CSC361

Computer Networking

Mantis Cheng

Dept of Computer Science

Unit 6

Transmission Control Protocol (TCP)

Important Concepts

- Transmission Control Protocol (TCP)
- TCP Header
- Checksum
- 3-Way Handshake
- 4-Way Take-Down
- Flow Control
- RTT & Timeout Estimation
- Congestion Control

What We Learned So Far

- UDP is unreliable.
- **Stop-and-Wait** protocols are inefficient.
- Knowing the **bandwidth** and **RTT**, efficiency improves by filling the **pipeline** with packets.
- Sliding Windows protocols rely on sender's window size to control the size of the pipeline.
- **Go-Back-N** and **Selective-Repeat** are two of the most common sliding windows protocols.

TCP Service Model (16:27)

(a quick overview of TCP in first 8 minutes)

Summary

- TCP is a reliable bi-directional byte stream protocol.
- TCP is a connection-oriented **end-to-end** protocol, not running inside switches/routers.
- A **3-way** handshake is used to establish a **bi-directional** connection.
- **Initial sequence numbers** are exchanged at the end of a successful connection.

Summary (continued)

- Each end views the other sending a **continuous** stream of bytes.
- TCP guarantees both streams are delivered **in order** and **reliably**.
- TCP connection is closed using a **4-way** takedown.
- TCP segments are sent one-by-one; each segment has a max. length 64K bytes.
- TCP uses flow control to improves performance, and congestion control to prevent **Internet collapse**.

TCP Service Model (8:09-16:00)

(a quick overview of TCP header at 8:00)

Summary

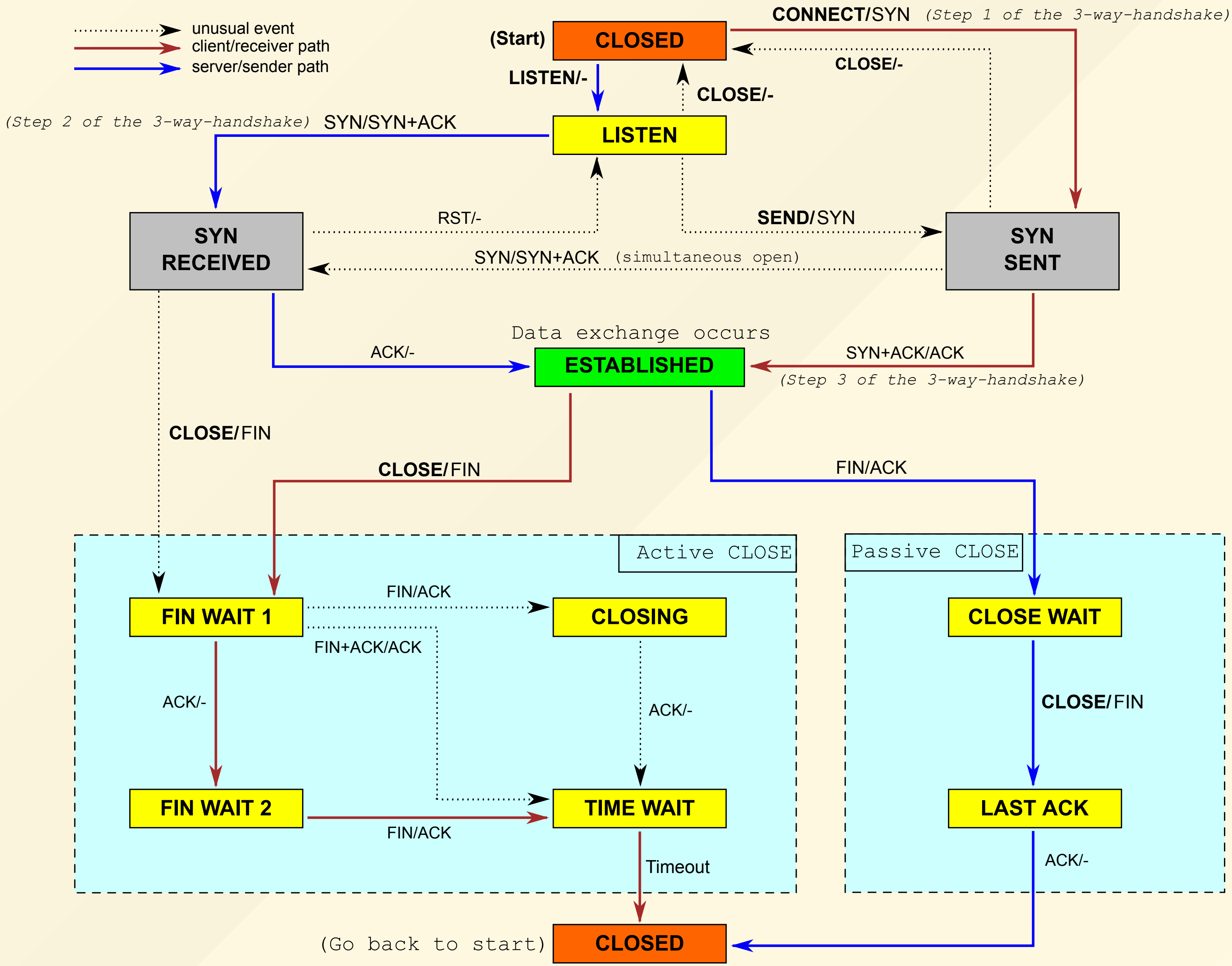
- Each TCP segment is preceeded with a header.
- The header includes: `Dest Port #`, `Source Port #`, `Seq. # of first byte`, `Ack Seq. # of last expected`, `Checksum`, and `Window Size`.
- The `Dest. Port #` identifies the TCP service needed.
- Each pair (`IP address`, `Port #`) identifes each endpoint; two pairs identifier a connection.
- The `Port #`s are used for multiplexing & demultiplexing.

