Transmitter



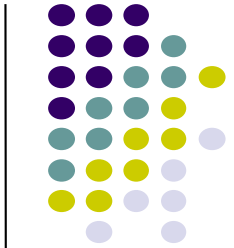
When error-free ACK frame arrives with R_{next} in interval between S_{last} and S_{recent}, then S_{last} is moved to R_{next}, since all frames prior to R_{next} have been received correctly.

Note that send window can move forward by more than one.

m-bit Sequence Numbering



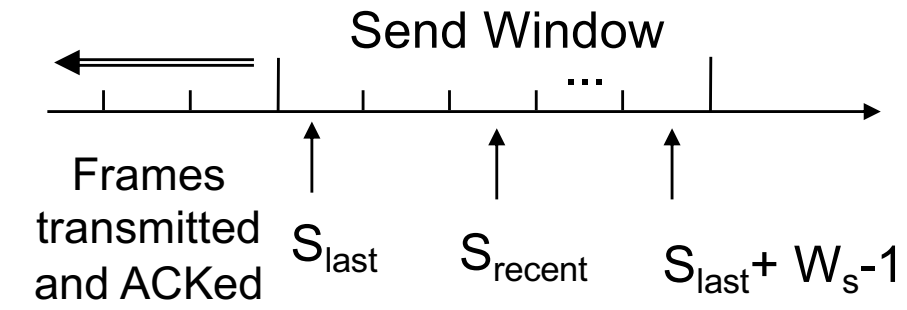
Selective Repeat ARQ: Send Many, Receive Many

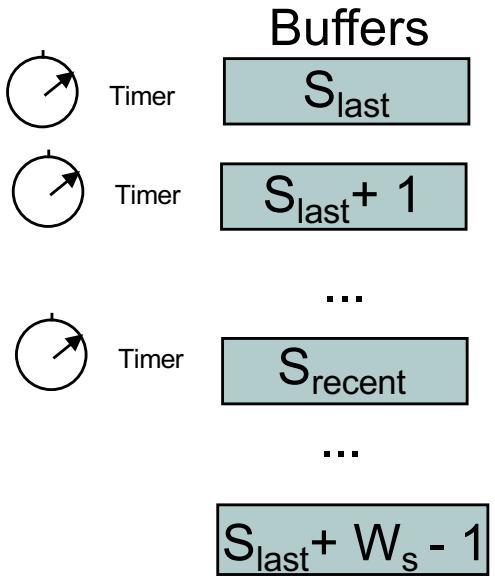


- Selective Repeat retransmits *only missing frames*
  - Timeout causes individual corresponding frame to be resent
- Receiver maintains a *receive window* W_r of sequence numbers that can be accepted
  - Error-free, but out-of-sequence frames with sequence numbers within the receive window are buffered
  - Arrival of frame with R_{next} causes window to slide forward by 1 or more
- $W_r + W_s \leq 2^m$ for unambiguous delivery

Selective Repeat ARQ

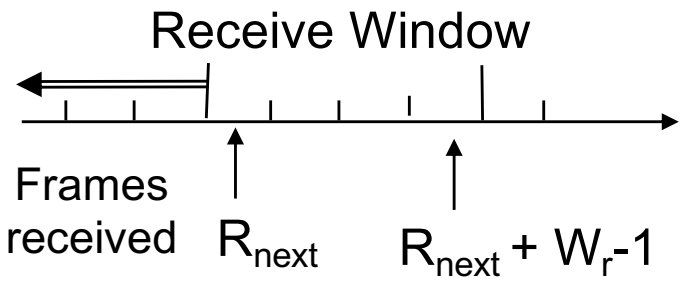
Transmitter

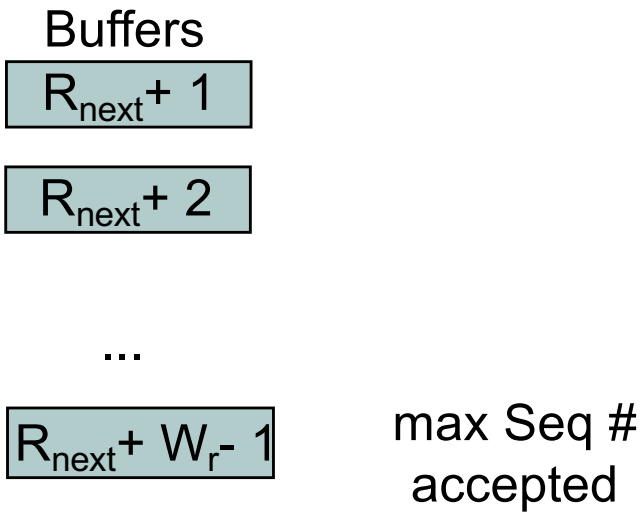




If error-free ACK frame arrives & Rnext in send window, then move Slast to Rnext. Update status of all frames in send window

Receiver

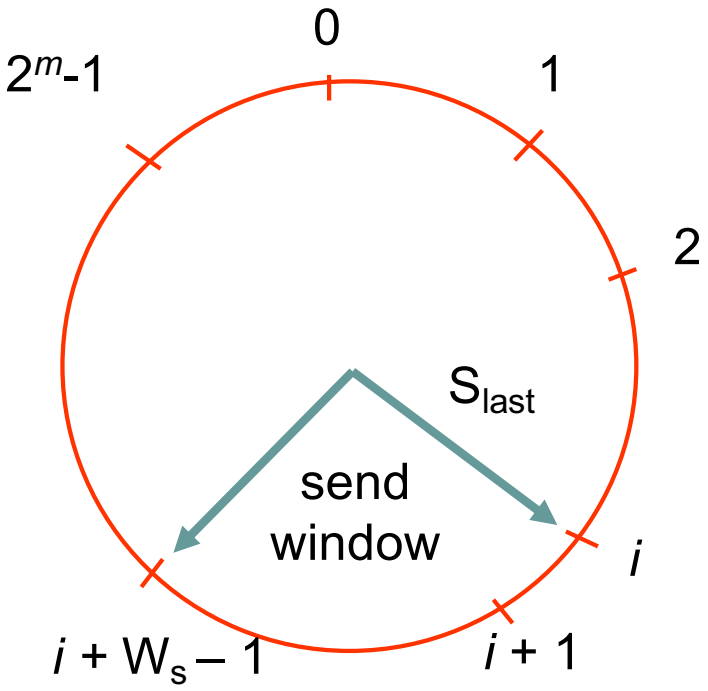




Accept error-free frames in receive window; Send ACK with new R_{next} And list of missing frames

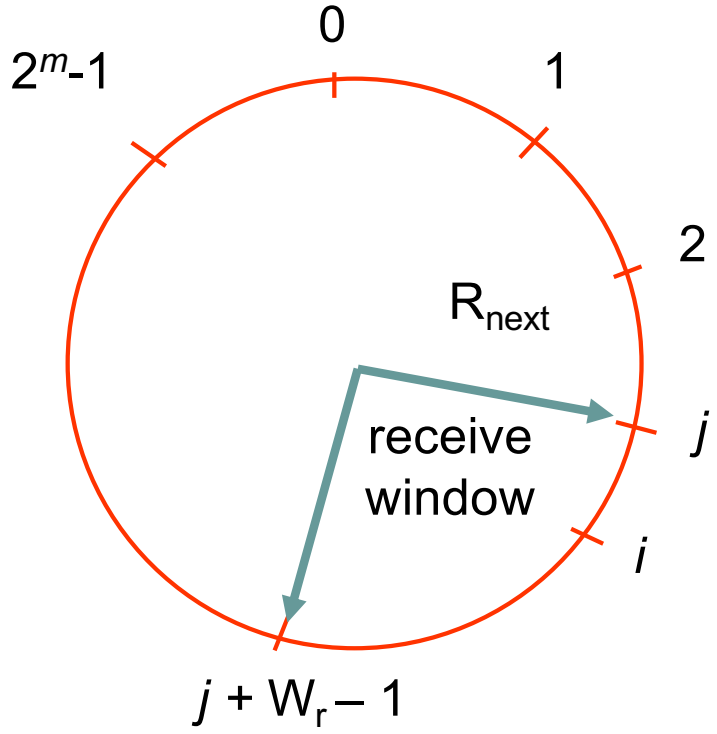
Send & Receive Windows

Transmitter



Moves k forward when ACK arrives with $R_{next} = S_{last} + k$
$k = 1, \ldots, W_s\text{-}1$

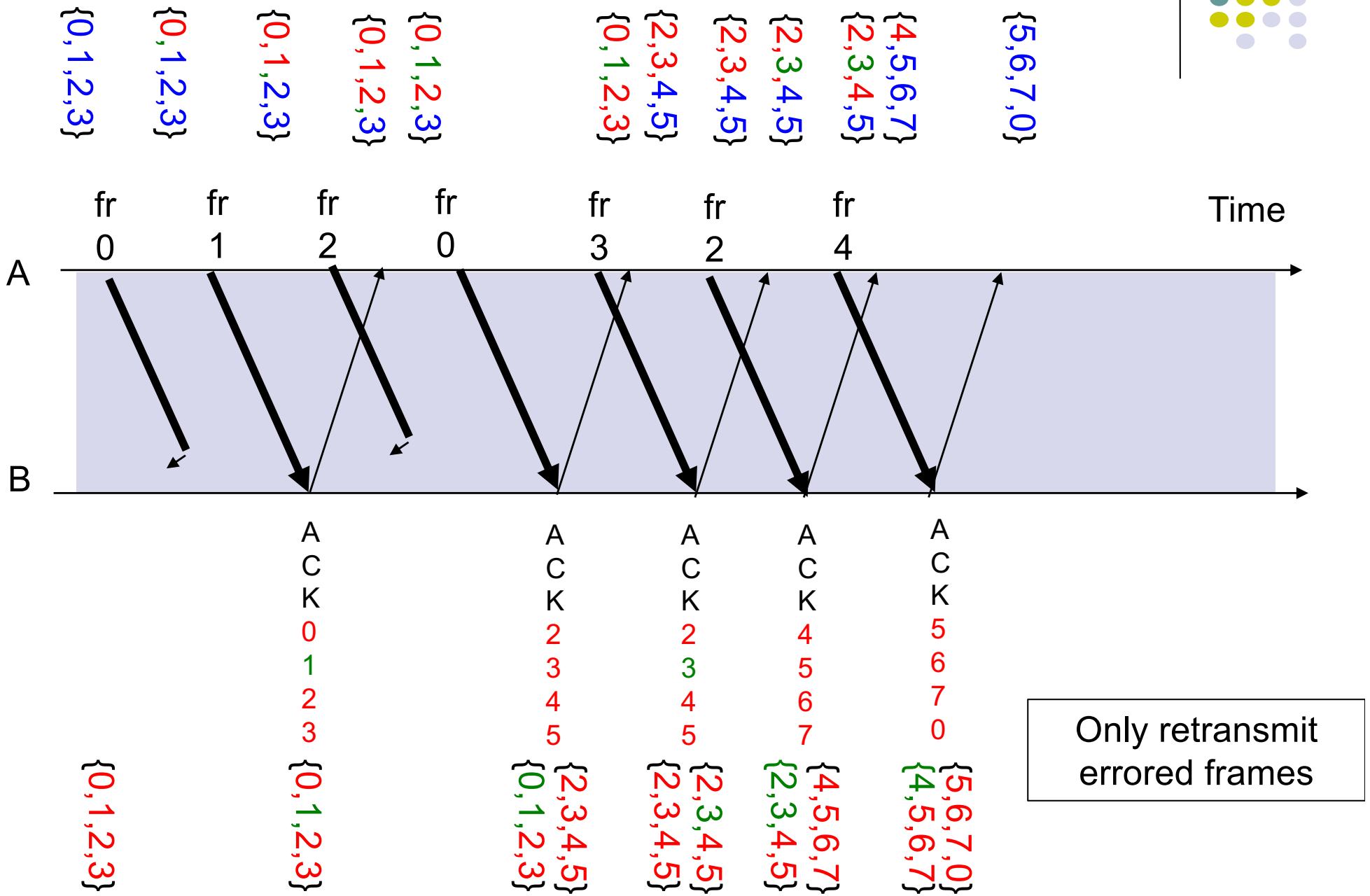
Receiver



Moves forward by 1 or more when frame arrives with Seq. # = R_{next}

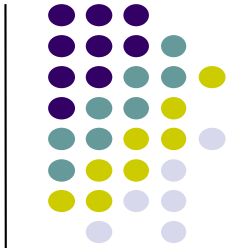
Selective Repeat ARQ

M = 2^3 = 8, *Ws*=4 , *Wr*=4



Only retransmit errored frames

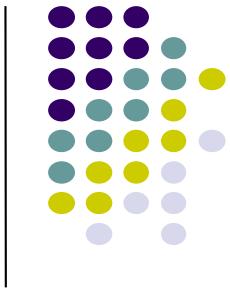
Efficiency of Selective Repeat



- Assume P_f frame loss probability, then number of transmissions required to deliver a frame is:
  - $t_f / (1-P_f)$

$$\eta_{SR} = \frac{\frac{n_f - n_o}{t_f/(1-P_f)}}{R} = (1-\frac{n_o}{n_f})(1-P_f)$$

