Computer Networks

Wenzhong Li
Nanjing University
Fall 2014

Chapter 5. End-to-End Protocols

- Transport Services and Mechanisms
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
- TCP Congestion Control
- Real-time Transport Protocol (RTP)
- Session Initiation Protocol (SIP)
- Real Time Streaming Protocol (RTSP)

User Datagram Protocol (UDP)

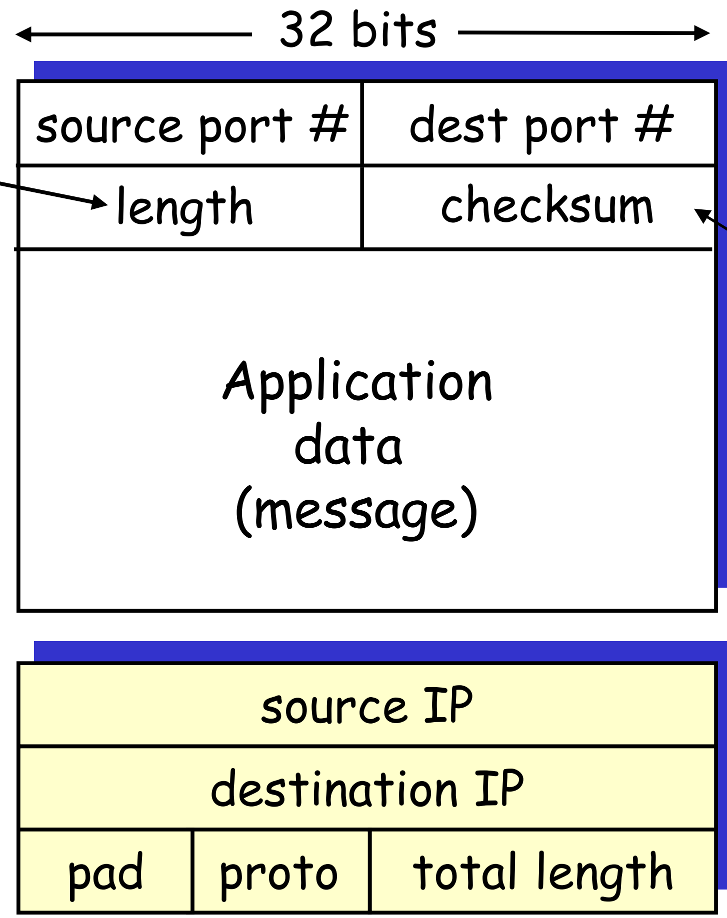
- User datagram protocol, RFC 768
- Connectionless service for application level processes
  - Unreliable, "best-effort" of IP
  - Each UDP segment handled independently of others
  - Delivery and duplication control not guaranteed
- Simple and reduced overhead
  - No connection establishment
  - No connection state at sender, receiver
  - Small segment header

UDP Uses

- Normal use
  - Inward data collection from sensors
  - Outward data dissemination
  - Real time applications
  - Request-Response (e.g. RPC), add reliability at application layer
- Example Apps based on UDP
  - DNS
  - SNMP

UDP Segment Format

Length in octets, including Header and Data



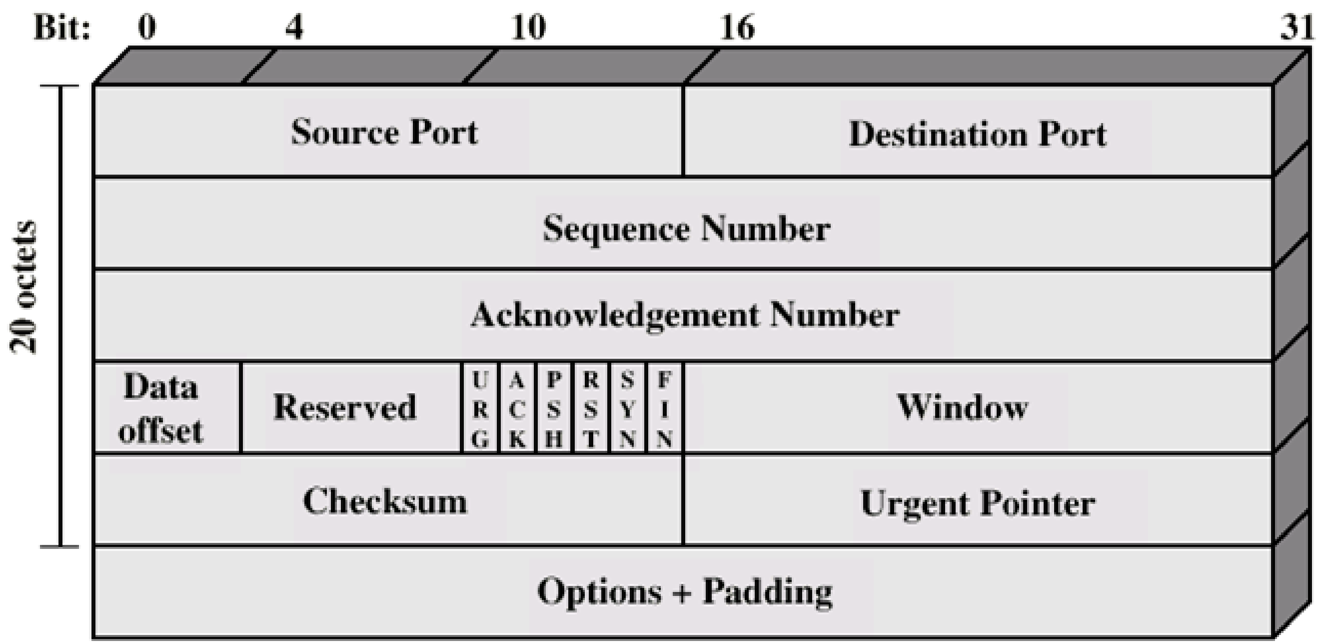
Header + Data + Pseudo-header;
Or set to 0 if no check

Pseudo-Header for Checksum

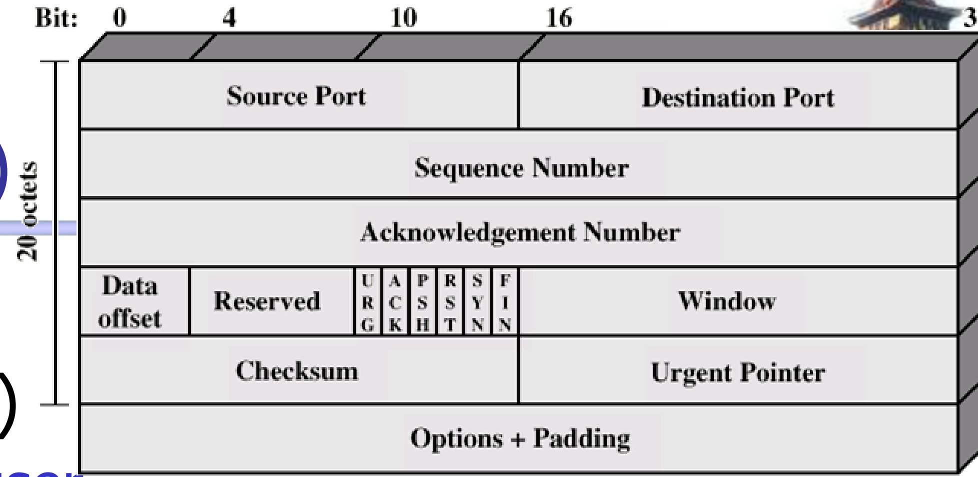
The same as UDP length

Transmission Control Protocol (TCP)