Transport Layer Header (7:51)

(a more detailed explanation)

Summary

- Each `Port #` is 16-bit long; `Window Size` is the receiver's window size used in flow control.
- `Sequence #` is the **first** byte of the segment in the TCP stream; `Ack #` is the next byte **expected**.
- `Checksum` includes a **pseudo IP header** plus the TCP segment.
- `SYN`, `ACK`, `FIN` bits are used in connection setup/takedown.
- **Initial sequence** #s are **randomly** generated for security reasons.



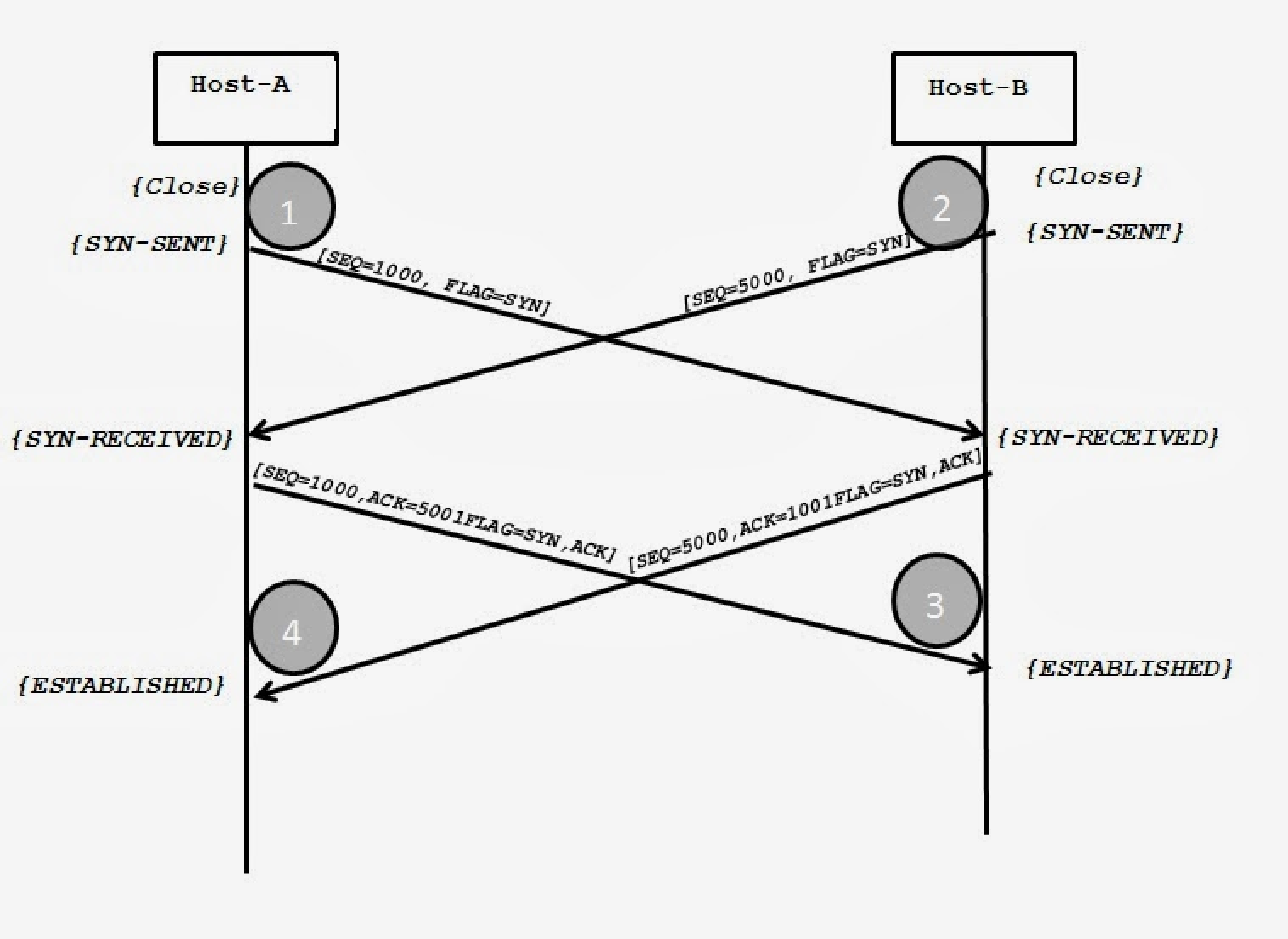
unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
LISTEN
(Step 2 of the 3-way-handshake) SYN/SYN+ACK
RST/-
SEND/SYN
SYN RECEIVED
SYN SENT
SYN/SYN+ACK (simultaneous open)
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
FIN+ACK/ACK
ACK/-
ACK/-
CLOSE/FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
ACK/-
(Go back to start)
CLOSED

Summary (3-way handshake)

- Both client and server start at CLOSED state.
- A server LISTENTs for client's connection initially.
- A client requests `CONNECT`, sends a `SYN` and enters SYN-SENT.
- The server gets a `SYN`, responds with a `SYN+ACK` then enters SYN-RECEIVED state.
- The client gets a `SYN+ACK`, responds with a `ACK` then enters ESTABLISHED.
- The server gets an `ACK` and enters ESTABLISHED.