Example: Impact Bit Error Rate on Selective Repeat



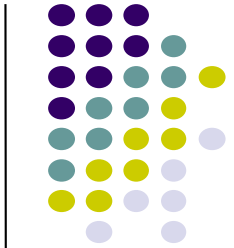
n_f=1250 bytes = 10000 bits, n_a=n_o=25 bytes = 200 bits

Compare S&W & SR efficiency for random bit errors with p=0, 10^{-6}, 10^{-5}, 10^{-4} and R= 1 Mbps & 100 ms

Efficiency	0	10^{-6}	10^{-5}	10^{-4}
S&W	8.9%	8.8%	8.0%	3.3%
SR	98%	97%	89%	36%

- *Selective Repeat outperforms S&W, but efficiency drops as error rate increases*

Comparison of ARQ Efficiencies



Assume

n_a & n_o are negligible relative to n_f,

$L = 2(t_{prop}+t_{proc})R/n_f = (W_s-1)$, then

Selective-Repeat:

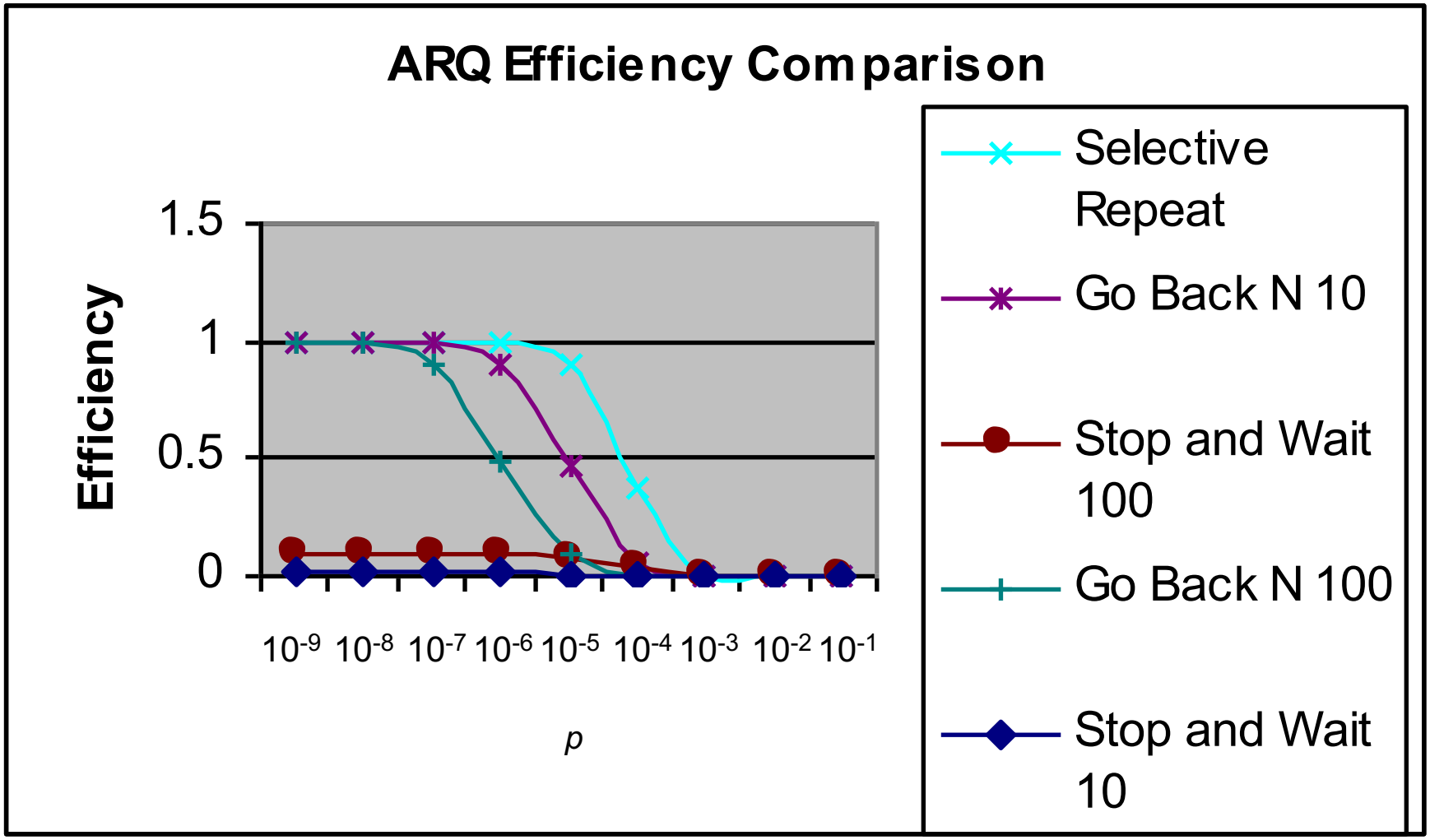
$$\eta_{SR} = (1-P_f)(1-\frac{n_o}{n_f}) \approx (1-P_f)$$

For $P_f \rightarrow 1$, GBN & SW same

Stop-and-Wait:

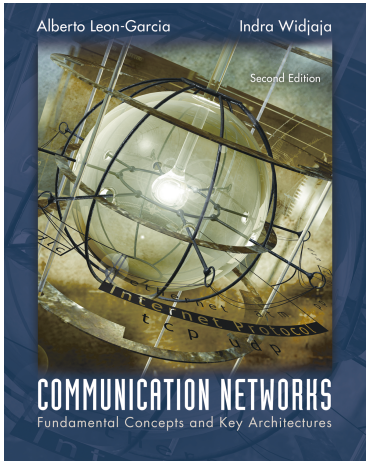
$$\eta_{SW} = \frac{(1-P_f)}{1+\frac{n_a}{n_f}+\frac{2(t_{prop}+t_{proc})R}{n_f}} \approx \frac{1-P_f}{1+L}$$

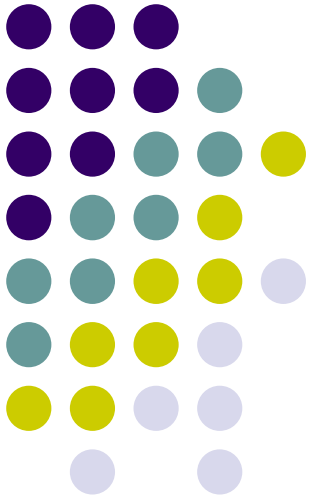
ARQ Efficiencies



Delay-Bandwidth product = 10, 100

TCP Protocol





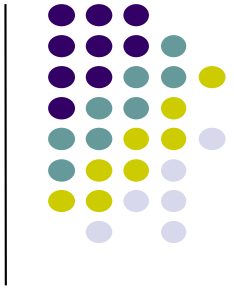
Reliable Stream Service

Flow Control

Congestion control

pp 320-324, 602-620

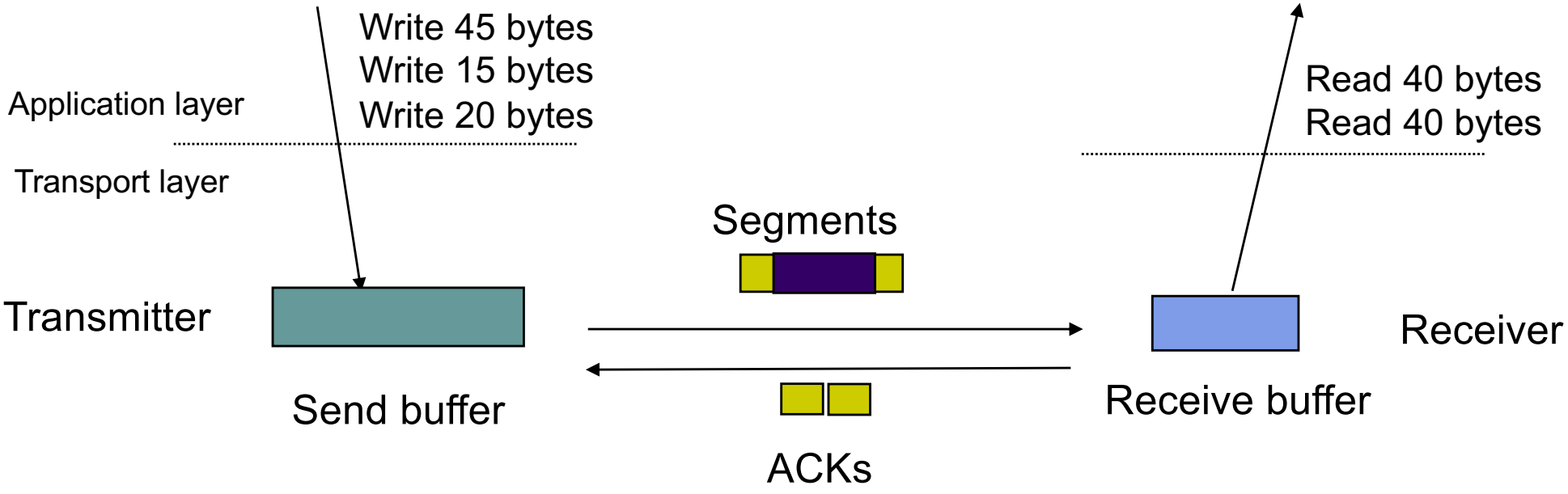
TCP Reliable Stream Service



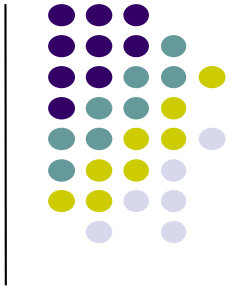
TCP transfers byte stream in order, without errors or duplications

Application Layer writes bytes into send buffer through socket

Application Layer reads bytes from receive buffer through socket

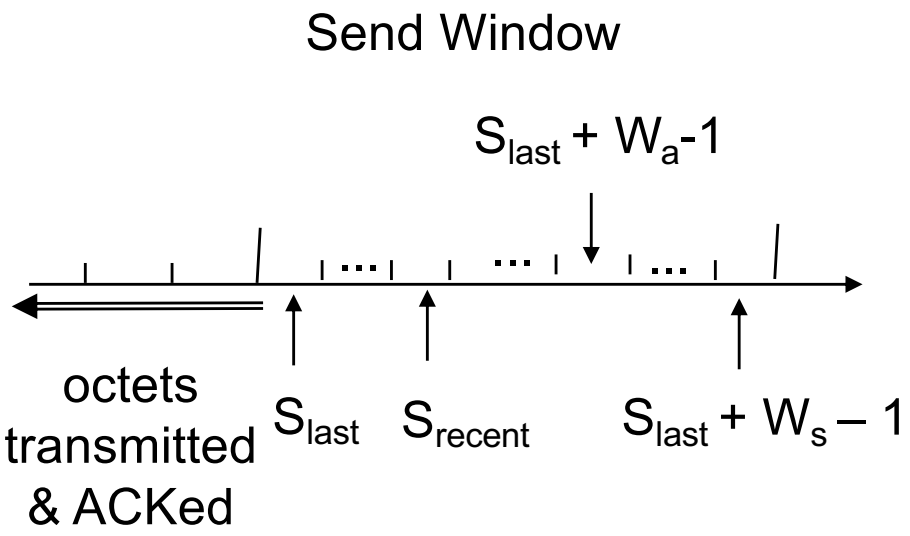


TCP ARQ Method



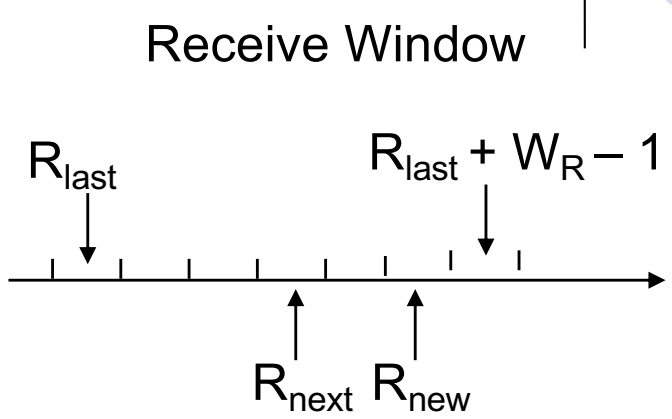
- TCP uses variation of *Selective Repeat ARQ*
  - Transfers ***byte stream*** without preserving boundaries
  - Accepts segments within a receive window
  - Uses R_{next} for acknowledgements
- Operates over best effort service of IP
  - Packets can arrive with errors or be lost
  - Packets can arrive out-of-order
  - Packets can arrive after very long delays
  - Duplicate segments must be detected & discarded
  - Must protect against segments from previous connections
- Sequence Numbers
  - Seq. # is number of first byte in segment payload
  - Very long Seq. #s (32 bits) to deal with long delays
  - Initial sequence numbers negotiated during connection setup (to deal with very old duplicates)