- Reliable communication between pairs of processes
  - Across variety of reliable and unreliable networks and internets
  - 20 bytes

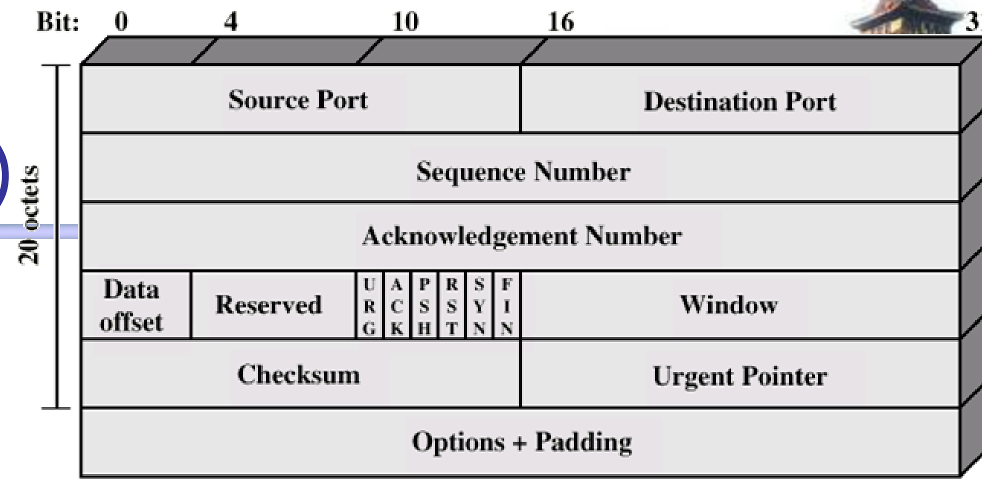


TCP Header Fields (1)



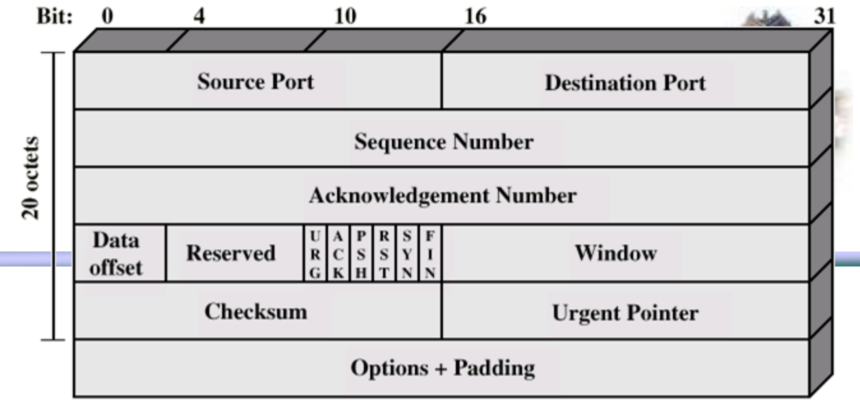
- Source port (16 bits)
- Destination port (16 bits)
  - Identify src and dest TCP user
- Sequence number (32 bits)
  - Seq number of first data octet
  - If SYN is set, it is ISN and first data octet is ISN+1
- ACK number (32 bits)
  - Piggybacked *ACK*
- Window (16 bits)
  - Credit allocation in octets, i.e. rcv_window of sender

TCP Header Fields (2)



- Data offset (4 bits)
  - Number of 32-bit words in the header （报头长度）
  - Largest data offset is 15×4=60 octets
- Checksum (16 bits)
  - Header + Data + Pseudo-header (src IP, dest IP, protocol No, total length)
- Reserved (6 bits)
- Options (Variable)
  - e.g. Maximum segment size the sender can accept
  - or Max value of rcv_window

TCP Header Fields (3)



- Flags (6 bits):
  - URG: urgent pointer field meaningful
  - ACK: acknowledgment field meaningful
  - PSH: push function
  - RST: reset the connection
  - SYN: synchronize the sequence number
  - FIN: no more data from sender
- Urgent Pointer (16 bits)
  - Points to last octet in a sequence of urgent data

Parameters Passed to IP

- TCP passes QOS parameters down to IP
  - Precedence
  - Normal delay / low delay
  - Normal throughput / high throughput
  - Normal reliability / high reliability
- IPv4 “Type of Service” or IPv6 “Traffic Class”

TCP Service Request Primitives

Primitive	Parameters	Description
Unspecified Passive Open	source-port, [timeout], [timeout-action], [precedence], [security-range]	Listen for connection attempt at specified security and precedence from any remote destination.
Fully Specified Passive Open	source-port, destination-port, destination-address, [timeout], [timeout-action], [precedence], [security-range]	Listen for connection attempt at specified security and precedence from specified destination.
Active Open	source-port, destination-port, destination-address, [timeout], [timeout-action], [precedence], [security]	Request connection at a particular security and precedence to a specified destination.
Active Open with Data	source-port, destination-port, destination-address, [timeout], [timeout-action], [precedence], [security], data, data-length, PUSH-flag, URGENT-flag	Request connection at a particular security and precedence to a specified destination and transmit data with the request
Send	local-connection-name, data, data-length, PUSH-flag, URGENT-flag, [timeout], [timeout-action]	Transfer data across named connection
Allocate	local-connection-name, data-length	Issue incremental allocation for receive data to TCP
Close	local-connection-name	Close connection gracefully
Abort	local-connection-name	Close connection abruptly
Status	local-connection-name	Query connection status