Wireshark Demo

(use info.cern.ch.pcap; pay attention to `relative` initial sequence numbers)


Host-A
Host-B
{Close}
{Close}
1
2
{SYN-SENT}
{SYN-SENT}
[SEQ=1000, FLAG=SYN]
[SEQ=5000, FLAG=SYN]
{SYN-RECEIVED}
{SYN-RECEIVED}
[SEQ=1000,ACK=5001FLAG=SYN,ACK]
[SEQ=5000,ACK=1001FLAG=SYN,ACK]
3
4
{ESTABLISHED}
{ESTABLISHED}

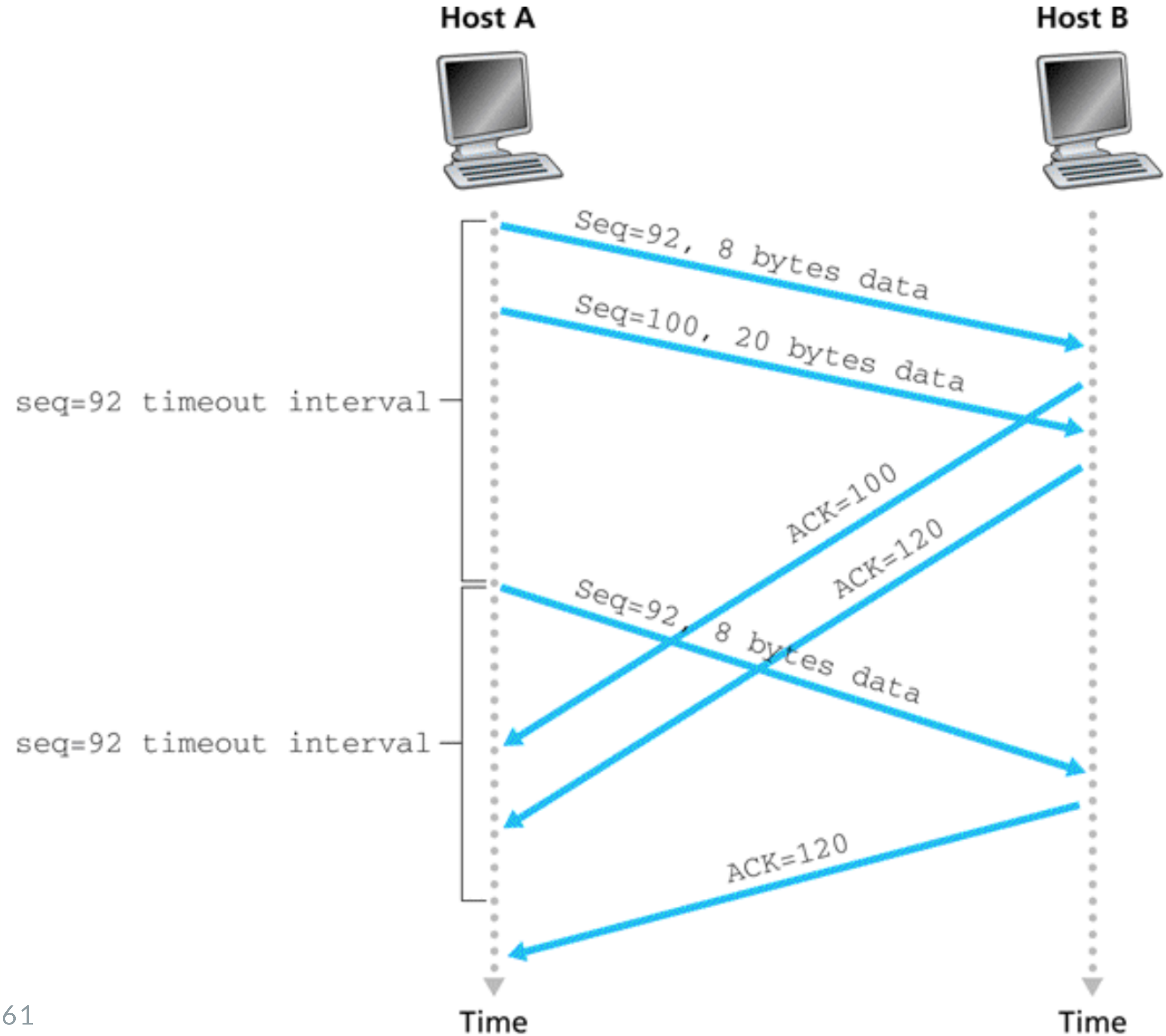
Summary (4-way Handshake)

