End-to-End Protocols

Hui Chen

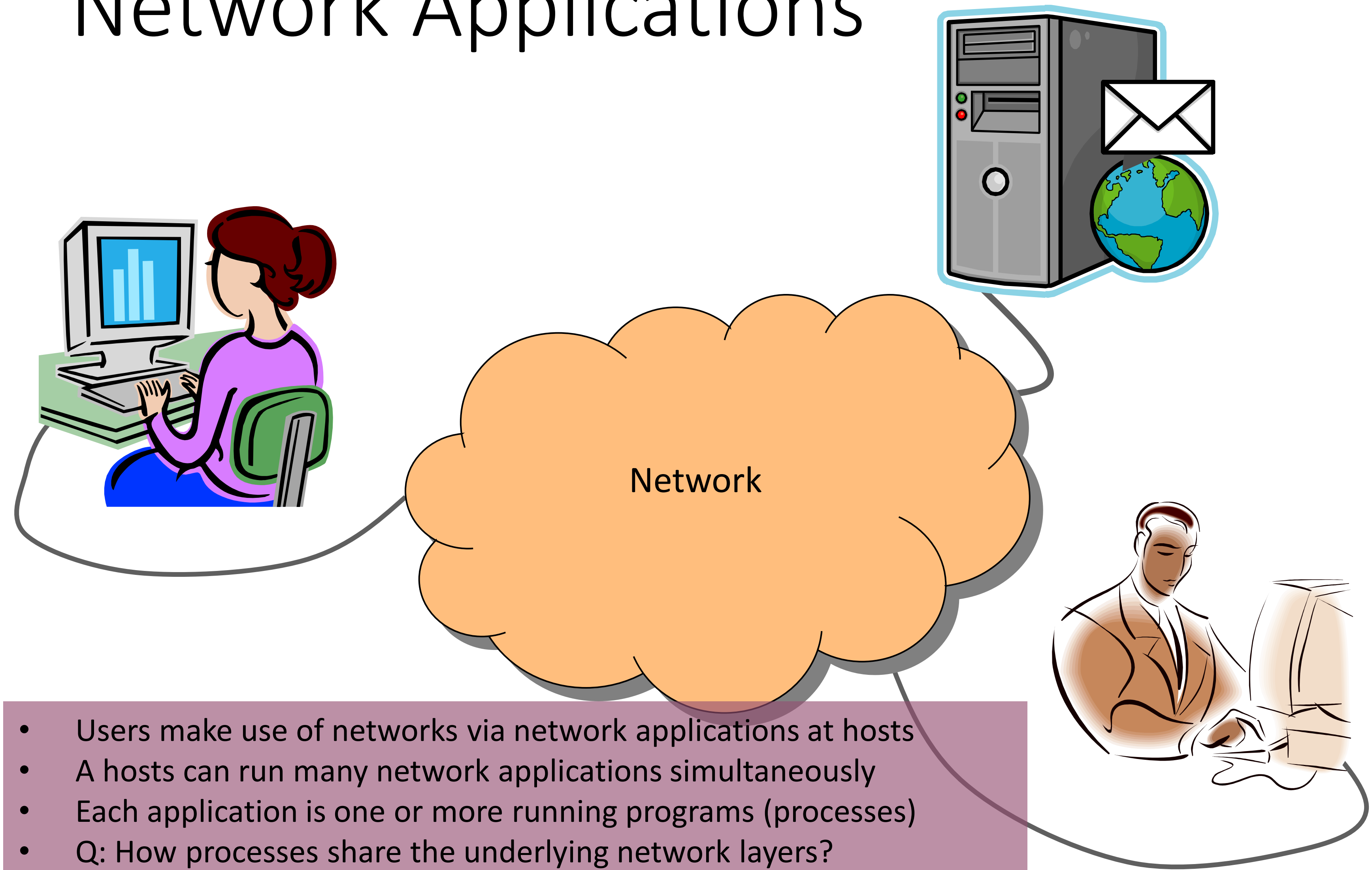
Department of Computer & Information Science

CUNY Brooklyn College

Outline

- Problem: the end-to-end communications?
- User Datagram Protocol
- Transmission Control Protocol