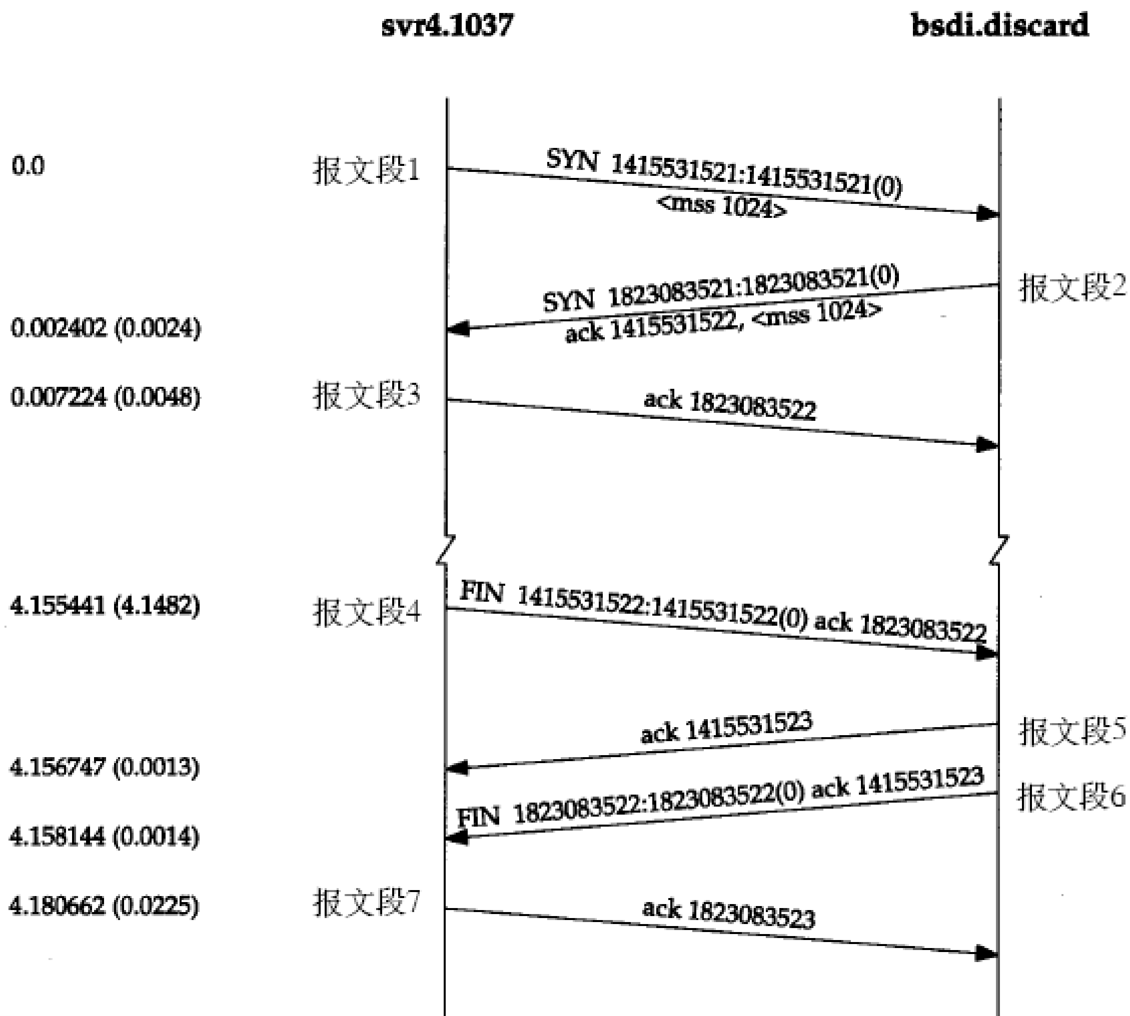
TCP Service Response Primitives

Primitive	Parameters	Description
Open ID	local-connection-name, source-port, destination-port*, destination-address*,	Informs TCP user of connection name assigned to pending connection requested in an Open primitive
Open Failure	local-connection-name	Reports failure of an Active Open request
Open Success	local-connection-name	Reports completion of pending Open request
Deliver	local-connection-name, data, data-length, URGENT-flag	Reports arrival of data
Closing	local-connection-name	Reports that remote TCP user has issued a Close and that all data sent by remote user has been delivered
Terminate	local-connection-name, description	Reports that the connection has been terminated; a description of the reason for termination is provided
Status Response	local-connection-name, source-port, source-address, destination-port, destination-address, connection-state, receive-window, send-window, amount-awaiting-ACK, amount-awaiting-receipt, urgent-state, precedence, security, timeout	Reports current status of connection
Error	local-connection-name, description	Reports service-request or internal error

TCP Mechanisms (1)

- Connection establishment
  - 3-way handshake
  - Between pairs of ports
  - One port can connect to multiple destination ports
- Connection termination
  - Graceful termination: CLOSE + FIN
  - Abrupt termination: ABORT + RST

Illustration of Connection establishment & Termination



TCP Mechanisms (2)

- Data transfer
  - Logical stream of octets
  - Octets numbered modulo 2^{32}
  - Flow control by credit allocation of number of octets
  - Data buffered at sender and receiver
  - User sets *PUSH* to force data transmission immediately
  - User may specify a block of data as urgent

Implementation Policy

- Send
- Deliver
- Accept
- Retransmit
- Acknowledge

Send

- If no *PUSH* or *CLOSE*, *TCP* entity transmits at its own convenience

- Data issued by *TCP* user buffered at transmit buffer
  - May construct segment per data batch
  - May wait for certain amount of data

Deliver

- In absence of *PUSH*, *TCP* entity delivers data at own convenience

- May deliver as each segment in order received
  - Deliveries (*I/O* interrupts) are frequent and small

- May buffer data from more than one segment
  - Deliveries are infrequent and large

Accept

- Segments may arrive out of order
- In order
  - Only accept segments in order
  - Discard out of order segments
  - Makes for a simpler implementation
- In windows
  - Accept all segments within receive window
  - Can reduce retransmission

Retransmit

- TCP entity maintains queue of segments transmitted but not acknowledged

- TCP will retransmit if not ACKed in given time
  - First only: one timer a queue, reset the timer after retransmission of first segment in queue
  - Batch: one timer a queue, reset after retransmission of all segments in queue
  - Individual: one timer each segment, reset after retransmission

Fast Retransmit

- Time-out period often relatively long
- Detect lost segments via duplicate *ACKs*
  - If segment is lost, there will likely be many duplicate *ACKs*
- If a *TCP* entity receives 3 *ACKs* for the same data, then segments after ACKed data must be lost
  - Trigger fast retransmit: resend segment before timer expires

Acknowledgement

- Immediate or Cumulative

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment sat at lower end of gap

TCP congestion control

TCP Congestion Control

- Congestion control
  - Too many sources sending too much data too fast for Internet to handle
  - RFC 1122, Requirements for Internet hosts
- End to end control, no network assistance
  - Retransmission timer
  - Window management

Retransmission Timer Management

- Estimate round trip delay by observing delay pattern
  - Simple average
  - Exponential average
- Set timer to value somewhat greater than estimate
  - RFC 793
  - RTT Variance Estimation (Jacobson's algorithm)
- How to set timer after retransmission
  - Exponential RTO backoff algorithm
- When to sample the round trip delay
  - Karn's Algorithm