TCP Sender



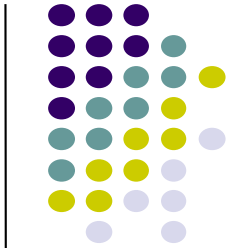
S_{last} oldest unacknowledged byte
S_{recent} highest-numbered transmitted byte
$S_{last}+W_a-1$ highest-numbered byte that can be transmitted
$S_{last}+W_s-1$ highest-numbered byte that can be accepted from the application

TCP Receiver



R_{last} highest-numbered byte not yet read by the application
R_{next} next expected byte
R_{new} highest numbered byte received correctly
$R_{last}+W_R-1$ highest-numbered byte that can be accommodated in receive buffer

TCP Connections

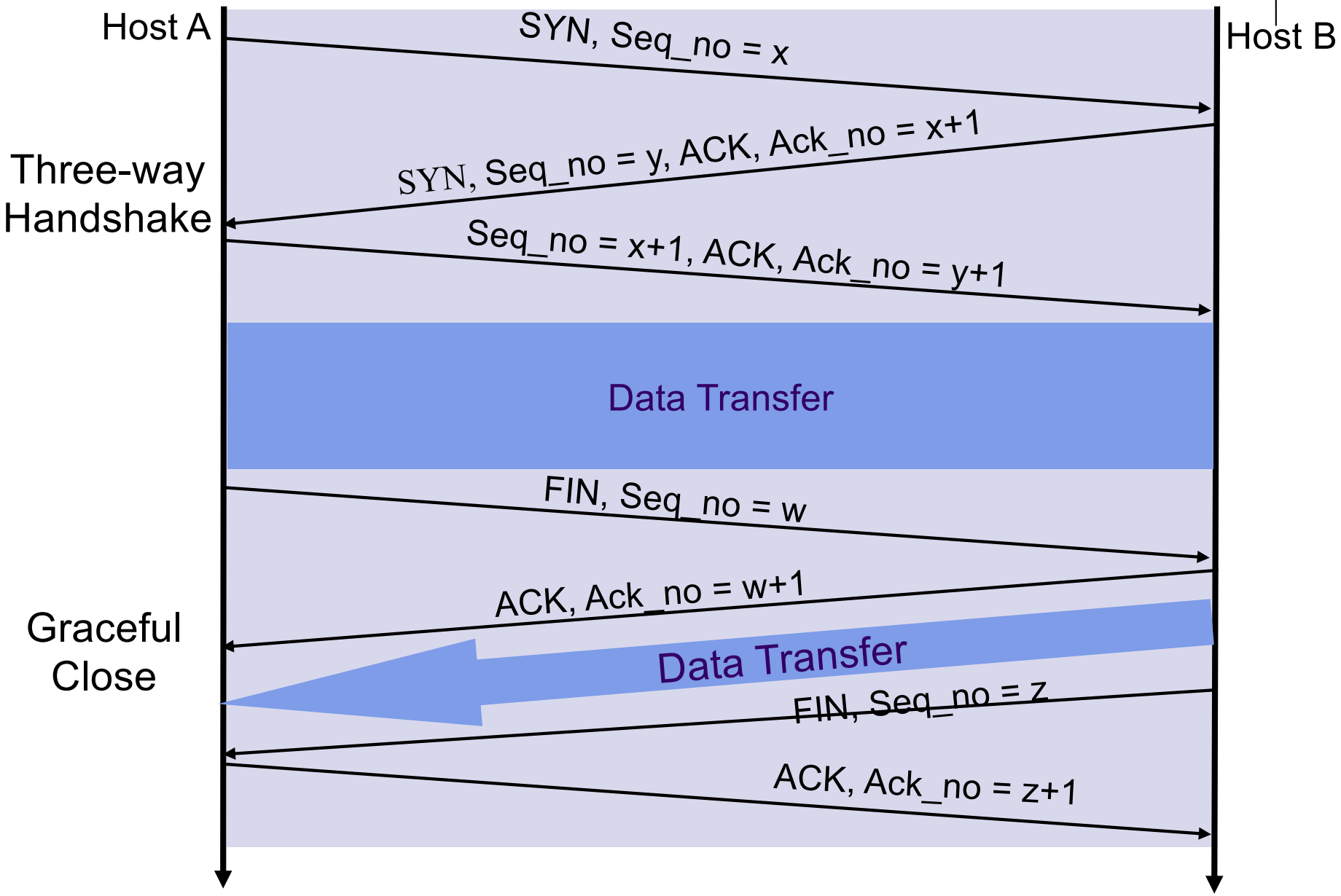


- TCP Connection
  - One connection each way
  - Each connection is identified uniquely by:

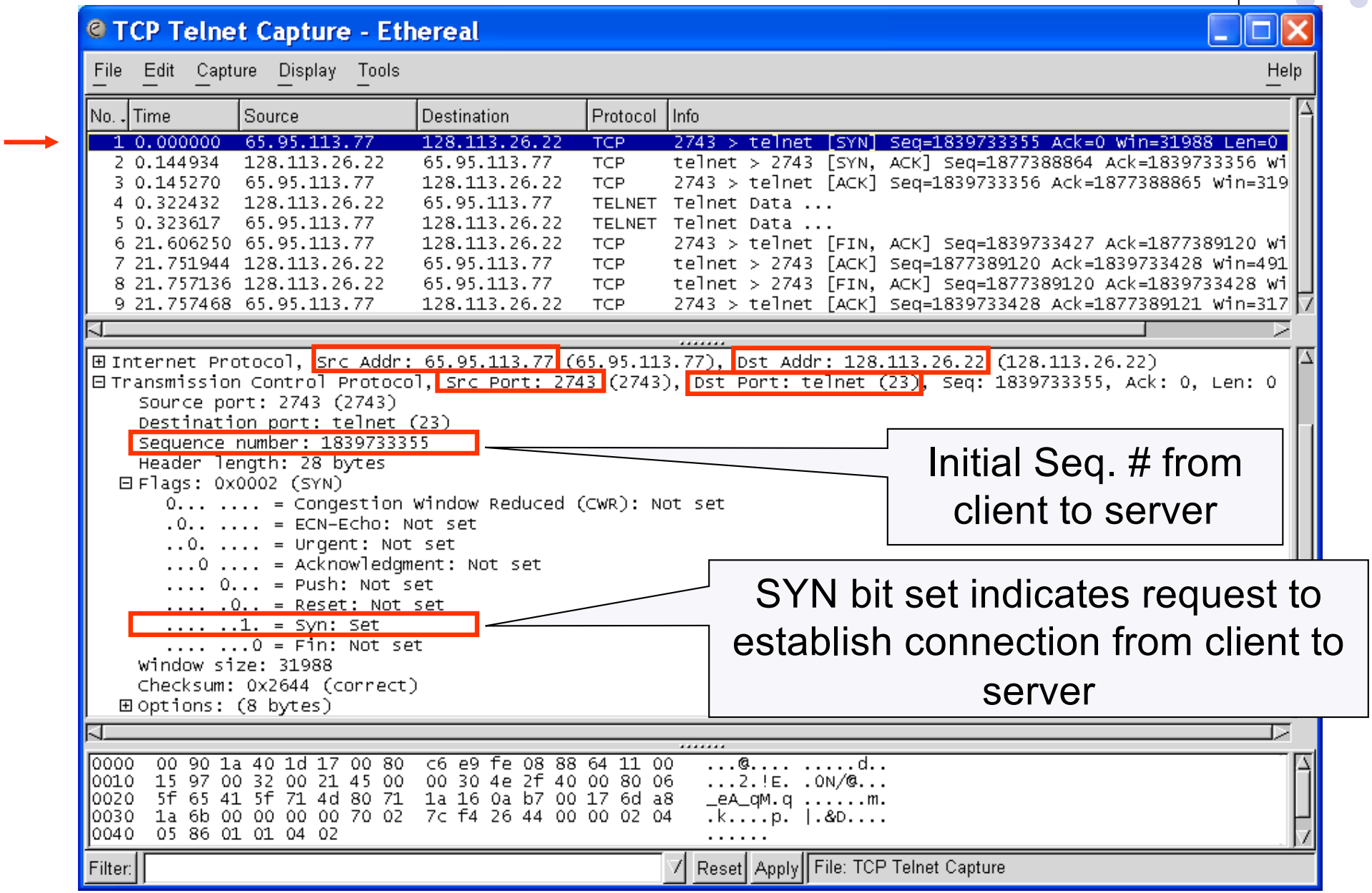
Send IP Address, Send TCP Port #, Receive IP Address, Receive TCP Port #

- Connection Setup with Three-Way Handshake
  - Three-way exchange to negotiate initial Seq. #'s for connections in each direction
- Data Transfer
  - Exchange segments carrying data
- Graceful Close
  - Close each direction separately