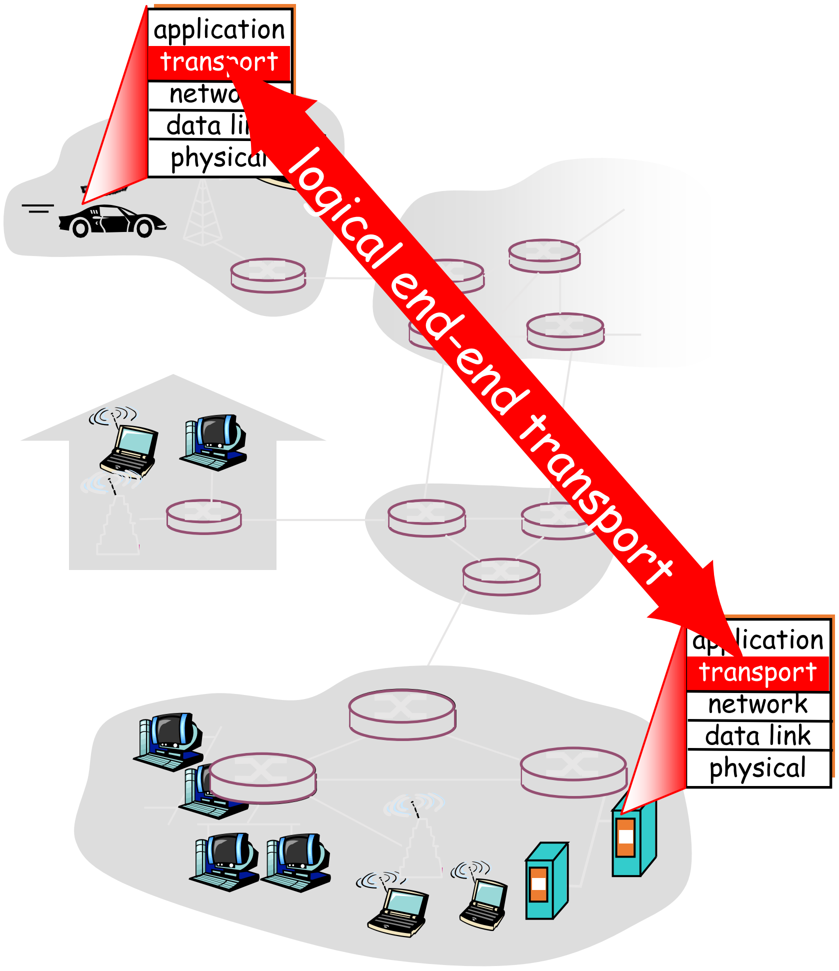
Network Applications



- Users make use of networks via network applications at hosts
- A hosts can run many network applications simultaneously
- Each application is one or more running programs (processes)
- Q: How processes share the underlying network layers?

Transport Layer Services and Protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols run in end systems
  - send side
    - breaks app messages into segments, passes to network layer
  - receive side:
    - reassembles segments into messages, passes to app layer
- more than one transport protocol available to applications
  - Internet: TCP and UDP



Transport vs. Network Layer (1)

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

Household analogy:

12 kids sending letters among themselves via their parents

- processes = kids
- application messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill (parents)
- network-layer protocol = postal service

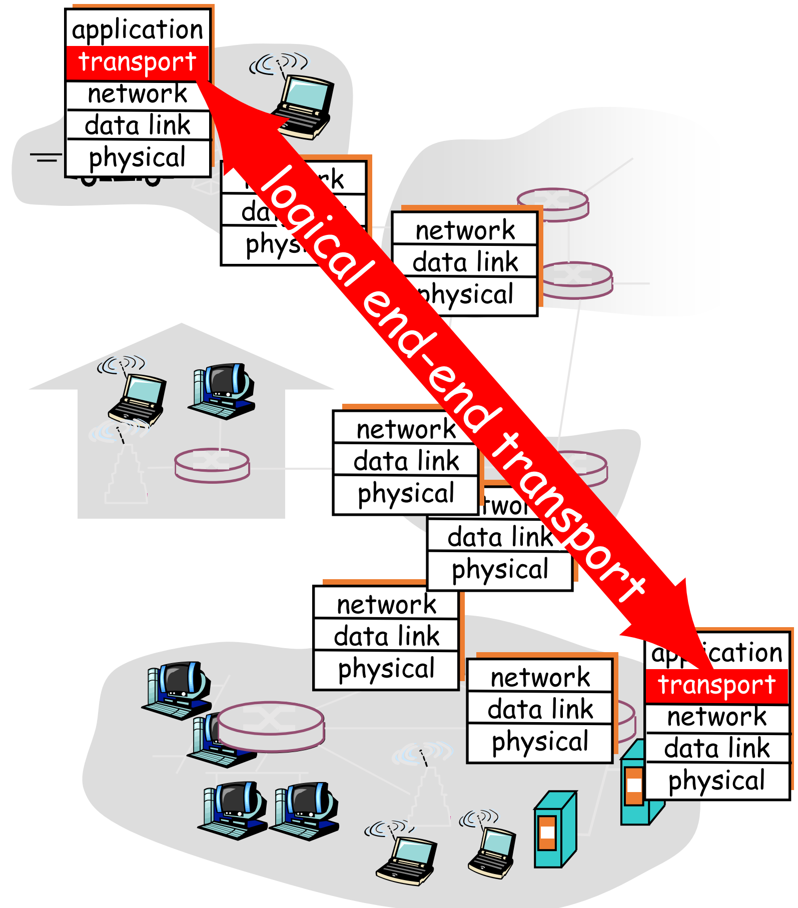
Transport vs. Network Layer (2)