Simple Average

- Term
  - RTT(i): round-trip time observed for the i^{th} transmitted segment
  - ARTT(k): average round-trip time for the first k segments
- Expression

$$ARTT(k+1) = \frac{1}{k+1}\sum_{i=1}^{k+1} RTT(i) \quad \text{or}$$

$$ARTT(k+1) = \frac{1}{k+1}\left(k \times ARTT(k) + RTT(i)\right)$$

$$\frac{k}{k+1} ARTT(k) + \frac{1}{k+1} RTT(i)$$

Exponential Average

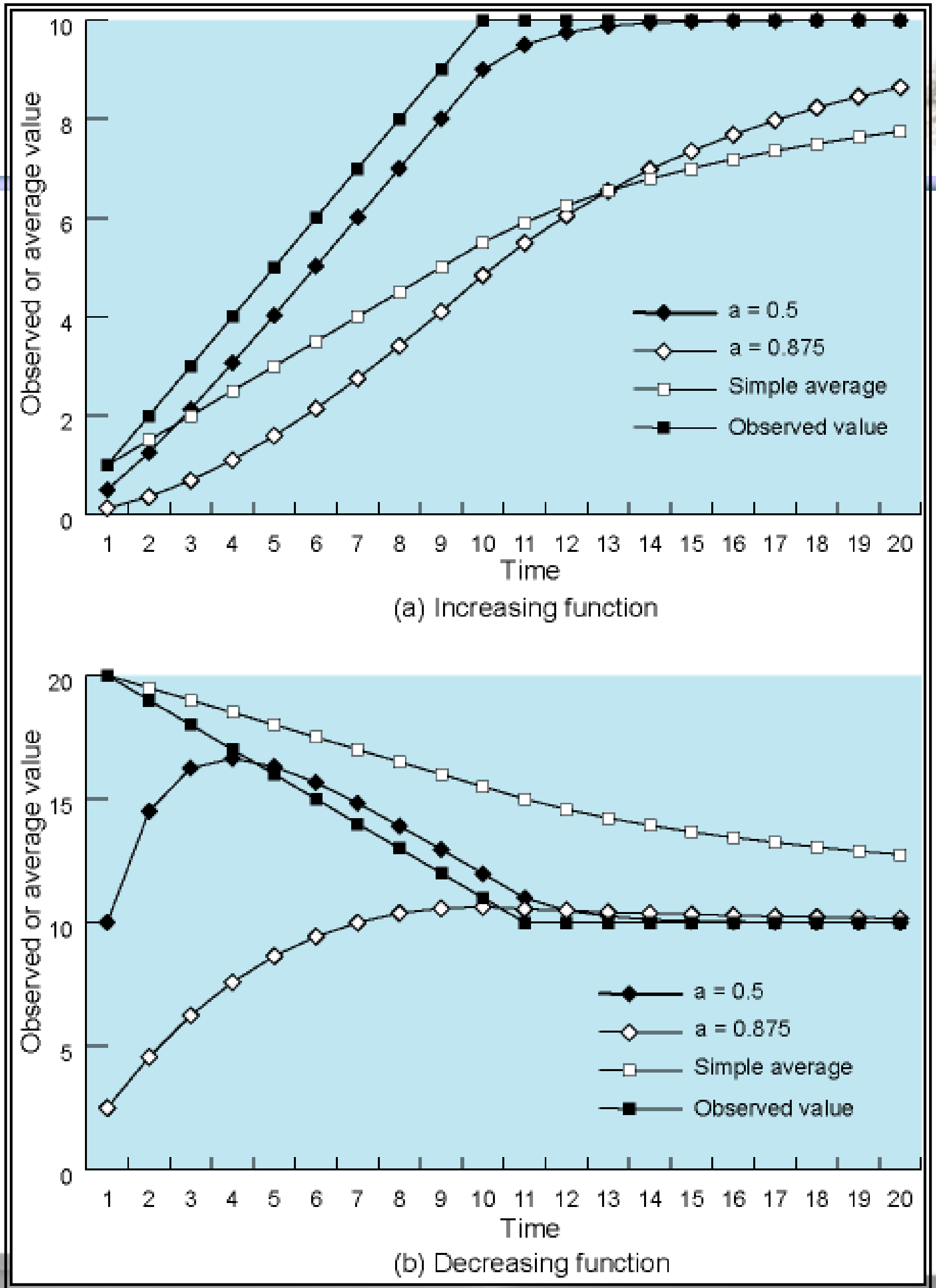
- Term
  - SRTT(k): smoothed round-trip time estimate for the first k segments

- Expression

$$SRTT(k+1) = \alpha \times SRTT(k) + (1-\alpha) \times RTT(k+1) \quad \text{i.e.}$$

$$SRTT(k+1) = (1-\alpha) \times RTT(k+1) + \alpha(1-\alpha) \times RTT(k) + \alpha^2(1-\alpha) \times RTT(k-1) + \ldots + \alpha^k(1-\alpha) \times RTT(1)$$

Simple and Exponential Averaging



RFC 793

- Term
  - RTO(k): retransmission timeout, i.e. the timer after the first k segments

- Expression

$$RTO(k+1) = Min\left(UBOUND, MAX\left(LBOUND, \beta \times SRTT(k+1)\right)\right)$$

  - Retransmission timer set between LBOUND~UBOUND
  - Suggested values, α: 0.8~0.9, β: 1.3~2.0

Jacobson's Algorithm (1)

Problem in RFC 793

- Not counting variance of RTT (network stability)

- When network is stable, RTT variance is low, but β=1.3 gives a higher RTO

- When network is unstable, RTT variance is high, β=2 is inadequate to protect against retransmissions

Jacobson's Algorithm (2)

- Term
  - SERR(k): smoothed error estimate, difference of round-trip time of segment k and the current SRTT
  - SDEV(k): standard deviation for round-trip time of first k segments