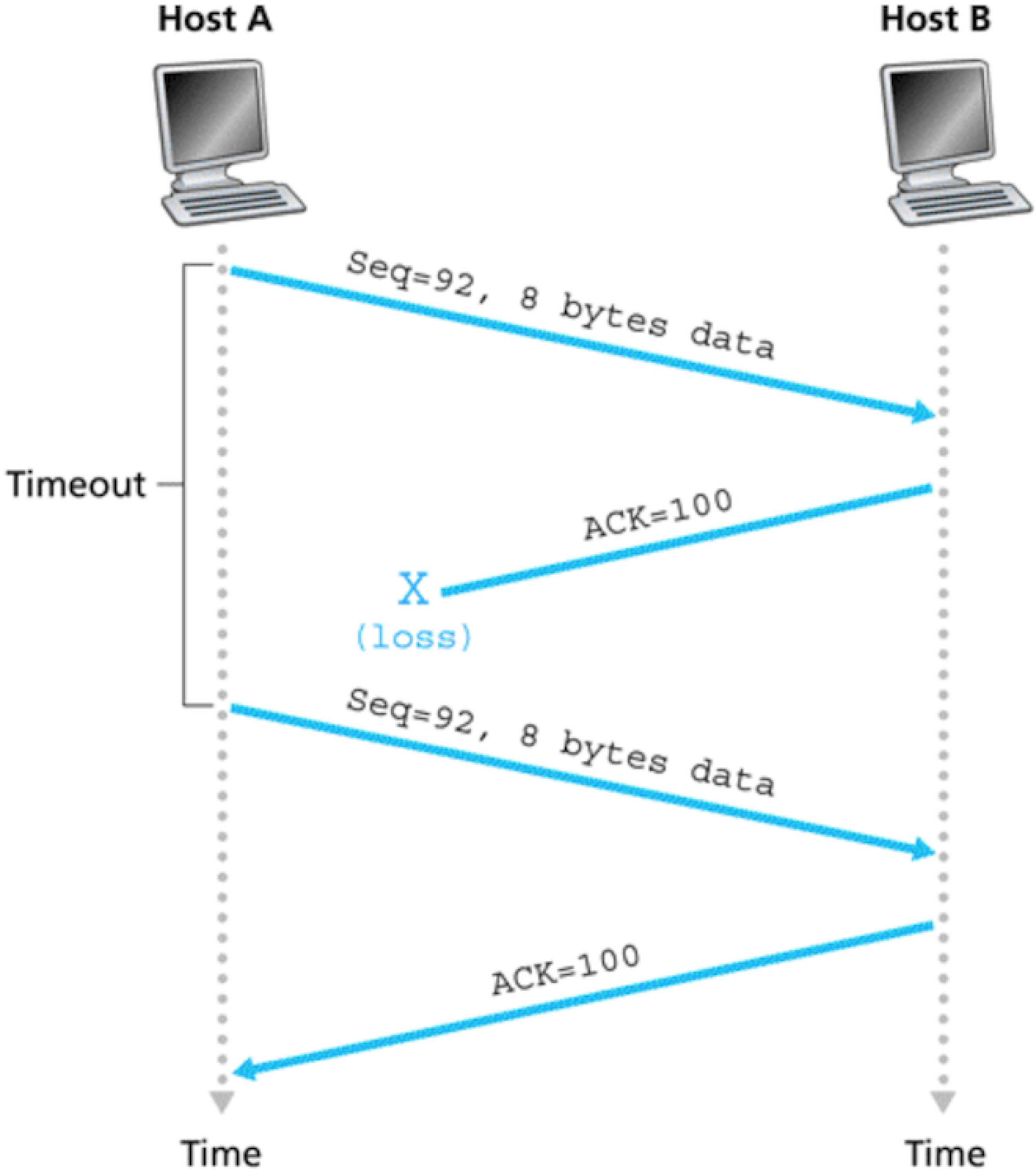
- A **simultaneous open** requires a 4-way handshake instead of 3-way.
- Both client and server are simultaneously **active** in a peer-to-peer application.
- Both send `SYN` at the same time, thus enter SYN-SENT state;
- Later, each receives a `SYN` from the other, thus responds with a `SYN+ACK` and enters SYN-RECEIVED state (**simultaneous open**).
- A `SYN+ACK` then establishes the connection.

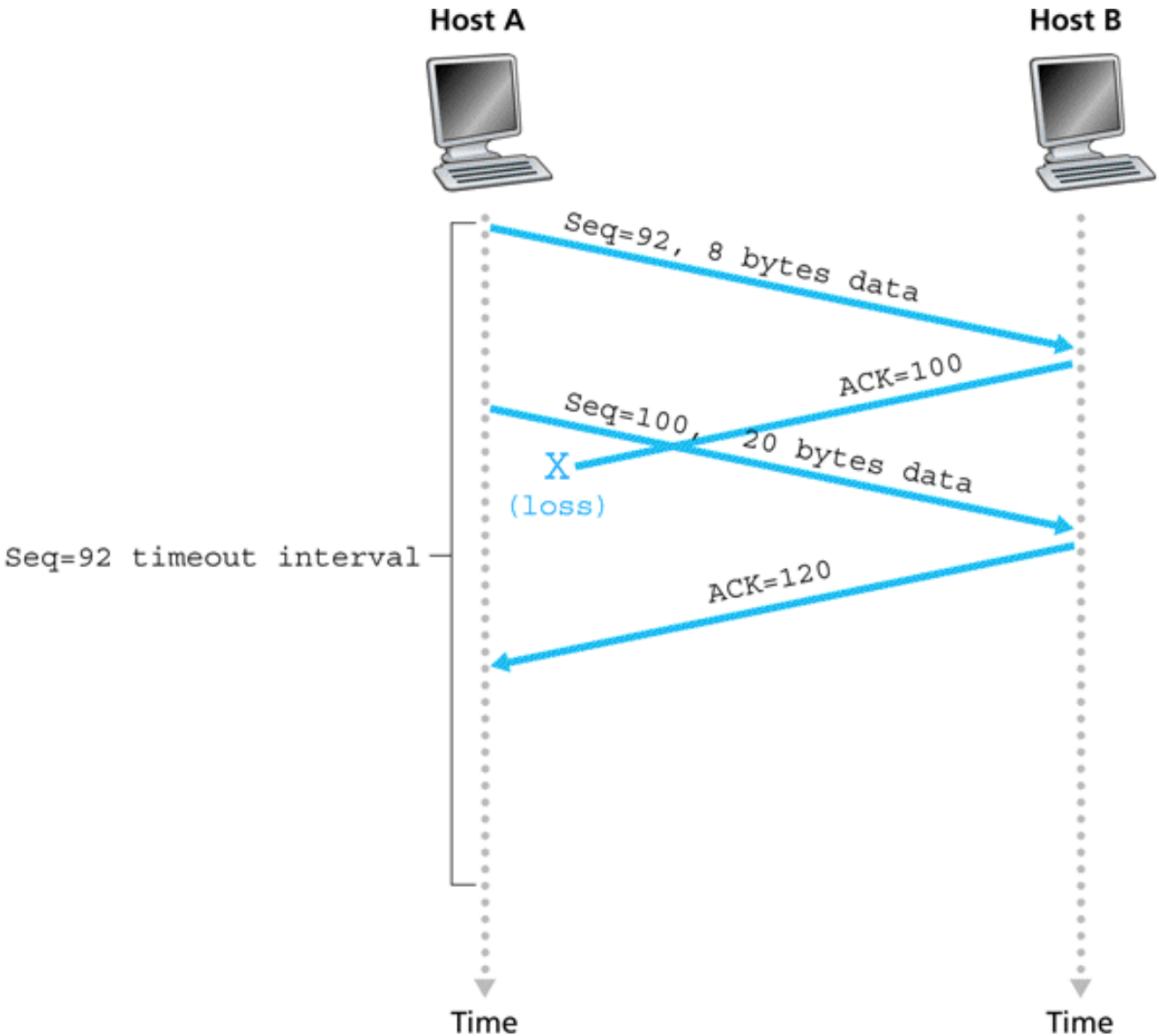
TCP Connection (19:49)