- Network layer: Underlying best-effort network
  - drop messages
  - re-orders messages
  - delivers duplicate copies of a given message
  - limits messages to some finite size
  - delivers messages after an arbitrarily long delay

- Transport Layer: Common end-to-end services
  - guarantee message delivery
  - deliver messages in the same order they are sent
  - deliver at most one copy of each message
  - support arbitrarily large messages
  - support synchronization
  - allow the receiver to flow control the sender
  - support multiple application processes on each host

Internet Transport-Layer Protocols

- Reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- Unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- Services not available:
  - delay guarantees
  - bandwidth guarantees



Multiplexing/Demultiplexing

<u>Host-to-host delivery</u>

<u>vs</u>

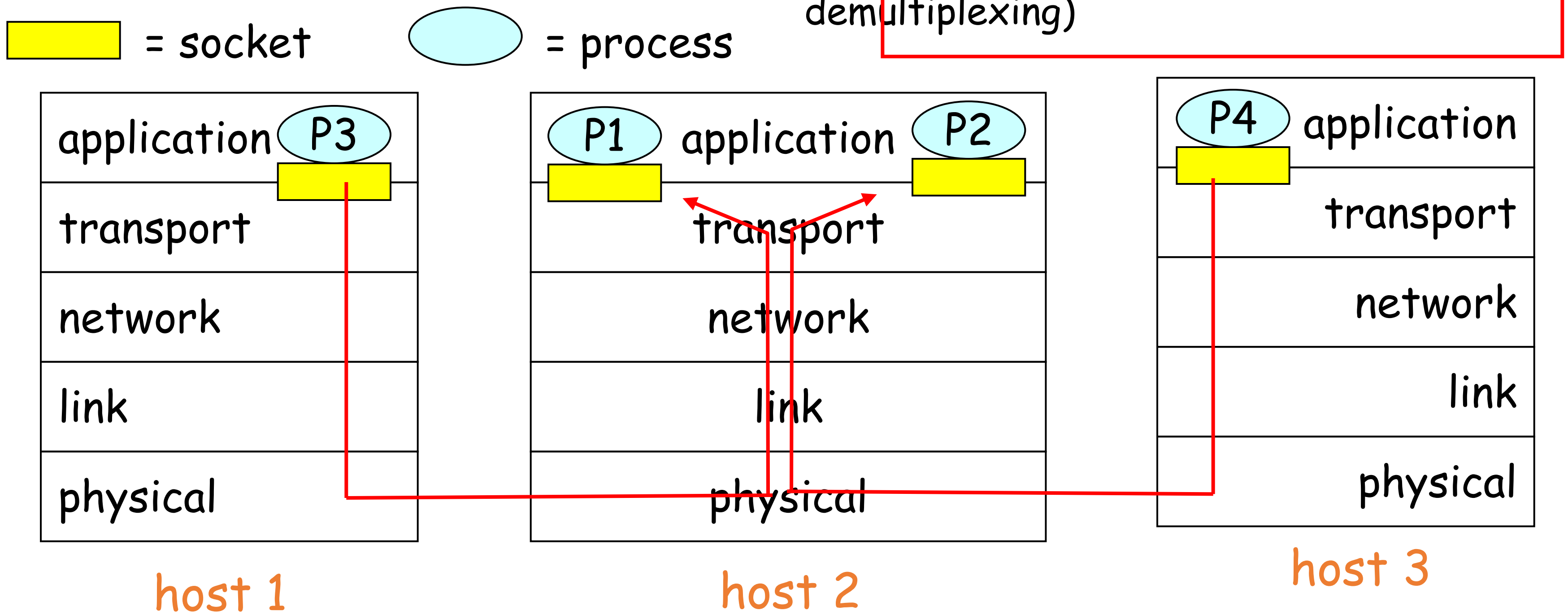
<u>process-to-process delivery</u>

Multiplexing/Demultiplexing

Host-to-host delivery ←→ process-to-process delivery

Demultiplexing at rcv host:
delivering received segments to correct socket

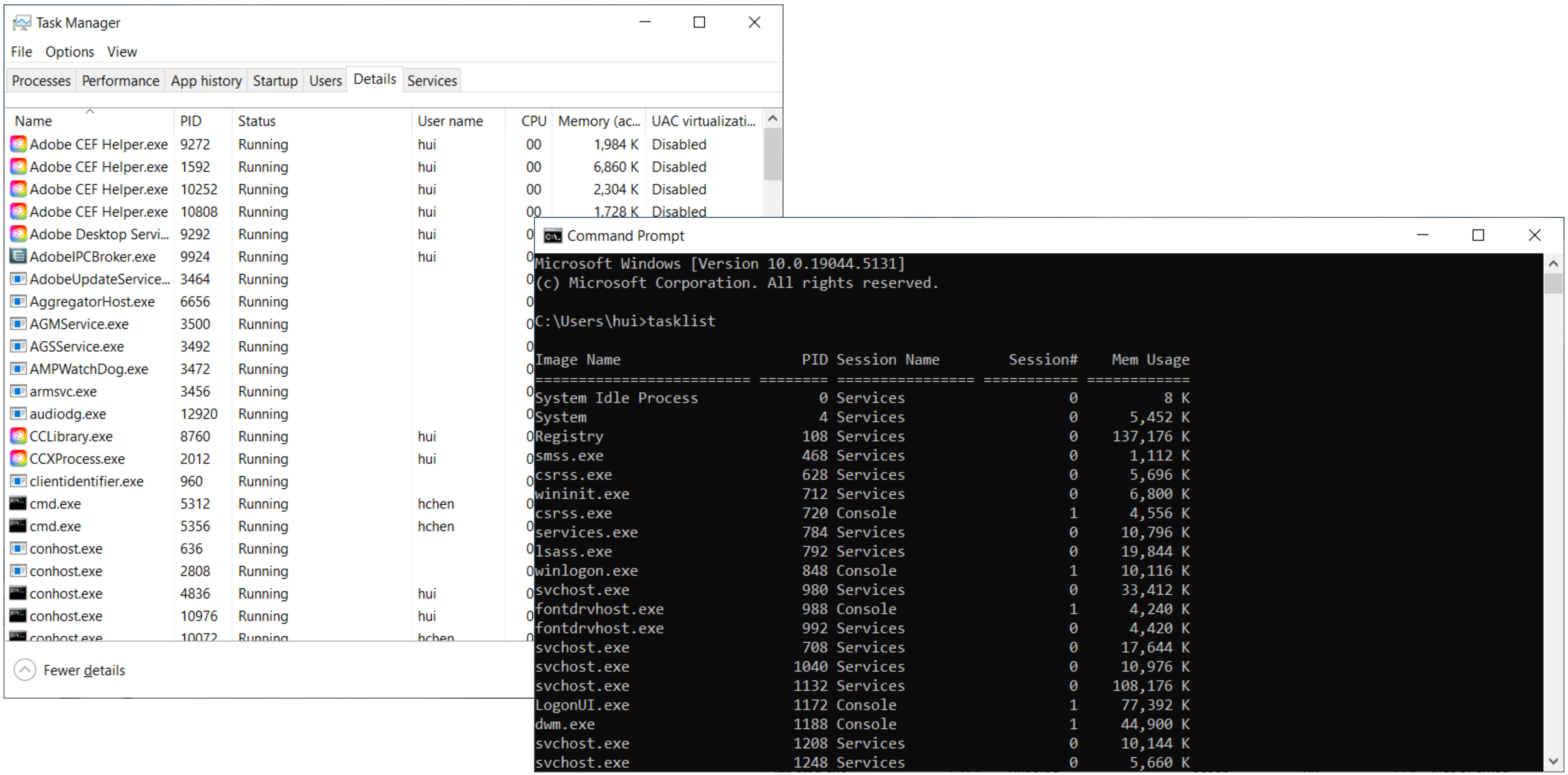
Multiplexing at send host:
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