- Expression

$$SRTT(k+1) = (1-g) \times SRTT(k) + g \times RTT(k+1)$$

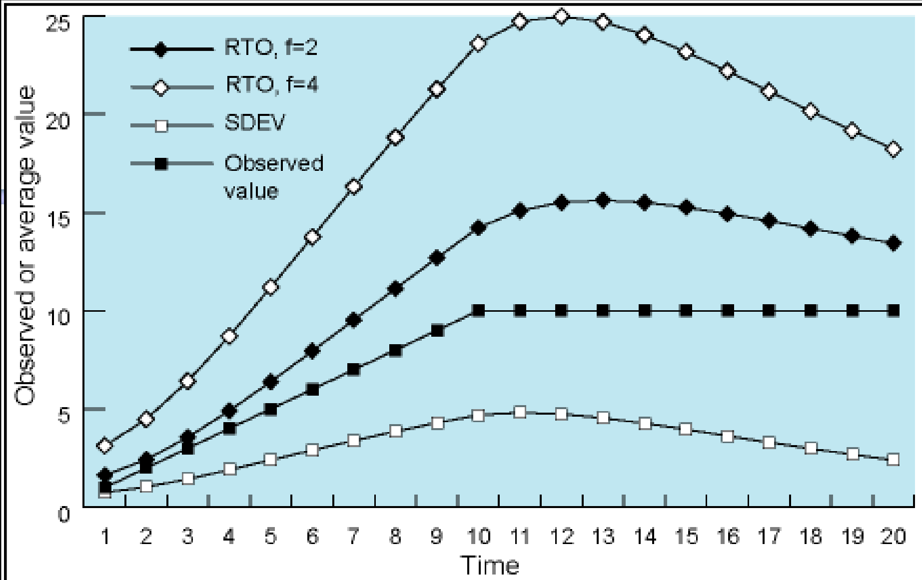
$$SERR(k+1) = RTT(k+1) - SRTT(k)$$

$$SDEV(k+1) = (1-h) \times SDEV(k) + h \times \left|SERR(k+1)\right|$$

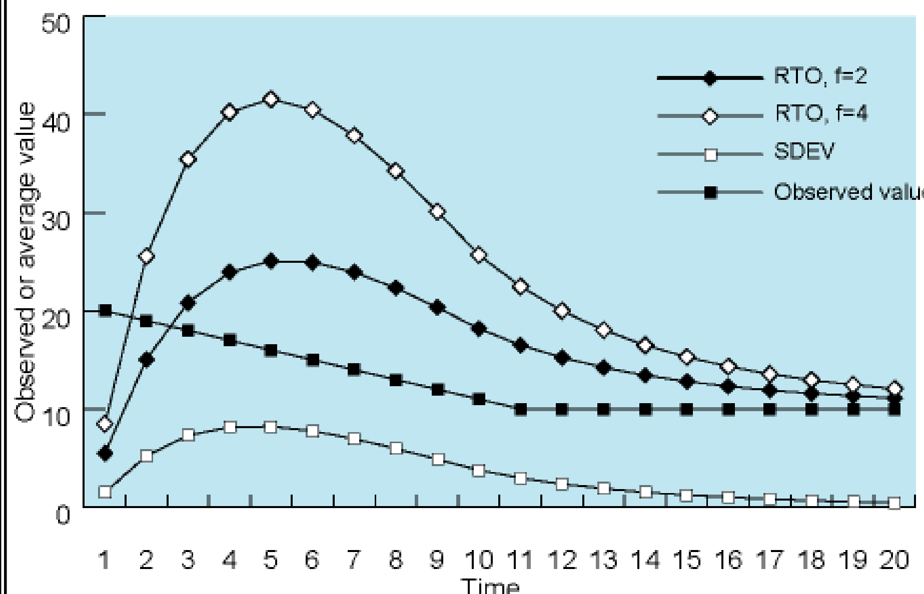
$$RTO(k+1) = SRTT(k+1) + f \times SDEV(k+1)$$

$$g = {}^{1}\!/_{8} = 0.125 \quad h = {}^{1}\!/_{4} = 0.25 \quad f = 2 \text{ or } 4$$

Jacobson's RTO Calculation



(a) Increasing function



(b) Decreasing function

Exponential RTO Backoff

- Timeout is often due to congestion by dropped packet or long round trip
  - Should slow down end system transmission
  - Maintaining RTO is not a good idea
- Similar to Binary exponential backoff in Ethernet
  - RTO multiplied each time a segment is re-transmitted
  - $RTO = q \times RTO$
  - Commonly $q=2$

Karn's Algorithm

- The problem
  - If a segment is re-transmitted, the *ACK* arriving may be
    - For the first copy of the segment, or for the second copy, or others
    - No way to tell

- RTT Sampling
  - Do not measure RTT for re-transmitted segments
  - Calculate RTO backoff when re-transmission occurs
  - Until *ACK* arrives for segment that has not been re-transmitted
  - Begin sampling, stop RTO backoff

Window Management

- Add congestion (send) window besides the credit (receive window)

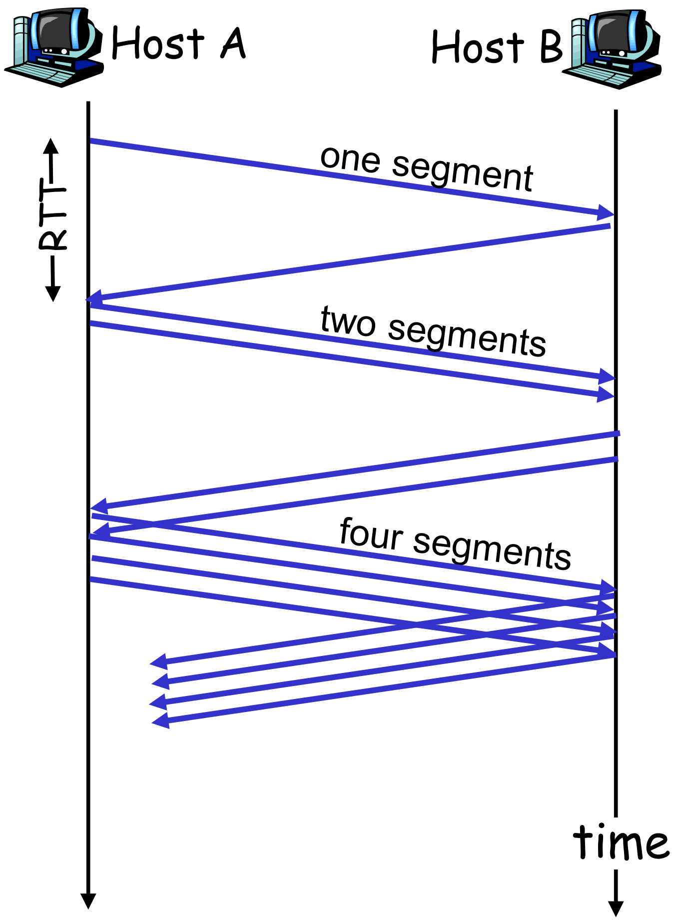
  $$awind = Min(credit, cwnd)$$

  - awnd: allowed window, in *MSS* (maximum segment size)
  - credit: the amount of unused credit granted in last *ACK*, in *MSS*
  - cwnd: congestion window, in *MSS*

- Manage congestion window
  - Slow Start: exponentially expending the cwnd at start of connection
  - Dynamic Window on Congestion: shrinking / expending the cwnd with stages when retransmission occurs

Slow Start

- When connection begins, cwnd = 1 MSS
- Each time an *ACK* received, cwnd increased by ACKed number of *MSSs* until Max value reached
- cwnd increased exponentially until first loss event occurs
  - Timeout or 3 duplicate *ACKs*



Dynamic Windows Sizing

- By Jacobson, set slow start threshold *ssthresh* = cwnd/2

- After 3 duplicate *ACKs*
  - Network still capable of delivering some segments
  - cwnd is set to be *ssthresh*
  - cwnd increases by 1 (linearly instead of exponentially) after each RTT or *ACK* received

- After timeout event
  - cwnd is set to 1
  - Same as Slow Start until *cwnd* reaches *ssthresh*
  - cwnd increases by 1 after each RTT or *ACK* received