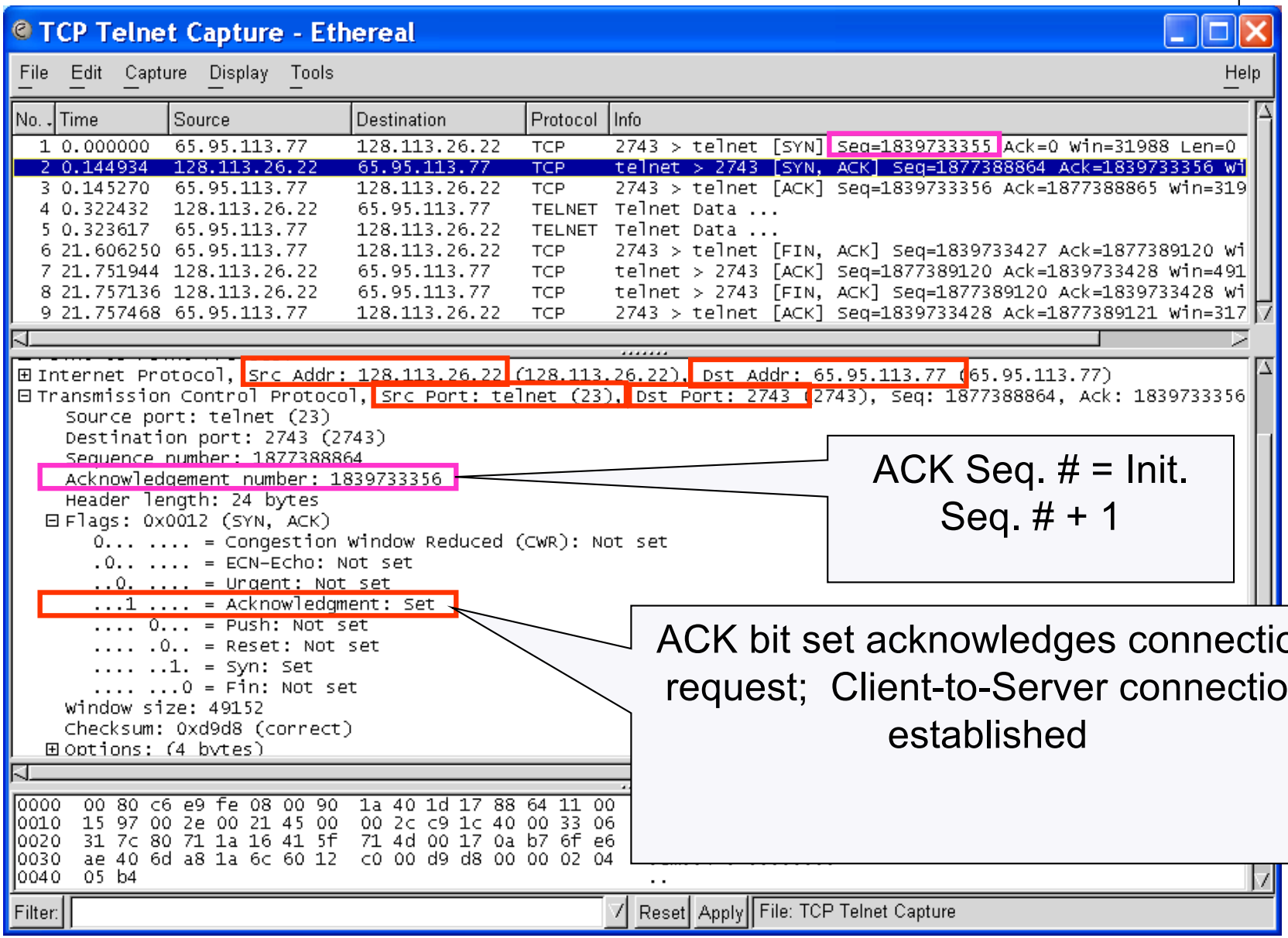
Three Phases of TCP Connection



1st Handshake: Client-Server Connection Request



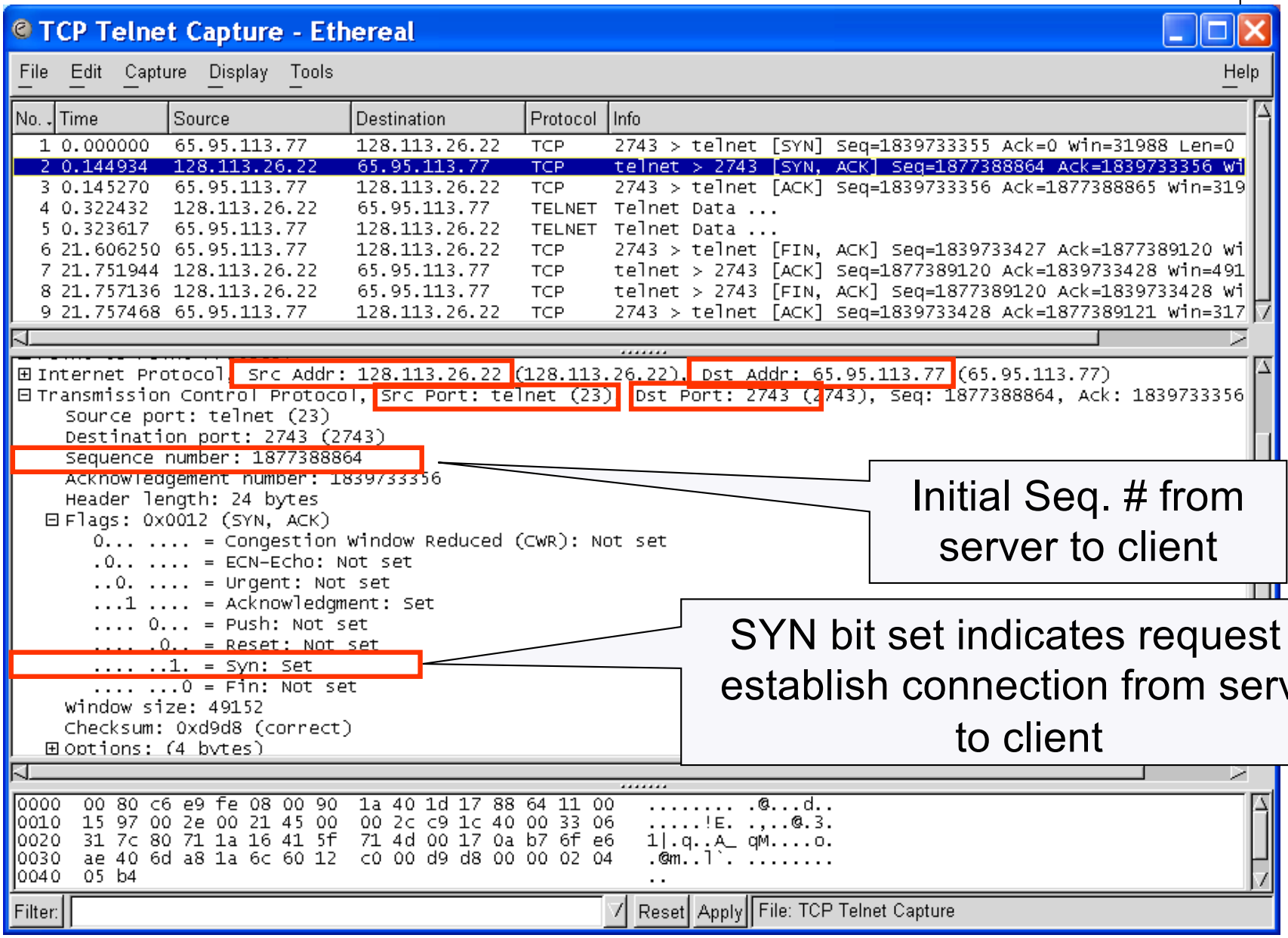
2nd Handshake: ACK from Server



ACK Seq. # = Init. Seq. # + 1

ACK bit set acknowledges connection request; Client-to-Server connection established

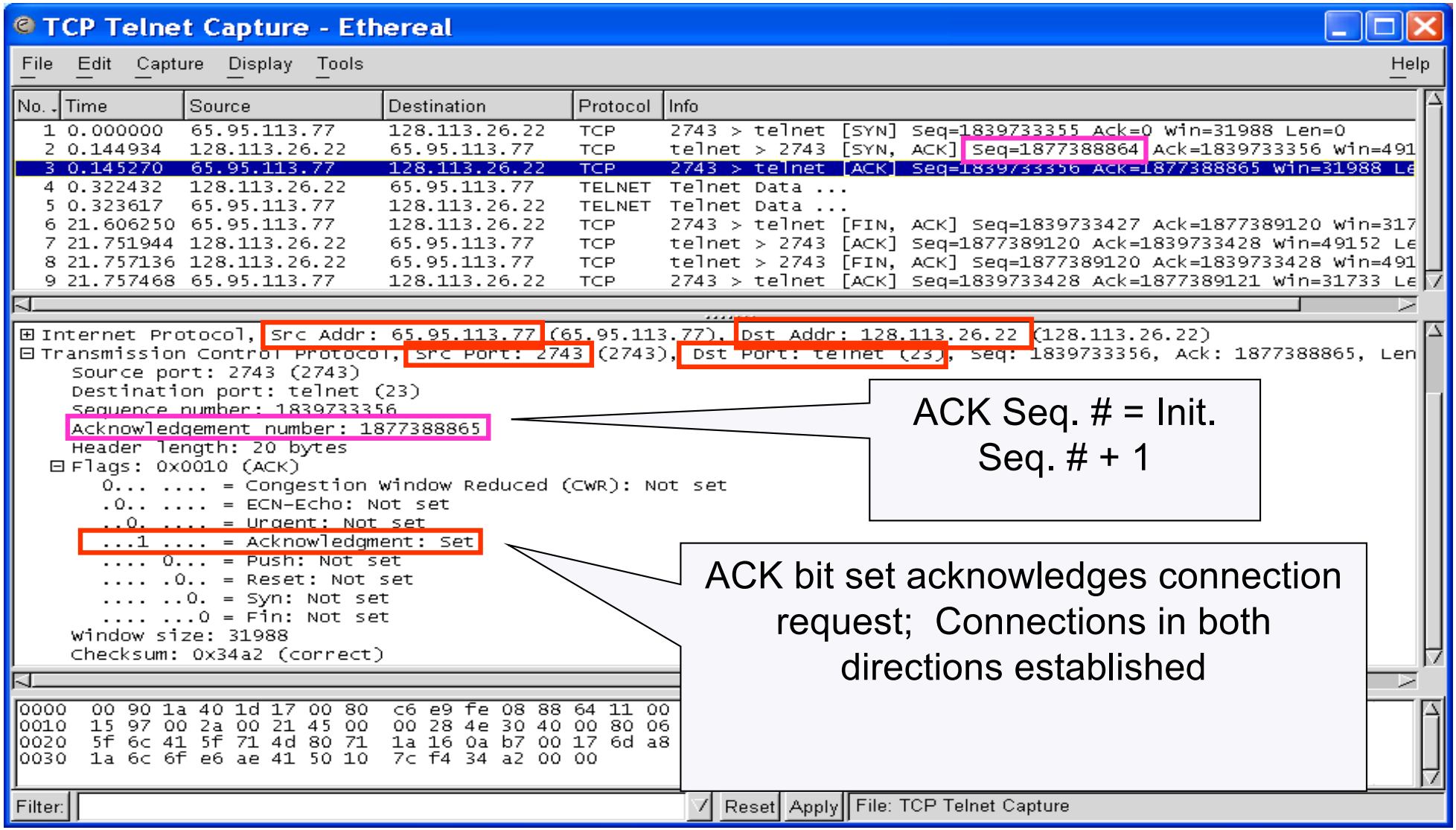
2nd Handshake: Server-Client Connection Request