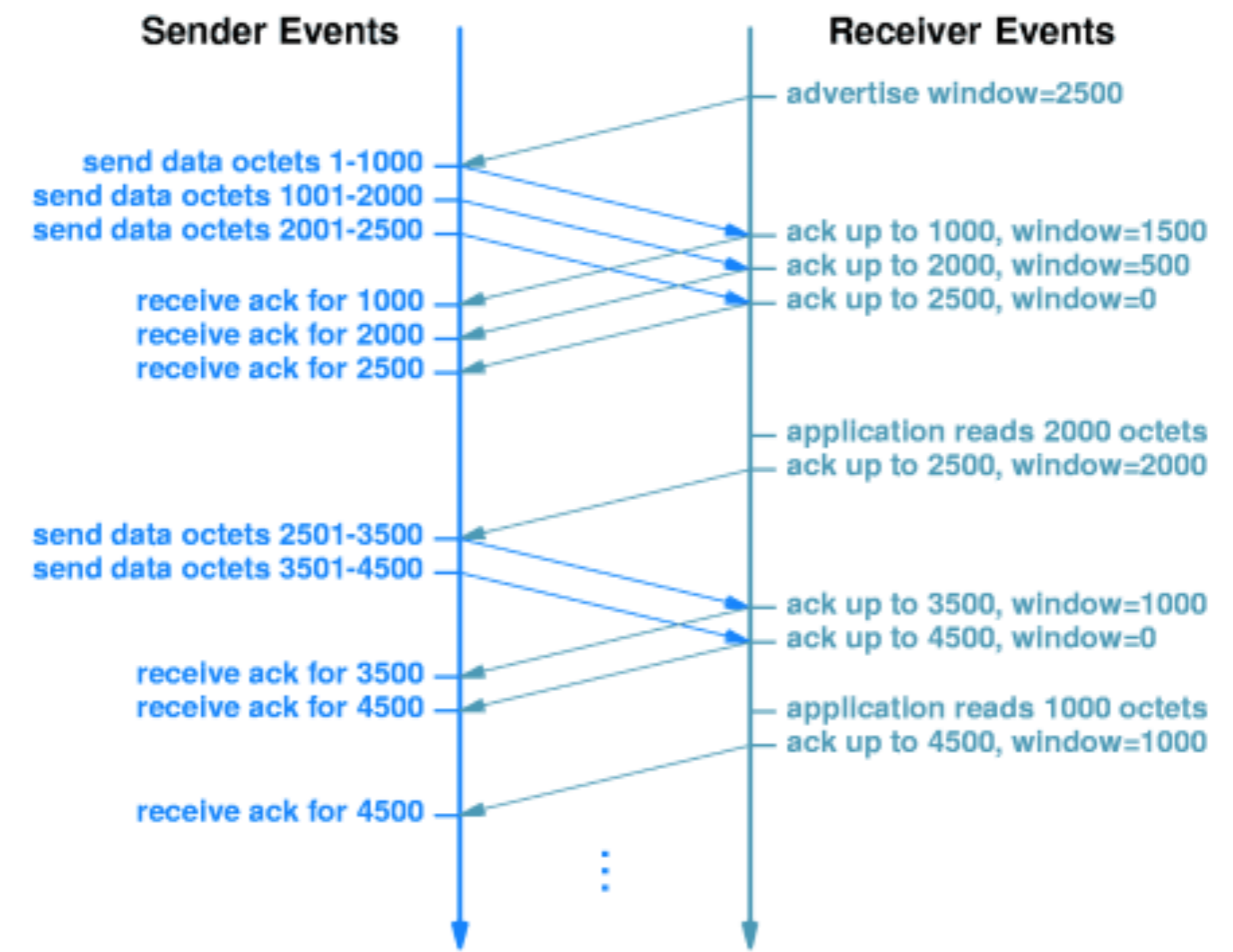
(a more detailed explanation of TCP 3-way or 4-way Handshake in the firs 10 minutes)