Illustration of Window Management

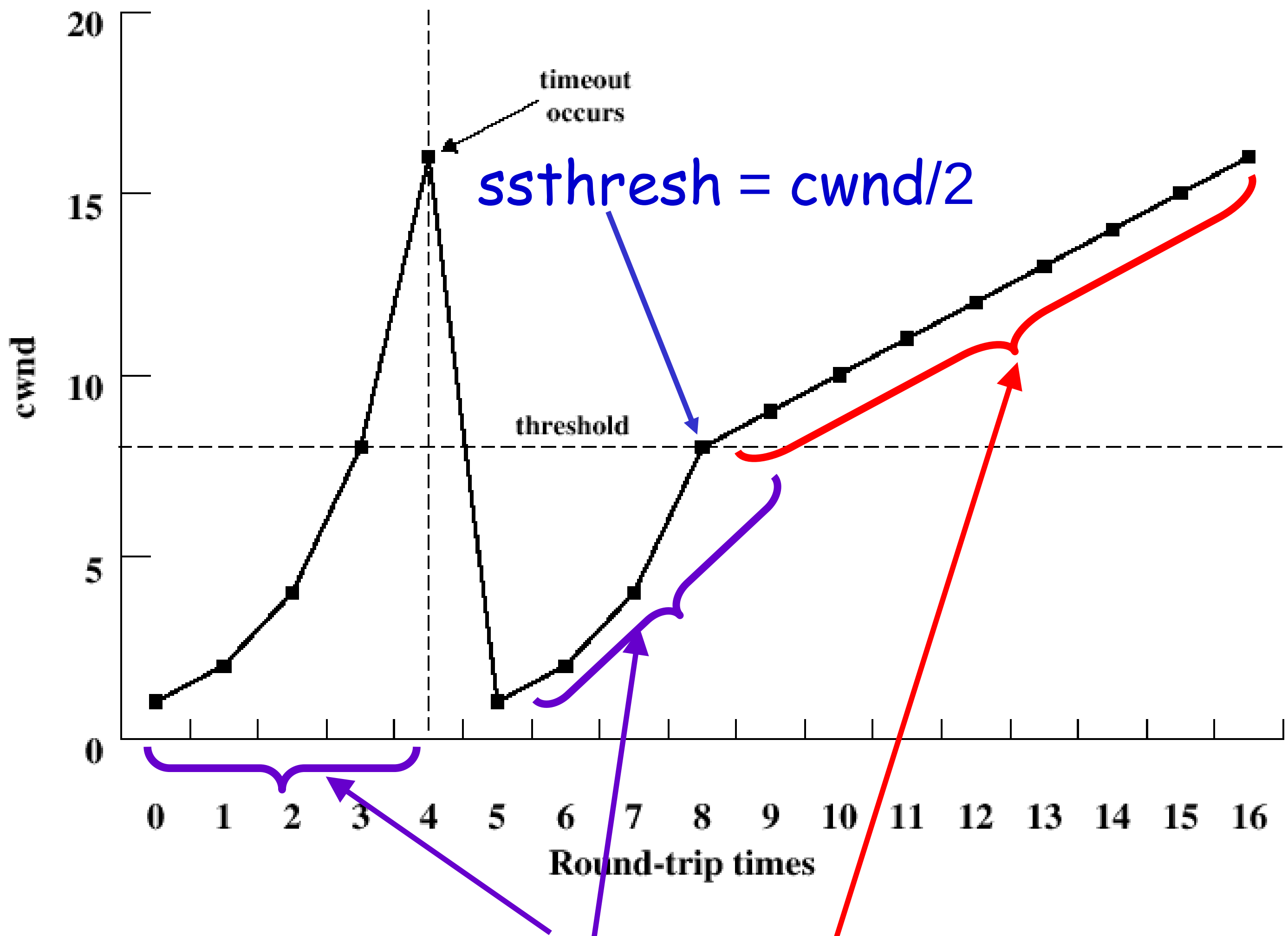


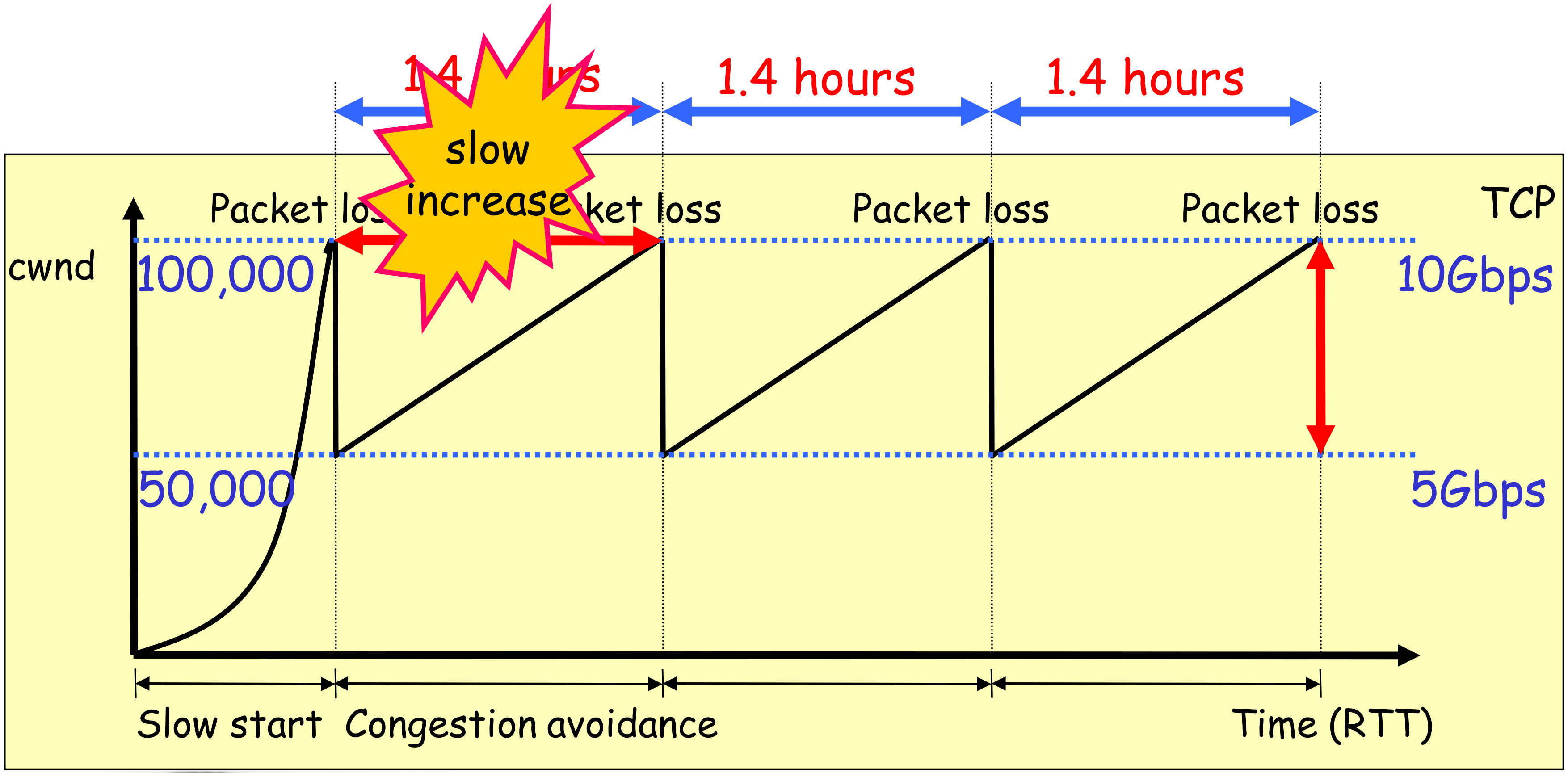
Figure 17.14 Illustration of Slow Start and Congestion Avoidance