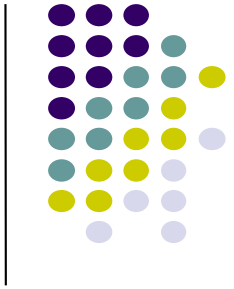
Initial Seq. # from server to client

SYN bit set indicates request to establish connection from server to client

3rd Handshake: ACK from Client

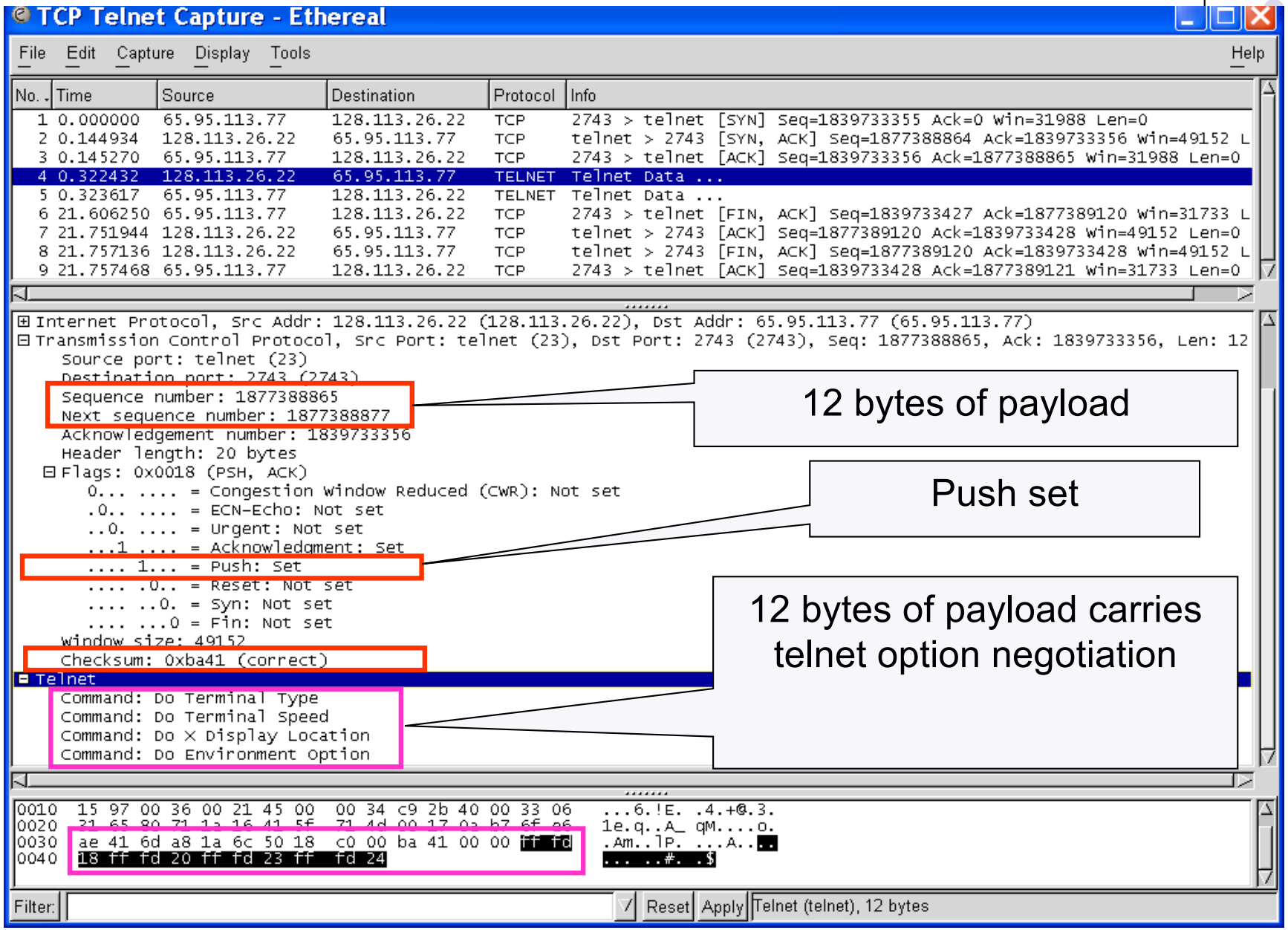


TCP Data Exchange

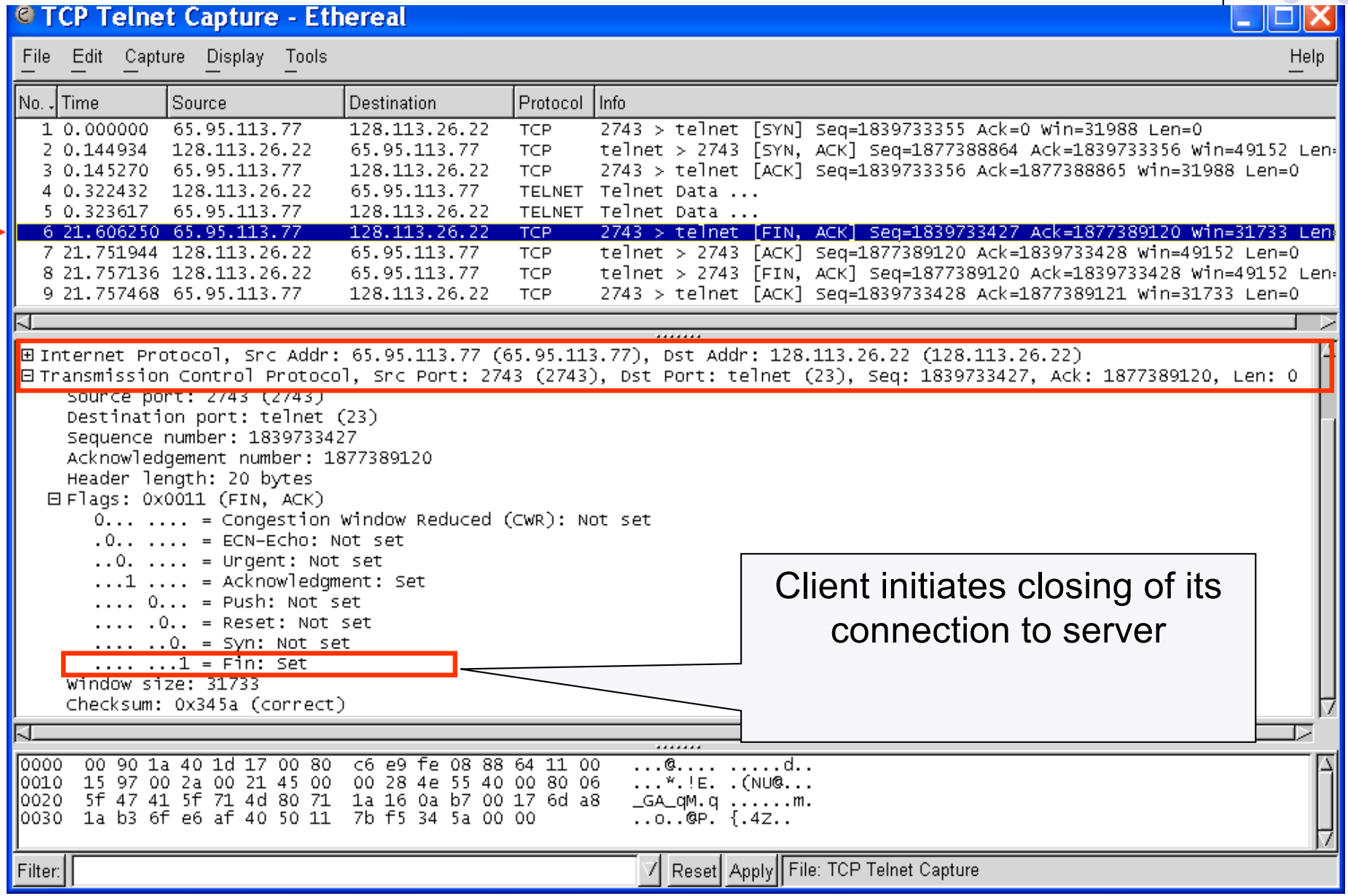


- Application Layers write bytes into buffers
- TCP sender forms segments
  - When bytes exceed threshold or timer expires
  - Upon PUSH command from applications
  - Consecutive bytes from buffer inserted in payload
  - Sequence # & ACK # inserted in header
  - Checksum calculated and included in header
- TCP receiver
  - Performs selective repeat ARQ functions, R_{next}
  - Writes error-free, in-sequence bytes to receive buffer

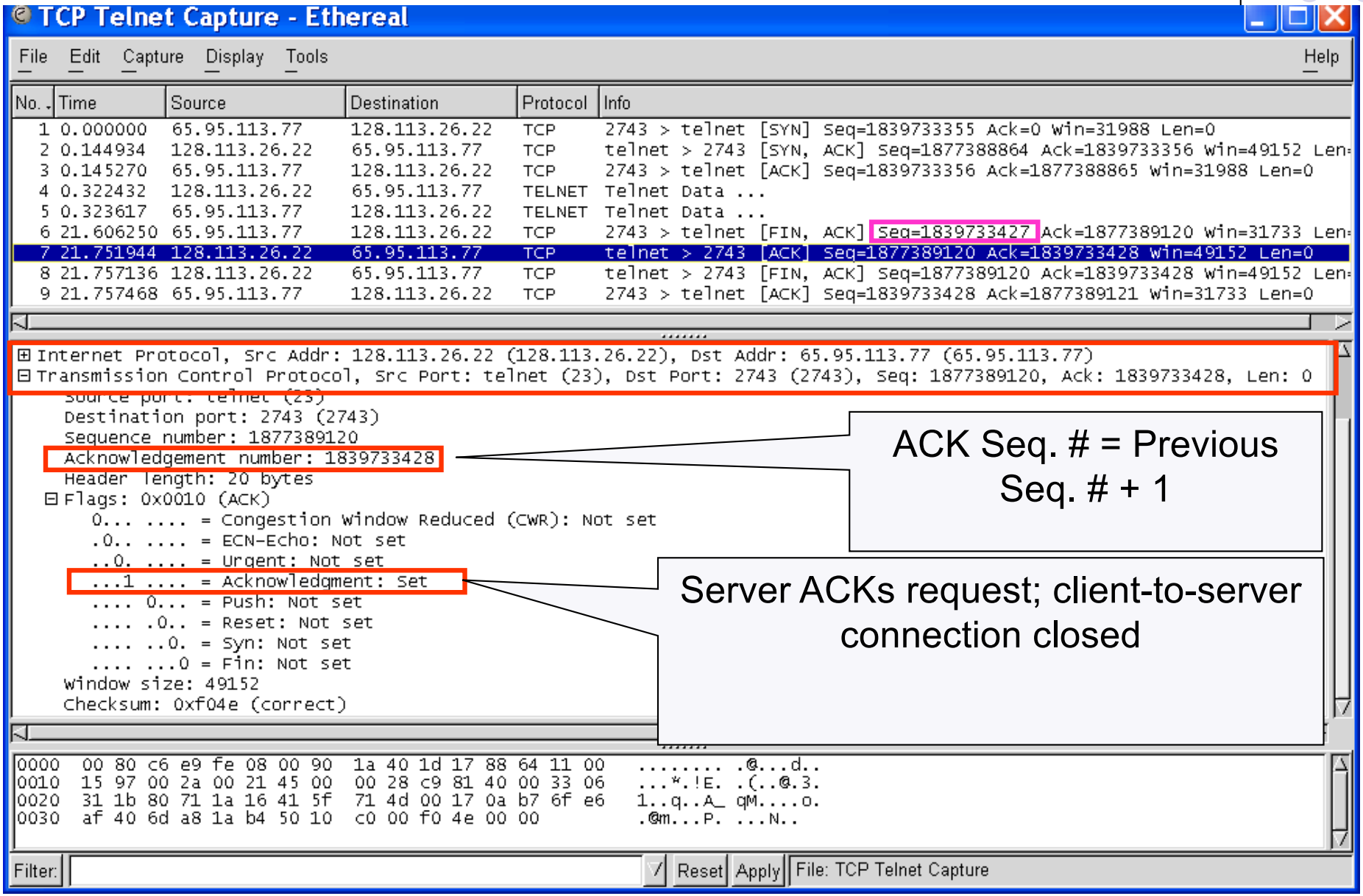
Data Transfer: Server-to-Client Segment



Graceful Close: Client-to-Server Connection

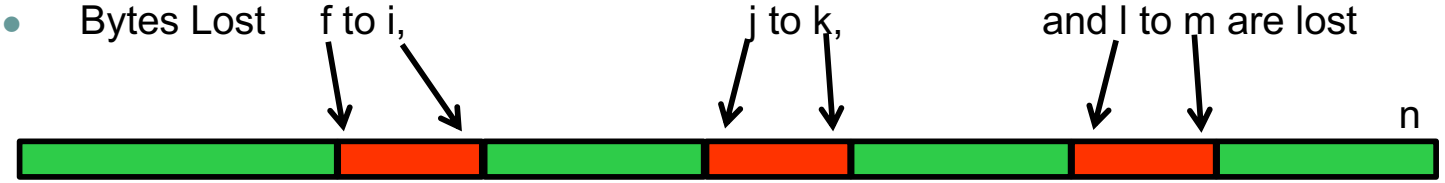


Graceful Close: Client-to-Server Connection



TCP Selective ACK (RFC 2018)

- SACK option allows selective retransmission
- Assume
  - Byte n is highest sequence # byte received correctly
  - Bytes Lost f to i, j to k, and l to m are lost



Receiver sends back:

- CACK=f, cumulative ACK “all bytes up to f-1 received correctly”
- SACK=(i+1 to j-1,k+1 to l-1, m+1 to n)

Sender then knows that:

- blocks f to i, j to k, and l to m are lost and retransmits them
- RFC 2018 uses 2 32-bit words to identify boundaries of up to 4 blocks

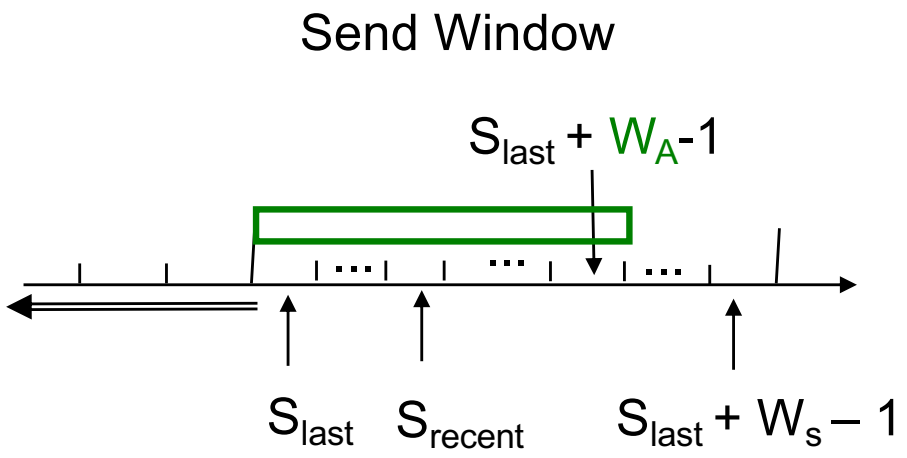
Flow Control

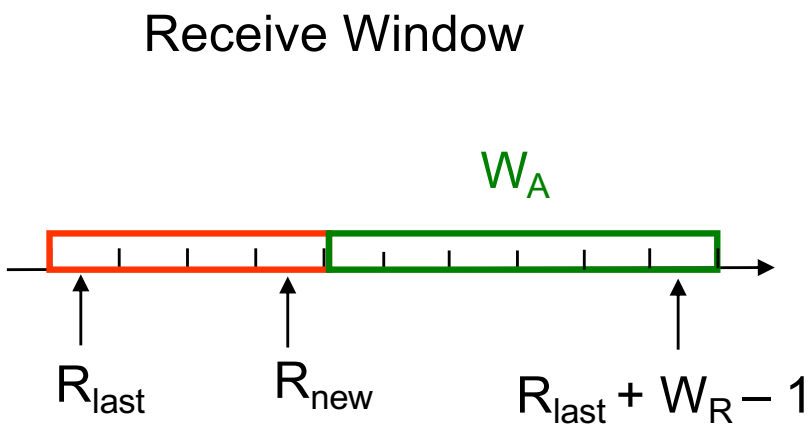
- TCP receiver controls rate at which sender transmits to prevent buffer overflow in the receiver buffer
- TCP receiver advertises a window size specifying number of bytes that can be accommodated by receiver