(a walk through of the TCP FSM after 10:20)


Host A
Host B
Seq=92, 8 bytes data
Seq=100, 20 bytes data
seq=92 timeout interval
ACK=100
ACK=120
Seq=92, 8 bytes data
seq=92 timeout interval
ACK=120
Time
Time


Host A
Host B
Seq=92, 8 bytes data
Timeout
ACK=100
X
(loss)
Seq=92, 8 bytes data
ACK=100
Time
Time


Host A
Host B
Seq=92, 8 bytes data
ACK=100
Seq=100, 20 bytes data
X
(loss)
Seq=92 timeout interval
ACK=120
Time
Time


Sender Events
Receiver Events
advertise window=2500
send data octets 1-1000
send data octets 1001-2000
send data octets 2001-2500
ack up to 1000, window=1500
ack up to 2000, window=500
ack up to 2500, window=0
receive ack for 1000
receive ack for 2000
receive ack for 2500
application reads 2000 octets
ack up to 2500, window=2000
send data octets 2501-3500
send data octets 3501-4500
ack up to 3500, window=1000
ack up to 4500, window=0
receive ack for 3500
receive ack for 4500
application reads 1000 octets
ack up to 4500, window=1000
receive ack for 4500

Flow Control I

(Flow Control Simulator)

Flow Control II

(Wireshark Demo with `web.uvic.ca.pcap`)

Summary

- TCP uses receiver's window size to flow control.
- The client sends as fast as possible as long as it doesn't overflow the receiver's window size.
- The client uses **a single timer** for **all** unACKed segments.
- The receiver buffers segments up to its windows size; it `ACK`s **cumulatively** up to **next** byte **expected**.
- Thus, RTT (or timeout estimation) is critical to the performance of TCP.

Timeout & Retransmisson

- If a client starts at `SEQ 1` and sends 5 segments of 1K each, and the server successfully receives all but the first segment, thus `ACK 1` four times.
- When the client times out or gets `ACK 1` four times, should it retransmit **all** segments or just the **first** segment?
- Retransmit **all** is wastful!
- Retransmit **only** the first segment if it knows only the first segment is missing. But, it doesn't!

RTT Estimation

- TCP relies on a **good** estimate of Round-Trip-Time (RTT) to set its timeout interval.
- RTT is **not** constant; it varies from time-to-time.
- TCP uses an **exponential averaging** algorithm to estimate the **effective** RTT from the **observed** RTT.
- A **timeout** is then calculated based on the deviations and the estimates of RTT.

Exponential Averaging on RTT (I)

- Let S_t be the **sampled** and E_t be the **estimated** round-trip delays.

$$E_t = (1-\alpha) * E_{t-1} + \alpha * S_t$$

where $\alpha = 0.125$. It essentially does a Low-Pass-Filter on the observed RTTs.

- E_0 is an **initial** estimate, and S_1 is the **first sampled** RTT.

Exponential Averaging on RTT (II)

- Let $\delta_t = |E_t - S_t|$ be the **estimated error** deviation at t, and

$$\Delta_t = (1 - \beta) * \Delta_{t-1} + \beta * \delta_t$$

where $\beta = 0.25$, and Δ_0 is the **initial** error deviation, then

$$Timeout_t = E_t + 4 * \Delta_t$$

Timeout

t	E_t	S_t	Delta_t	ABS(E_t-S_t)	Timeout
0	95.00		4.00		1000
1	95.63	100	4.09	4.38	112
2	96.80	105	5.12	8.20	117
3	99.70	120	8.92	20.30	135
4	99.11	95	7.71	4.11	130
5	100.72	112	8.61	11.28	135

[Retransmission Timeouts (10:07)]

Fast Retransmit

- Due to the nature of **cumulative** ACKs, multiple **duplicate** `ACK n` of the same `n` can be received by the sender in succession!
- It is likely the result of a **missing first** segments followed by a series of delivered segments.
- When a sender gets **triple** duplicate `ACK n`s (actually four `ACK n`s), it **immediately retransmits** the segment beginning at `n` without waiting for a timeout.

Approaches to Congestion Control (19:22)

(basic approaches to Congestion Control)

Summary

- Network-based congestion control is complex and ineffective!
- Transport-based congestion control is **end-to-end** congestion control.
- TCP uses timeouts (packet loss) and self-clocking to control congestion.
- TCP flow control is **receiver-based** windowing; TCP congestion control is **sender-based** Windowing.

Summary (continued)

- The sender varies its sliding window according to flow control and congestion control.
- **AIMD** (Additive Increase Multiplicative Decrease) principle is about **bandwidth probing**! What is the **effective** bandwidth that the sender can use?
- Upon receiving an `ACK`, the sender increases its congestion window by $1/W$, where W is its congestion window size.
- How does the sender detect **packet loss**?

TCP Tahoe '88 (22:18)

(basics of TCP Congestion Control)

AIMD Principle

(Additive Increase Multiplicative Decrease)

[TCP Congestion Control (6:09)]

(summary of what Congestion Control is about)

Summary

- How to maximize link utilization (throughput in bps) without getting into congestion?
- What is the **effective** bandwidth-delay product? How big is the connection **pipe** (in bits)?
- If we can transmit p bits within an RTT (in secs), then the throughput is p/RTT bps.
- One would like to keep the **pipe** full without losing too many packets; an `ACK` indicates **success**; a `timeout` indicates packet **loss**.