New Window Management Algorithm

- Problem of linear increase
  - Many long fat networks: large bandwidth with long delay
  - Slow response of TCP in such networks leaves sizeable unused bandwidth
- An example
  - A TCP connection with *1250-Octet MSS* （*Maxitum Segment Size*） and *100ms RTT* on *10Gbps* network
  - To fully use the network, credit is big, and nearly ***1.4 hour*** is needed for linear increase

Too Slow Linear Increase

- Q: how to improve it

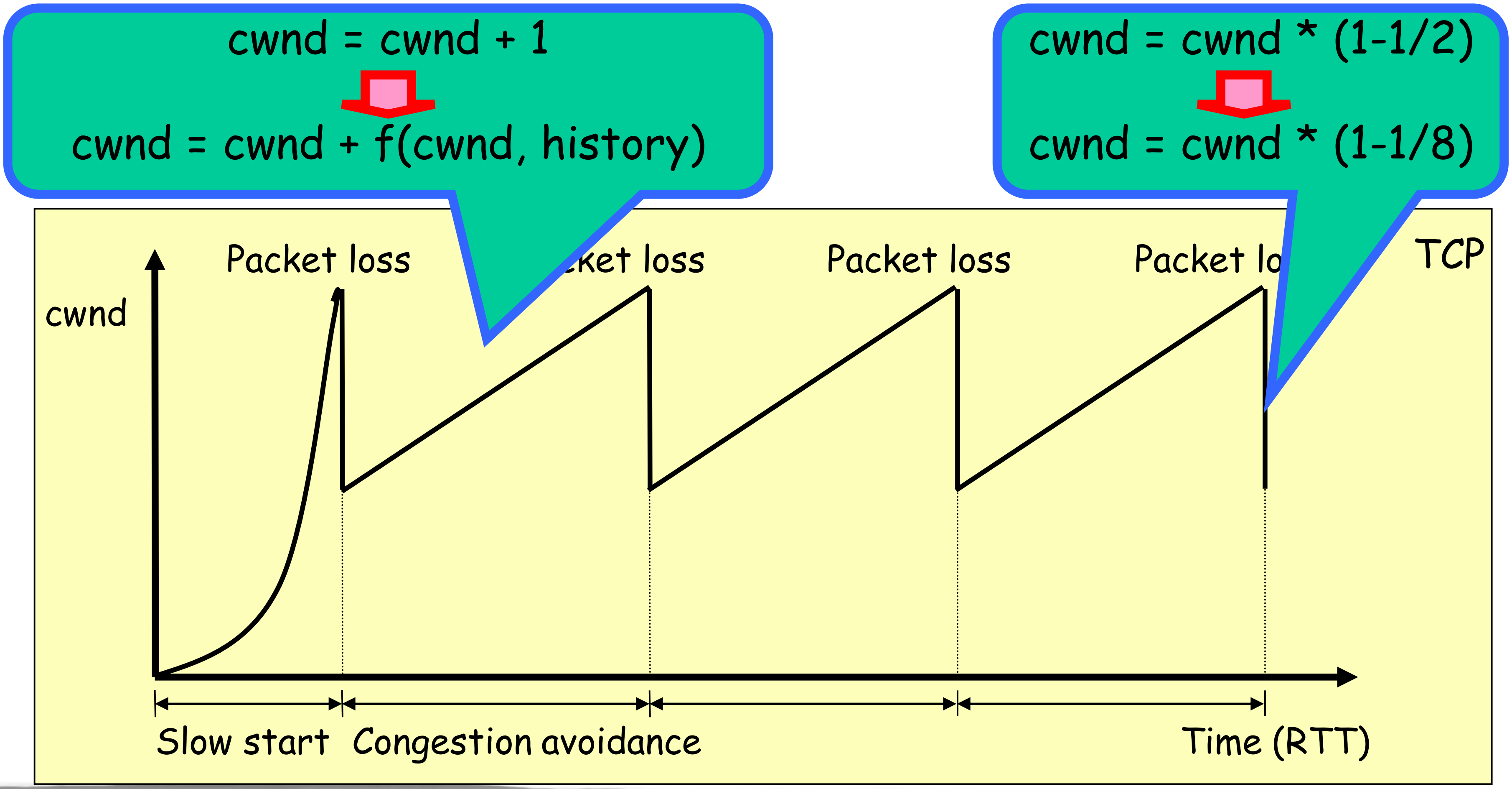


BIC and CUBIC

- BIC (Binary Increase Congestion control)
  - Implemented and used by default in Linux kernels 2.6.8
- CUBIC
  - The window is a cubic function of time since the last congestion event
  - Implemented and used by default in Linux kernels 2.6.19 and above

BIC

- BIC adaptively increase cwnd, and decrease cwnd by 1/8



BIC Overview

- 2 stages
  - Binary Search: increase window after congestion
  - Max Probing: search for better window size (until max credit)
    - （Max window size if set from history information. If there are more size available, Max_Probing is used for the purpose of exploring larger window size）
- 4 parameters defined
  - ***Smax*** : the maximum increment, e.g. *1/8×credit*
  - ***Smin*** : the minimum increment, e.g. *2 MSS*
    - *（Using S_max, S_min to avoid jitter: If a binary search steps too large, the traffic will change quickly. Thus S_max is used to limit the maximum change in one step）*
  - ***Wmax*** : maximum window size of current search, e.g. window size just before the lost
  - ***Wmin*** : minimum window size of current search, e.g. current window size without lost