$$W_A = W_R - (R_{new} - R_{last})$$

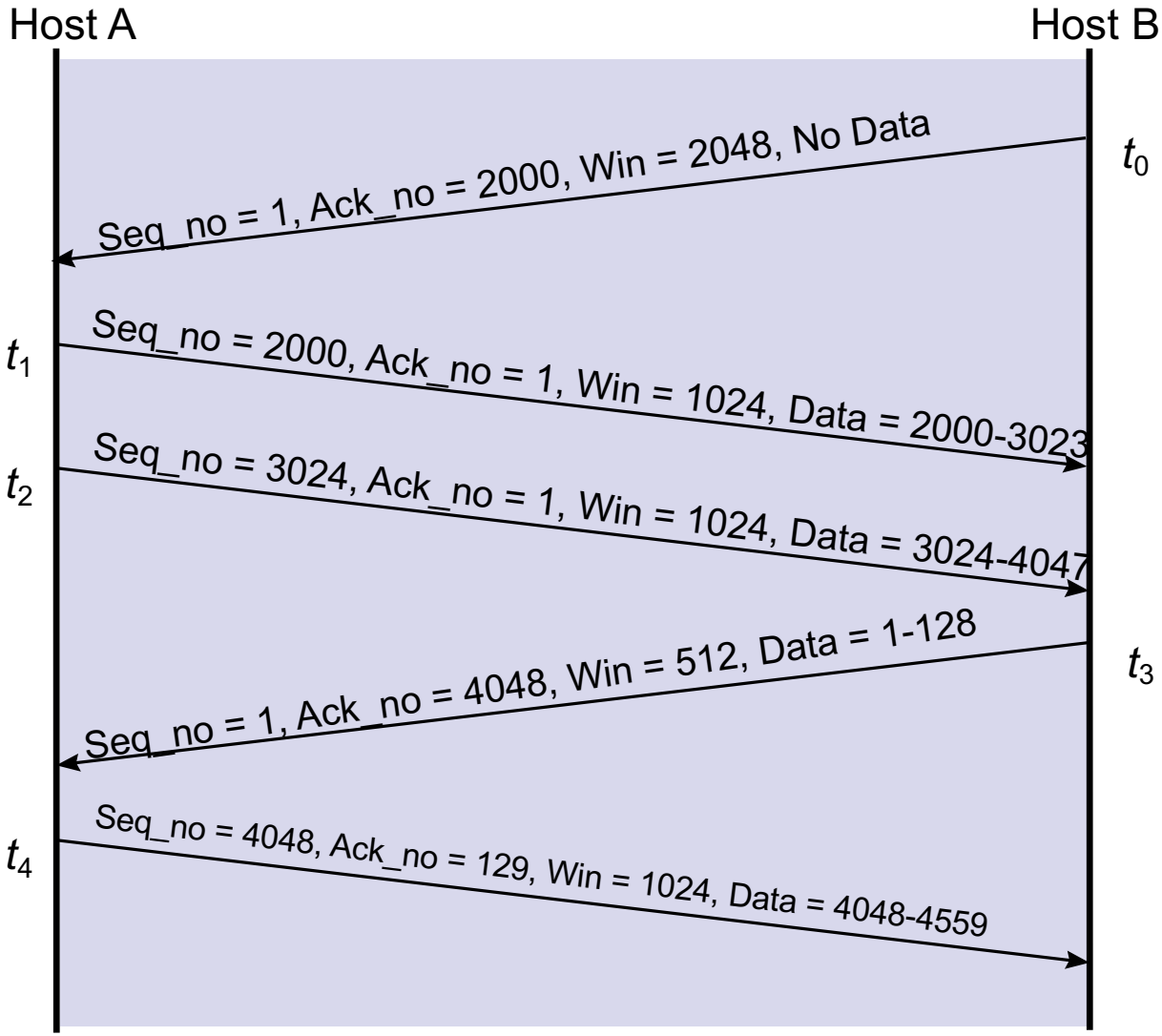
- TCP sender obliged to keep # outstanding bytes below W_A

$$(S_{recent} - S_{last}) \leq W_A$$

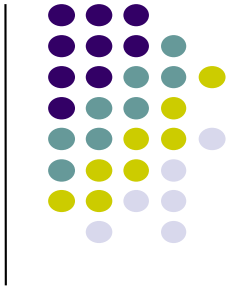




TCP window flow control



TCP Retransmission Timeout

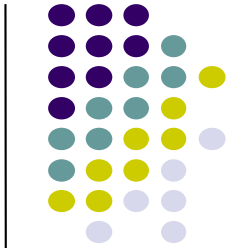


- TCP retransmits a segment after timeout period
  - If timeout too short: excessive number of retransmissions
  - If timeout too long: recovery too slow
  - Timeout depends on RTT: time from when segment is sent to when ACK is received
- Round trip time (RTT) in Internet is highly variable
  - Routes vary and can change in mid-connection
  - Traffic fluctuates
- TCP uses adaptive estimation of RTT
  - Measure RTT each time ACK received: τ_n

$$t_{RTT}(\text{new}) = \alpha\, t_{RTT}(\text{old}) + (1 - \alpha)\, \tau_n$$

  - $\alpha = 7/8$ typical

RTT Variability



- Estimate variance σ^2 of RTT variation
- Estimate for timeout:

$$t_{out} = t_{RTT} + k\ \sigma_{RTT}$$

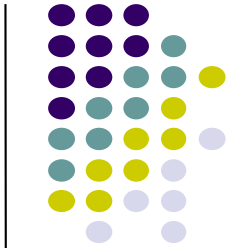
- If RTT highly variable, timeout increase accordingly
- If RTT nearly constant, timeout close to RTT estimate

- Approximate estimation of deviation

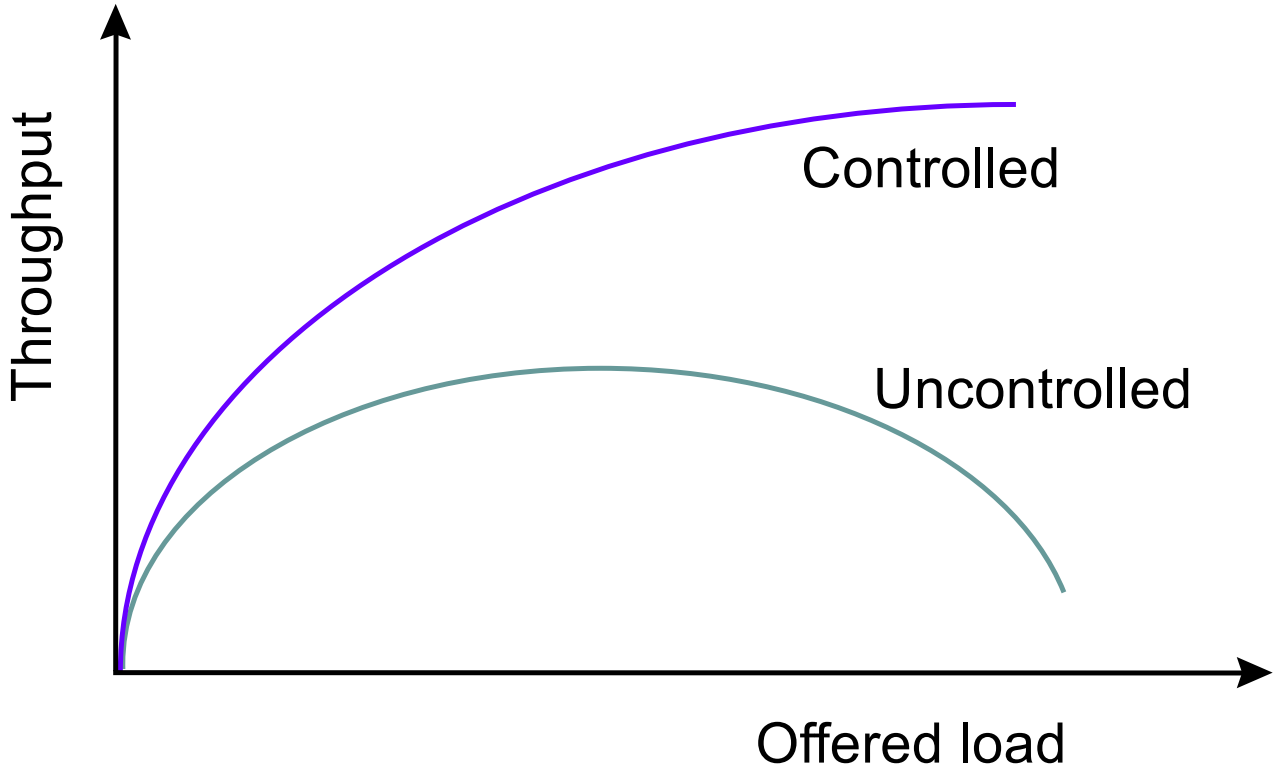
$$d_{RTT}(new) = \beta\, d_{RTT}(old) + (1-\beta)\,|\,\tau_n - t_{RTT}\,|$$

$$t_{out} = t_{RTT} + 4\ d_{RTT}$$

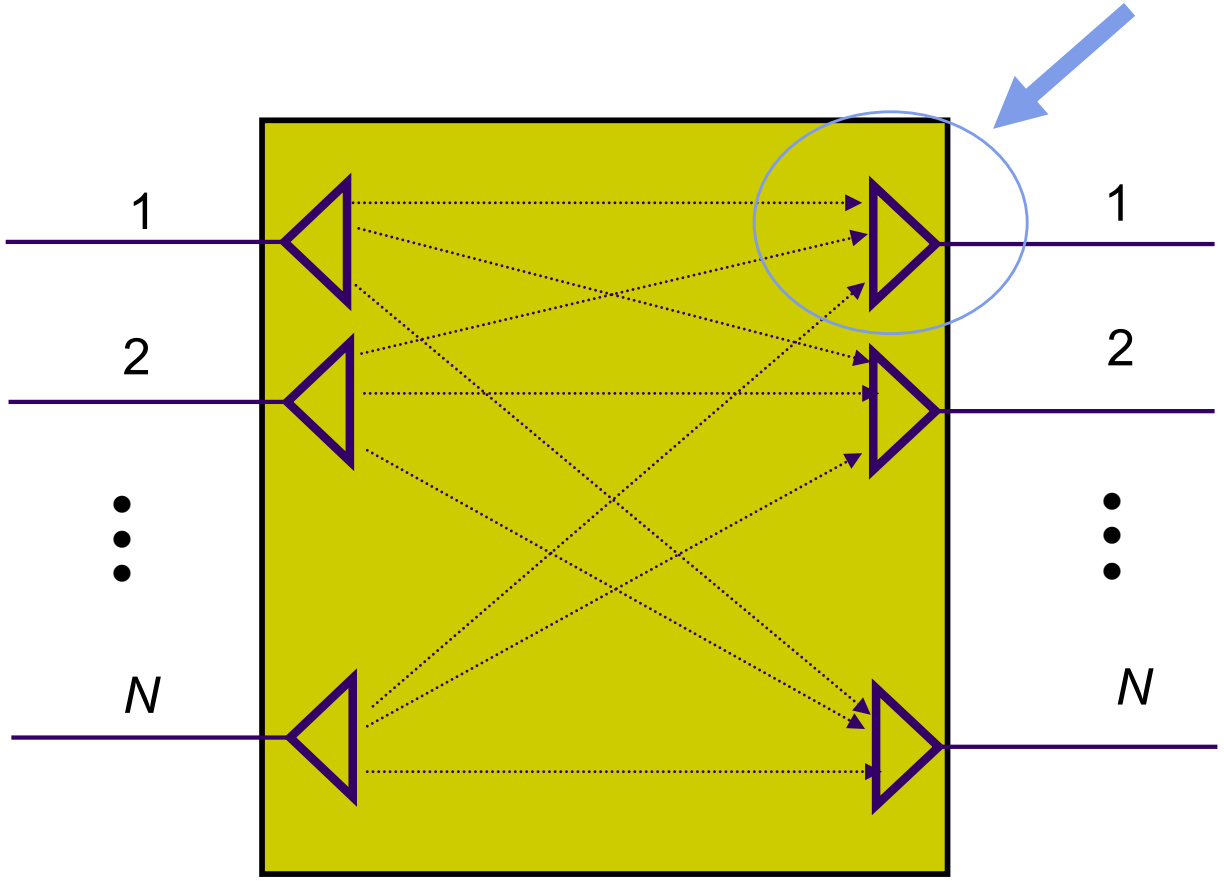
Congestion Control



Ideal effect of congestion control:
Resources used efficiently up to capacity available
Throughput (Bytes/second) increases steadily without drop

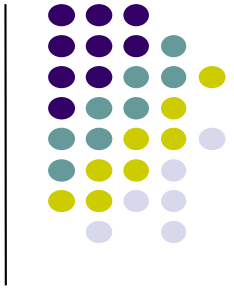


Multiplexers inherent in Packet Switches

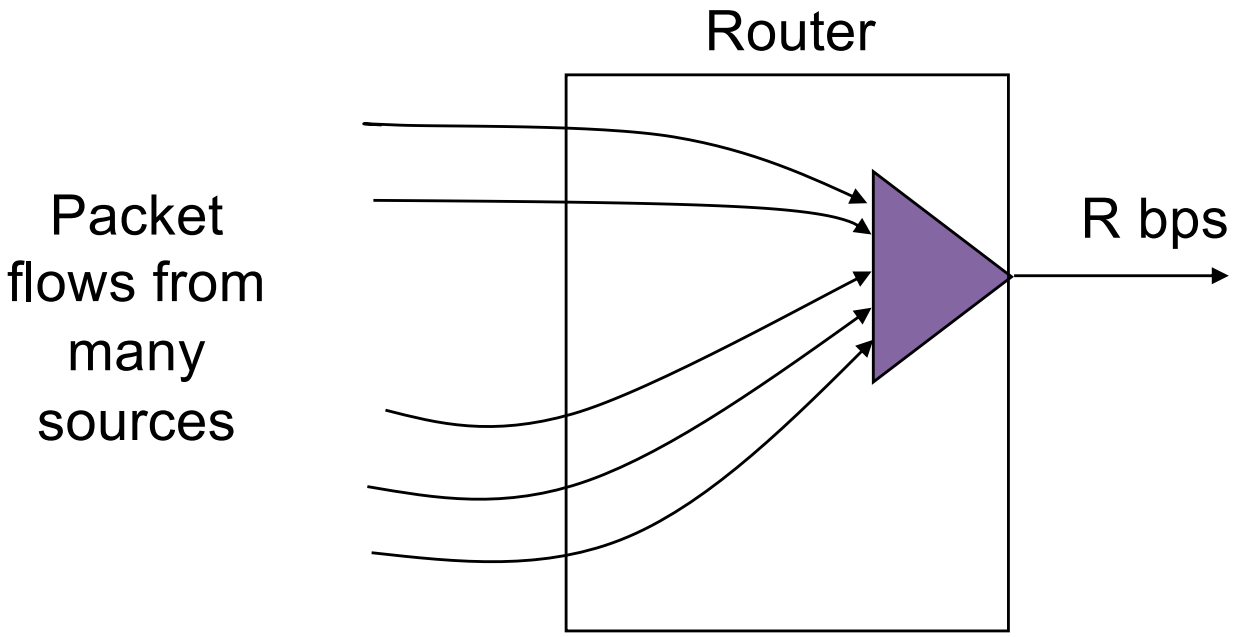


- Packets/frames forwarded to buffer prior to transmission from switch
- Multiplexing occurs in these buffers