TCP Congestion Control II (13:38)

(explains the design goals of Congestion Control)

Summary

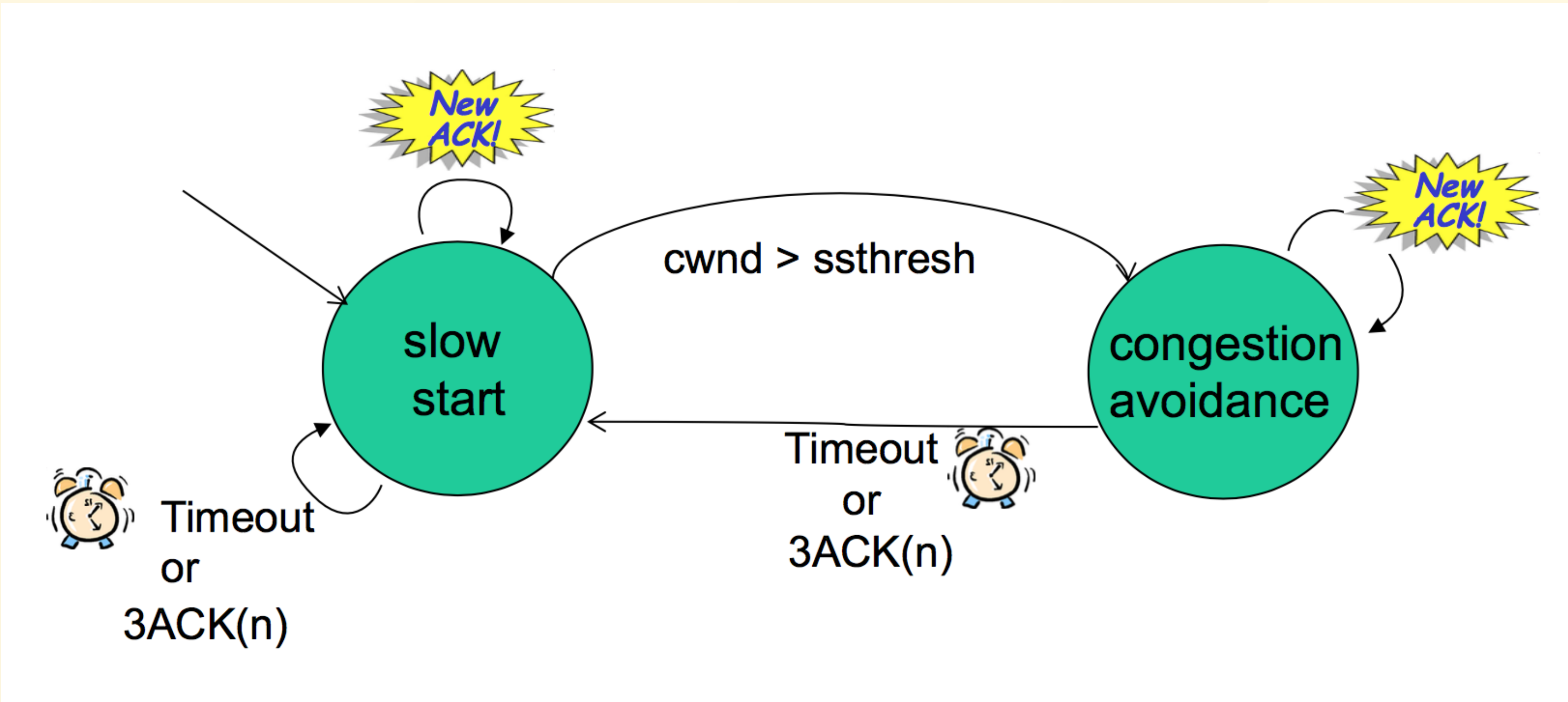
- How fast can a sender transmit its packets? Not how much? But, how fast?
- If it sends w packets per RTT (sec), then its transmission rate is $w * l/RTT$ (bps), where l is the packet length.
- By varying the window size w per RTT, a sender can throttle its transmissioin rate. Ideally, $w * l = p$.
- w depends on the **effective** bandwidth-delay.

[TCP Tahoe (14:21)]

(detailed explanation of TCP Tahoe'88)

Summary

- A **congestion windows** (`cwnd`) is used for **self-clocking**, which is determined by the rate of `ACK` received.
- Its goal is to probe the **effective bandwidth** by relying on the rate of positive `ACK`s.
- Initially, `cwnd`=64KB/`MSS`, (in `MSS` units). `MSS` is typically `1460` bytes.
- A `ssthresh` state variable is used to trigger a **phase change**.