BIC Stages

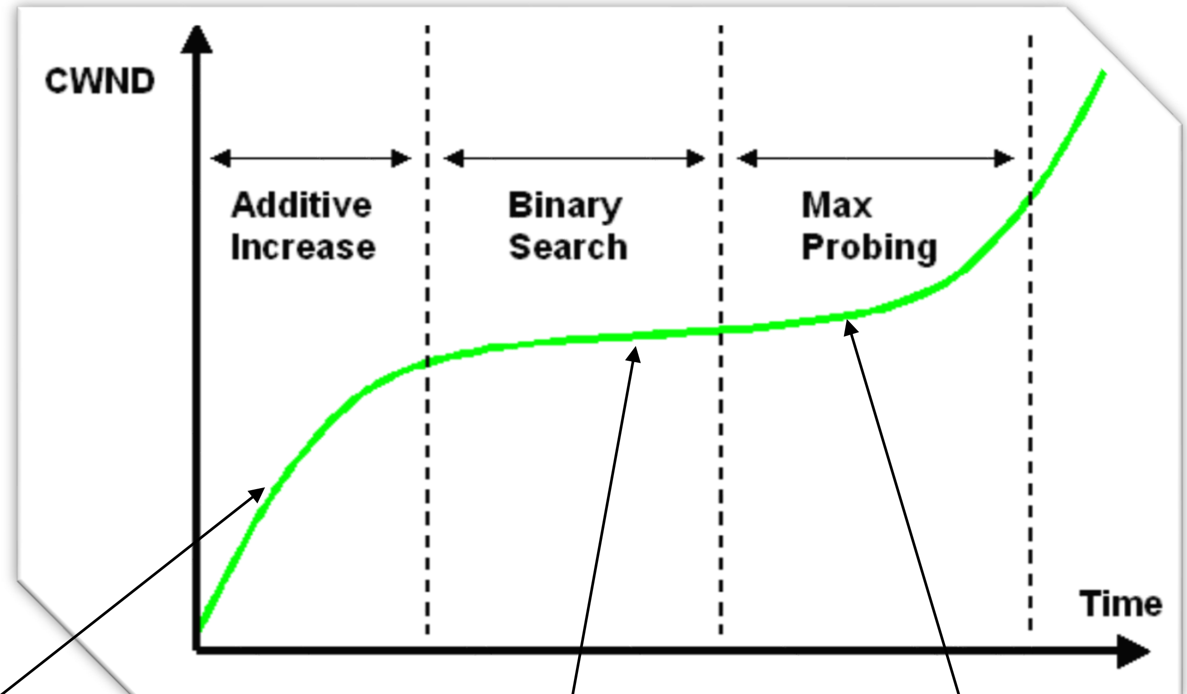
- Additive Increase
  - Linear increase with *inc*
- Binary search
  - How to set *inc*

```
while (Wmin <= Wmax){
    inc = (Wmin+Wmax)/2 - cwnd;
    if (inc > Smax) inc = Smax;
    else if (inc < Smin) inc = Smin;
    cwnd += inc;

    if (no packet losses) Wmin = cwnd;
    else {
        Wmax = cwnd;
        Wmin = cwnd×β    (e.g. 0.8)
    }
}
```

BIC Stages

- Binary Search Stage: Additive increase + Binary search
- Max Probing Stage: Binary search + Additive increase



At the beginning, each step increases S_max

In the middle, pure binary search

To probing more window size, each step increases by binary search or S_min

CUBIC

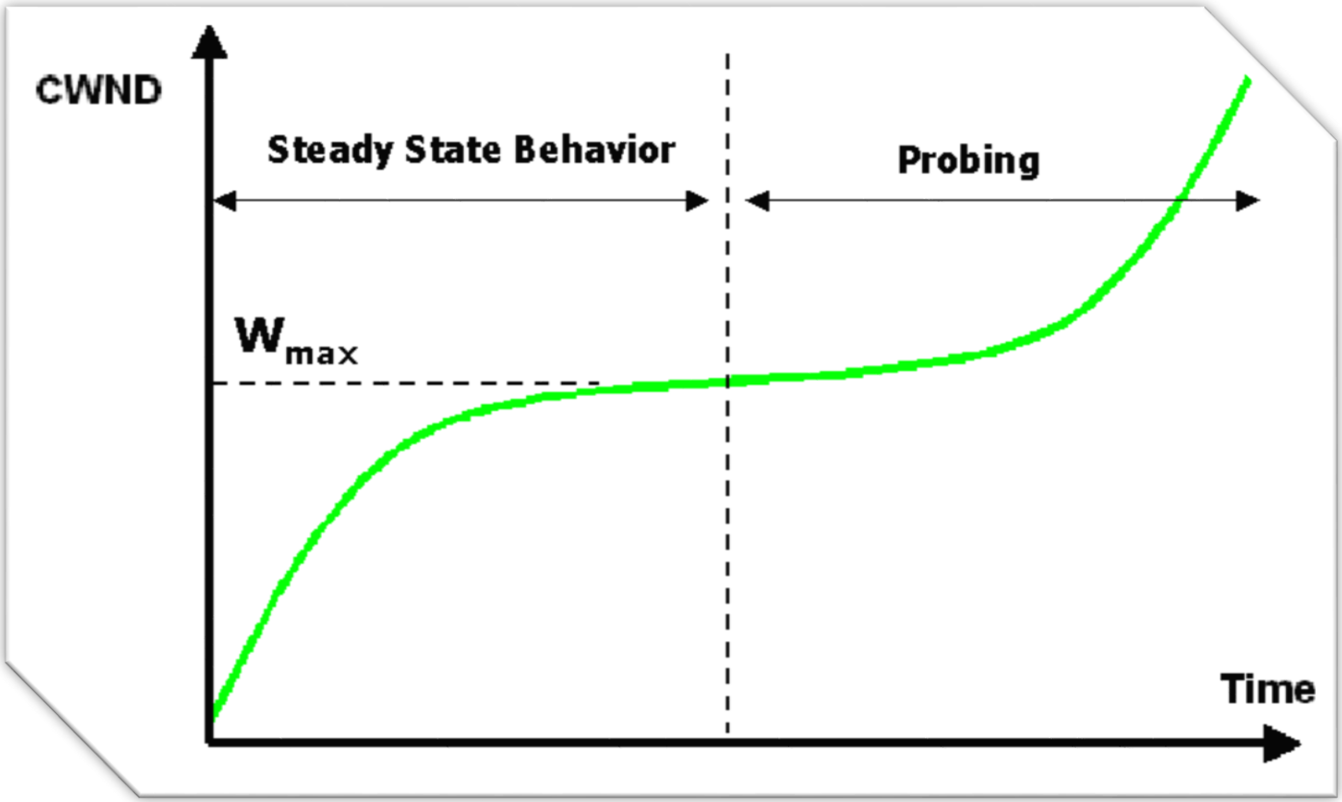
- BIC problem
  - The BIC's growth function may be too aggressive for TCP
  - BIC is not suitable for short RTT or low speed networks (may cause unfairness)
- Handle
  - Express the multi-stage BIC curve with a single cubic function

CUBIC Overview

- Parameters
  - ***Wcubic***: current cwnd
  - ***Wmax***: window size just before the last lost
  - ***T***: elapsed time from the last lost
  - ***C***: a scaling constant
- Function $W_{cubic} = C(T-K)^3 + W_{max}$

$$where\, K = \sqrt[3]{\frac{W_{max} \times (1-\beta)}{C}}$$

CUBIC Overview



Summary

- UDP & TCP
- TCP header fields
- TCP congestion control
  - Retransmission timer
  - Window management

Homework

- 书第22章习题：22.3，22.4，22.9，22.14，22.16，22.17，22.18，22.21