TCP Congestion Control

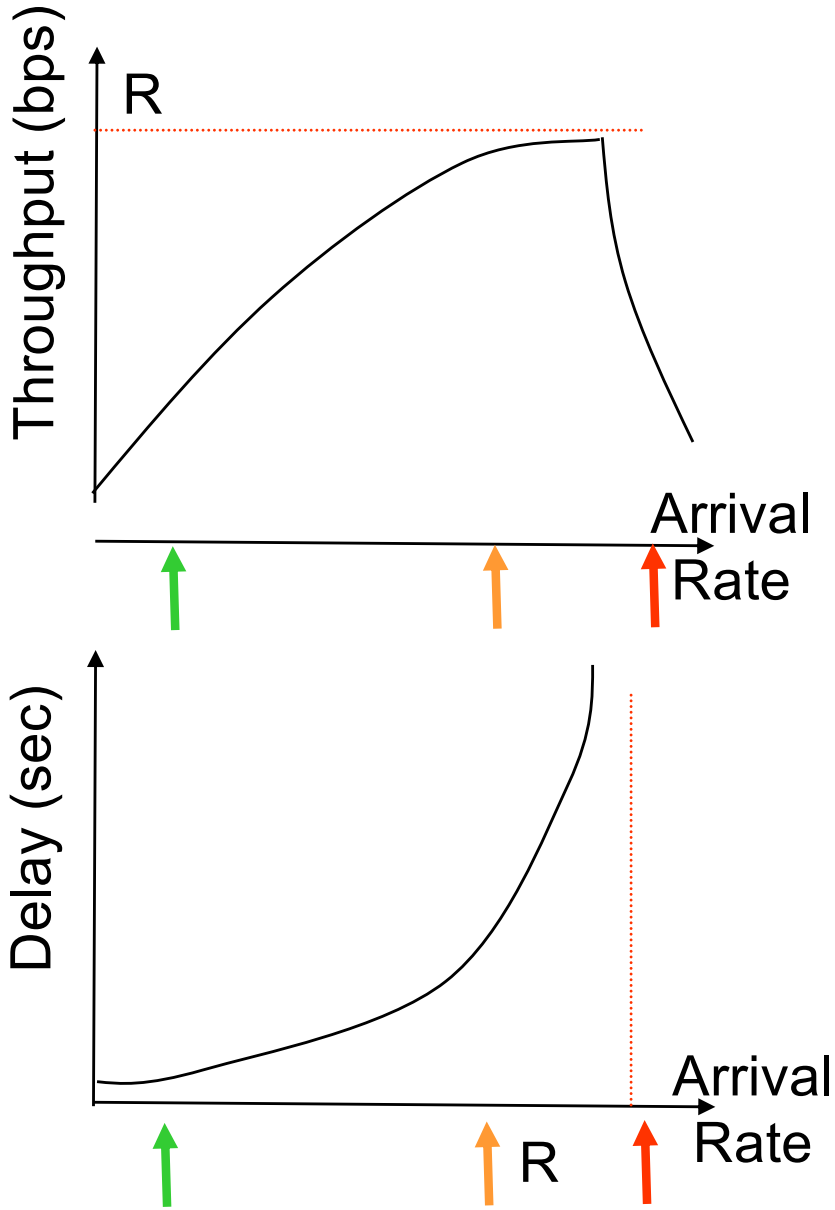


- *Advertised window* size is used to ensure that receiver's buffer will not overflow
- However, buffers at intermediate routers between source and destination may overflow



- Congestion occurs when total arrival rate from all packet flows exceeds R over a sustained period of time
- Buffers at multiplexer will fill and packets will be lost

Phases of Congestion Behavior



Light traffic

- Arrival Rate << R
- Low delay
- Can accommodate more

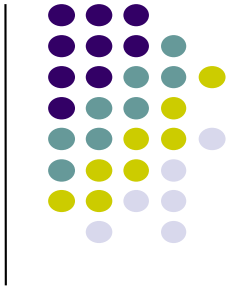
Knee (congestion onset)

- Arrival rate approaches R
- Delay increases rapidly
- Throughput begins to saturate

Congestion Collapse

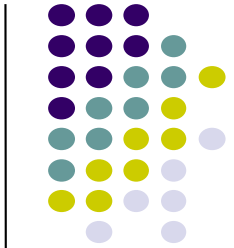
- Arrival rate > R
- Large delays, packet loss
- Useful application throughput drops

Window Congestion Control



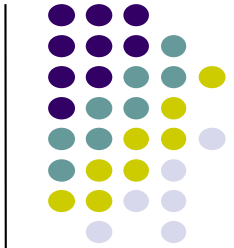
- Desired operating point: just before knee
  - Sources must control their sending rates so that aggregate arrival rate is just before knee
- TCP sender maintains a *congestion window* cwnd to control congestion at intermediate routers
- Effective window is minimum of congestion window and advertised window
- Problem: source does not know what its “fair” share of available bandwidth should be
- Solution: adapt dynamically to available BW
  - Sources probe the network by increasing cwnd
  - When congestion detected, sources reduce rate
  - Ideally, sources sending rate stabilizes near ideal point

Congestion Window

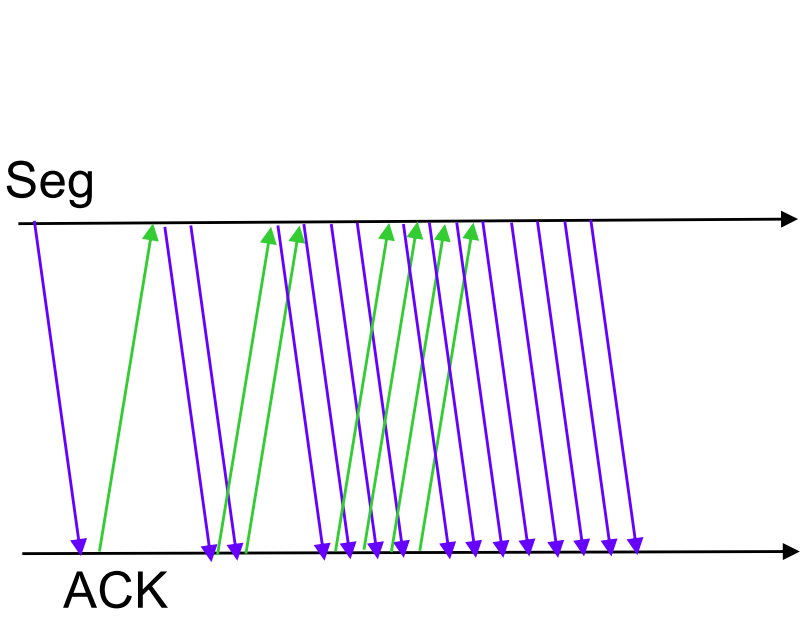


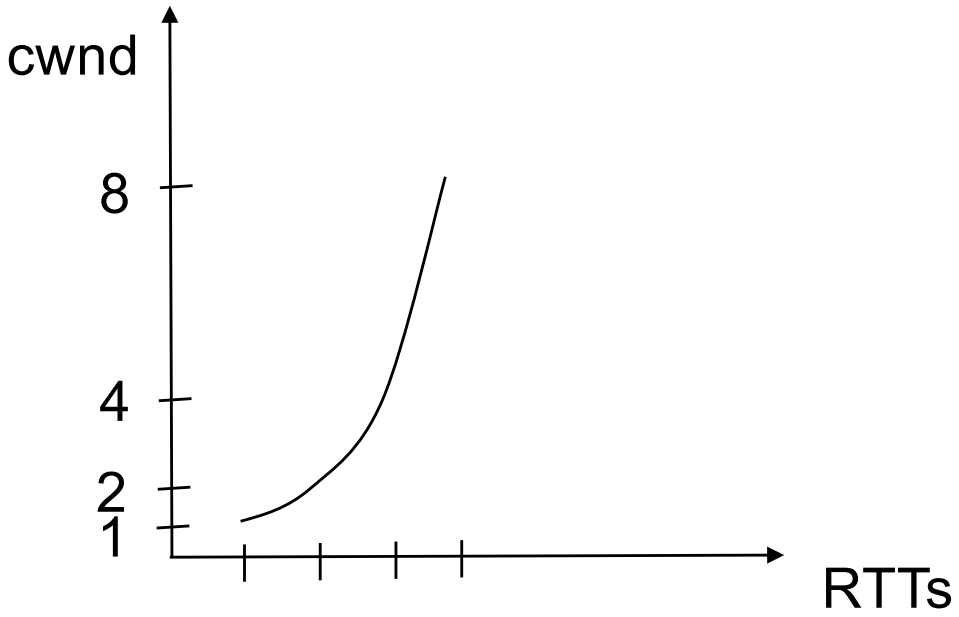
- How does the TCP congestion algorithm change congestion window dynamically according to the most up-to-date state of the network?
- At light traffic: each segment is ACKed quickly
  - Increase cwnd aggresively
- At knee: segment ACKs arrive, but more slowly
  - Slow down increase in cwnd
- At congestion: segments encounter large delays (so retransmission timeouts occur); segments are dropped in router buffers (resulting in duplicate ACKs)
  - Reduce transmission rate, then probe again

TCP Congestion Control: Slow Start

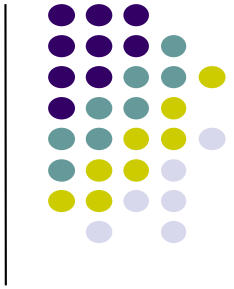


- **Slow start**: increase congestion window size by one segment upon receiving an ACK from receiver
  - initialized at 1 segment
  - used at (re)start of data transfer
  - congestion window increases exponentially