New ACK!
cwnd > ssthresh
New ACK!
slow start
congestion avoidance
Timeout or 3ACK(n)
Timeout or 3ACK(n)

TCP Tahoe'88

Slow-Start (SS) ('88)

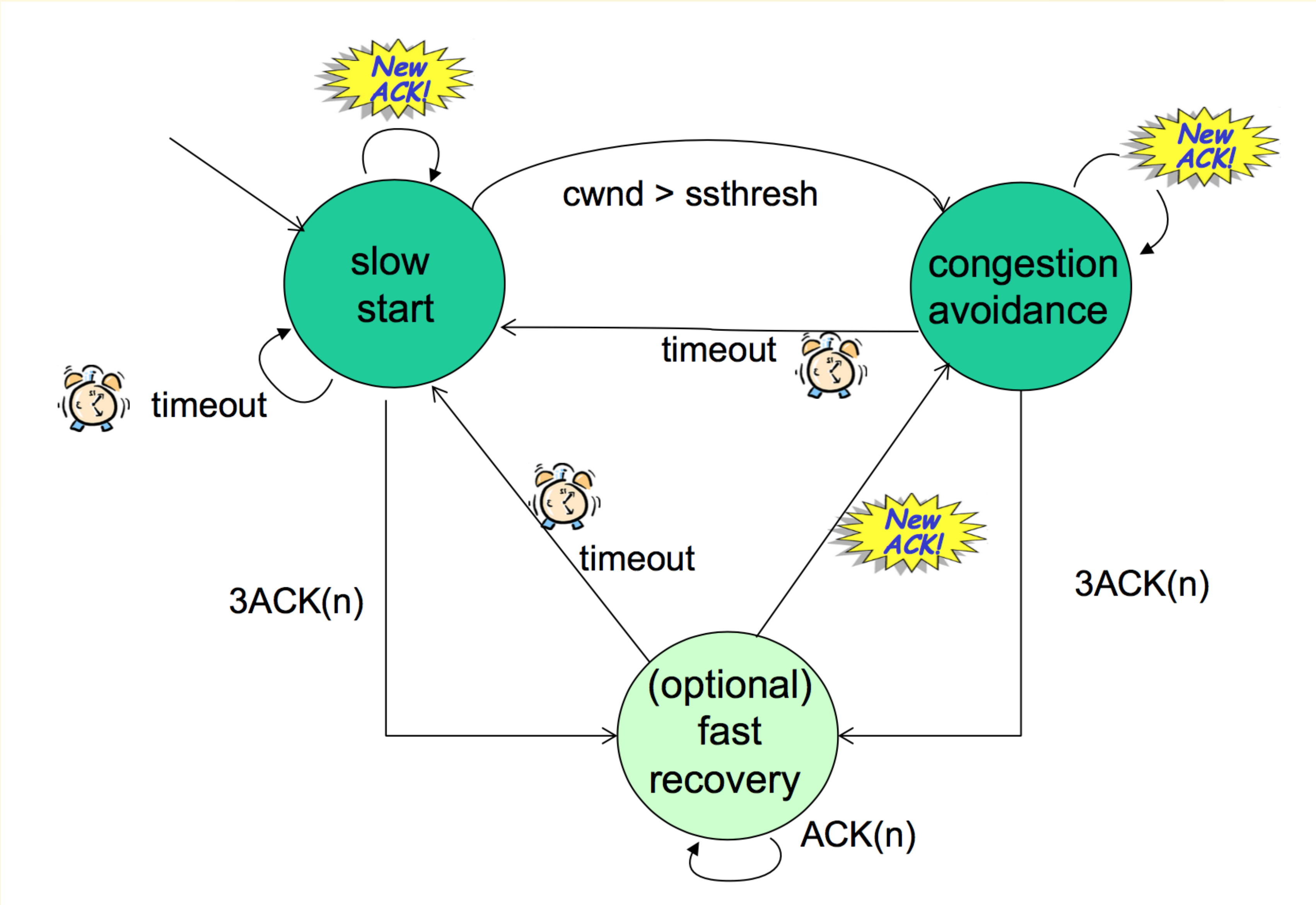
- `ssthresh` = `cwnd`/2; // multiplicative decrease
- `cwnd`=1;
- repeat // bandwidth probing
  1. on **new** `ACK`:
     - `cwnd` += 1; transmit new segments;
     - if (`cwnd`>=`ssthresh`) goto **CA**
  2. on `timeout`: retransmit all unACK'ed segments; go to **SS**

Congestion-Avoidance (CA)('88)

repeat: // additive increase

1. on **new** `ACK`:
   - `cwnd` += 1/`cwnd`; transmit new segments;
2. on `timeout`: goto **SS**.

[TCP Reno'90 (16:01)]


New ACK!
slow start
cwnd > ssthresh
New ACK!
congestion avoidance
timeout
timeout
timeout
New ACK!
3ACK(n)
3ACK(n)
(optional) fast recovery
ACK(n)

TCP Reno'90

Slow-Start (SS) ('90)

- `ssthresh`=`cwnd`/2; `cwnd`=1;
- repeat // bandwidth probing

1. on **new** `ACK`:
   - `cwnd` += 1; transmit new segments;
   - if (`cwnd`>=`ssthresh`) goto **CA**;
2. on `timeout`:
   - retransmit all unACK'ed segments, goto **SS**;
3. on `triple ACK(n)`: goto **FR**;

Congeston-Avoidance (CA) ('90)

- repeat: // linear increase

  1. on **new** `ACK`:
     - `cwnd` += 1/`cwnd`; transmit new segments;
  2. on `timeout`: go to **SS**.
  3. on `triple ACK(n)`: goto **FR**;

Fast Recovery (FR) ('90)

- retransmit **missing** segment `n`;
- `ssthresh`= `cwnd`/2; `cwnd`= `ssthresh`+3;
- repeat

  1. on **new** `ACK`: goto **CA**;
  2. on `timeout`: goto **SS**;
  3. on **duplicate** `ACK(n):

     - `cwnd` += 1; retransmit all segments from `n` up to `cwnd`;

The End