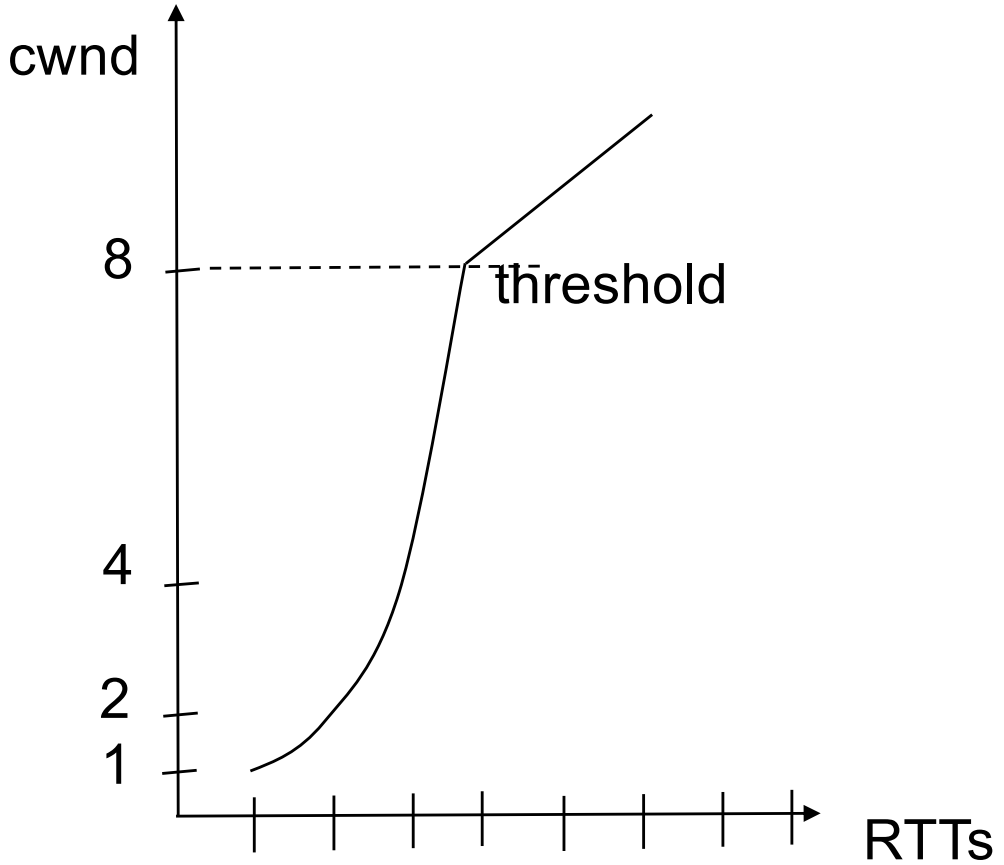




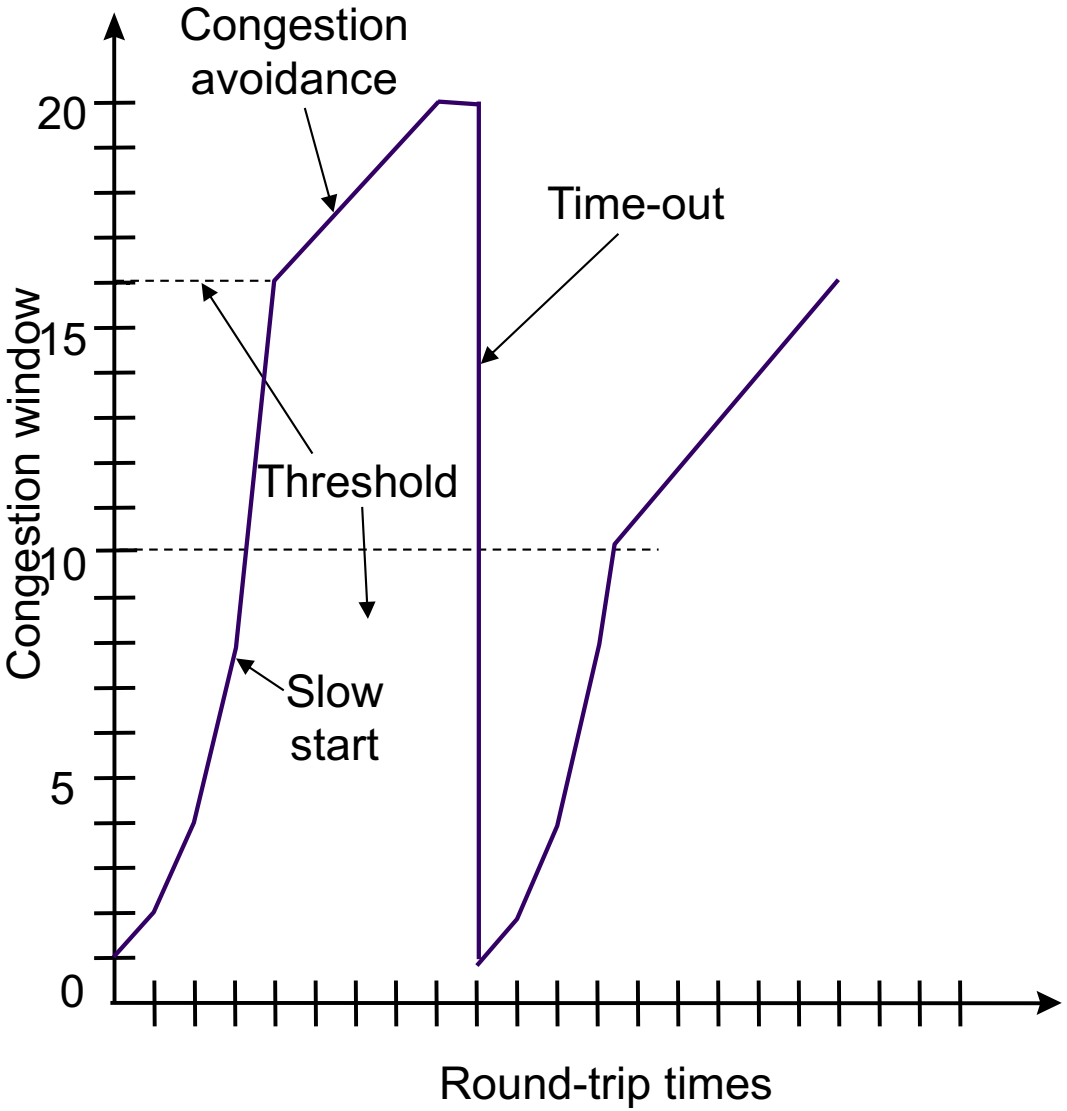
TCP Congestion Control: Congestion Avoidance



- Algorithm progressively sets a *congestion threshold*
  - When cwnd > threshold, slow down rate at which cwnd is increased
- Increase congestion window size by one segment per round-trip-time (RTT)
  - Each time an ACK arrives, cwnd is increased by 1/cwnd
  - In one RTT, cwnd segments are sent, so total increase in cwnd is cwnd x 1/cwnd = 1
  - cwnd grows linearly with time

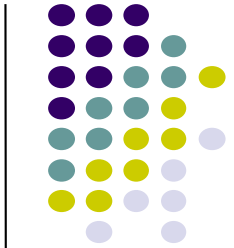


TCP Congestion Control: Congestion

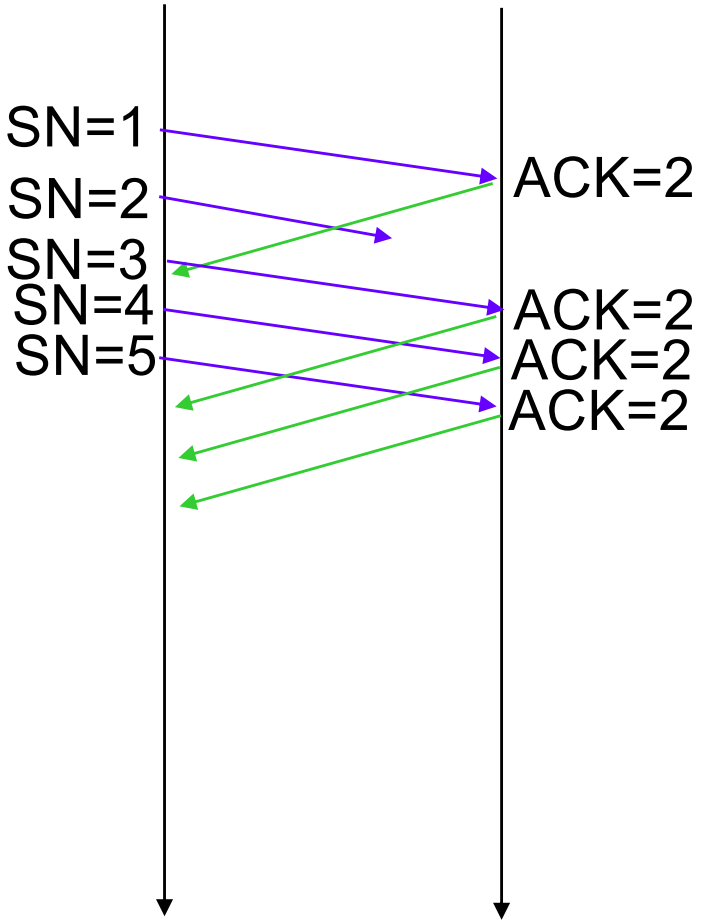


- Congestion is detected upon timeout or receipt of duplicate ACKs
- Assume current cwnd corresponds to available bandwidth
- Adjust congestion threshold = ½ x current cwnd
- Reset cwnd to 1
- Go back to slow-start
- Over several cycles expect to converge to congestion threshold equal to about ½ the available bandwidth

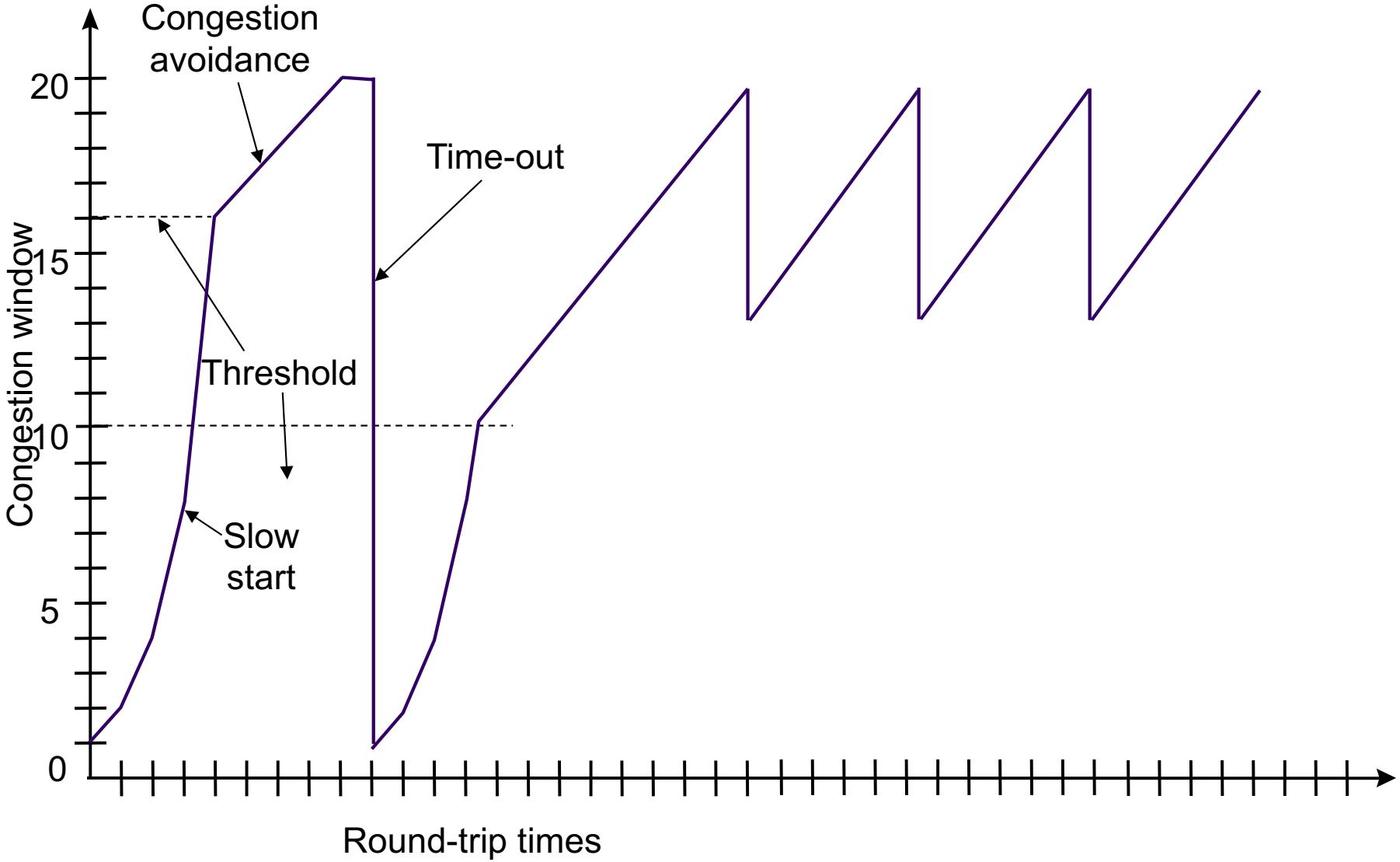
Fast Retransmit & Fast Recovery



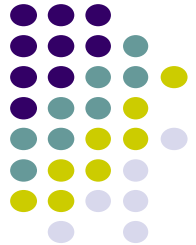
- Congestion causes many segments to be dropped
- If only a single segment is dropped, then subsequent segments trigger duplicate ACKs before timeout
- Can avoid large decrease in cwnd as follows:
  - When three duplicate ACKs arrive, retransmit lost segment immediately
  - Reset congestion threshold to ½ cwnd
  - Reset cwnd to congestion threshold + 3 to account for the three segments that triggered duplicate ACKs
  - Remain in congestion avoidance phase
  - However if timeout expires, reset cwnd to 1
  - In absence of timeouts, cwnd will oscillate around optimal value



TCP Congestion Control: Fast Retransmit & Fast Recovery



Random Early Detection (RED)



- Packets produced by TCP will reduce input rate in response to network congestion
- Early drop: discard packets before buffers are full
- Random drop causes some sources to reduce rate before others, causing gradual reduction in aggregate input rate

Algorithm:

- Maintain running average of queue length
- If Q_{avg} < minthreshold, do nothing
- If Q_{avg} > maxthreshold, drop packet
- If in between, drop packet according to probability
- Flows that send more packets are more likely to have packets dropped

Packet Drop Profile in RED