Simple Demultiplexer (1)

- Need to know to or from which process the data is sent or come
  - Identify processes on hosts
- How to identify processes on hosts?
  - Introduce concept of “port”
  - *Q: why not to use process id?*

Processes: Windows Example



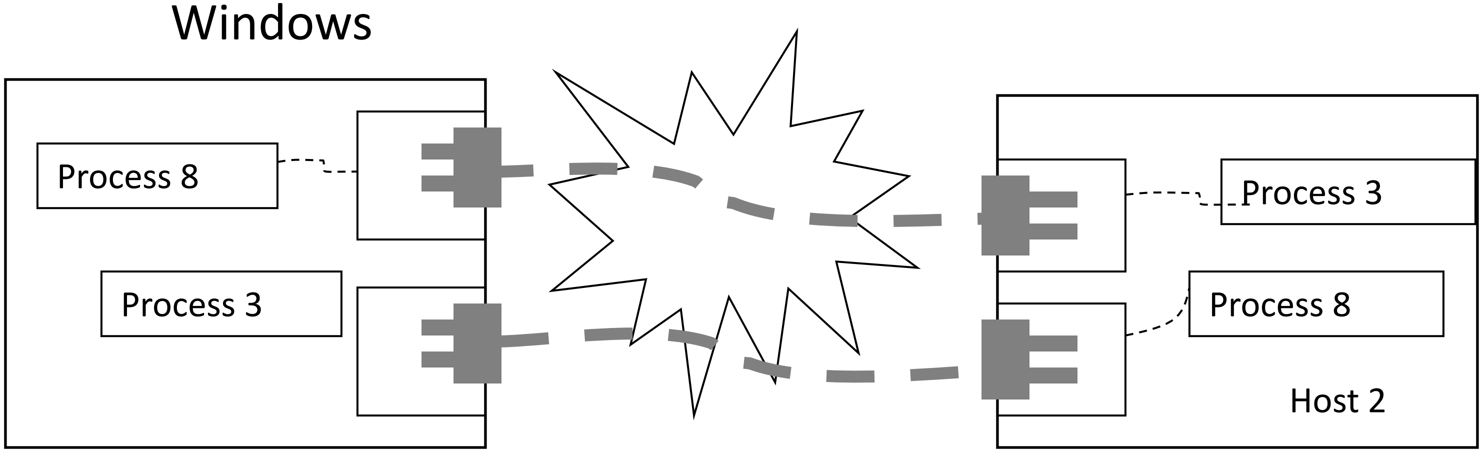
Processes: Linux/Unix Example

brooklyn@midwood: ~

```
brooklyn@midwood:~$ ps -ax
  PID TTY       STAT   TIME COMMAND
    1 ?         Ss     0:01 /sbin/init
    2 ?         S      0:00 [kthreadd]
    3 ?         I<     0:00 [rcu_gp]
    4 ?         I<     0:00 [rcu_par_gp]
    5 ?         I      0:00 [kworker/0:0-ata_sff]
    6 ?         I<     0:00 [kworker/0:0H-kblockd]
    7 ?         I      0:00 [kworker/u2:0-events_unbound]
    8 ?         I<     0:00 [mm_percpu_wq]
    9 ?         S      0:00 [ksoftirqd/0]
   10 ?         I      0:00 [rcu_sched]
   11 ?         I      0:00 [rcu_bh]
   12 ?         S      0:00 [migration/0]
   13 ?         I      0:00 [kworker/0:1-events_power_efficient]
   14 ?         S      0:00 [cpuhp/0]
   15 ?         S      0:00 [kdevtmpfs]
   16 ?         I<     0:00 [netns]
   17 ?         S      0:00 [kauditd]
   18 ?         S      0:00 [khungtaskd]
   19 ?         S      0:00 [oom_reaper]
   20 ?         I<     0:00 [writeback]
   21 ?         S      0:00 [kcompactd0]
   22 ?         SN     0:00 [ksmd]
```

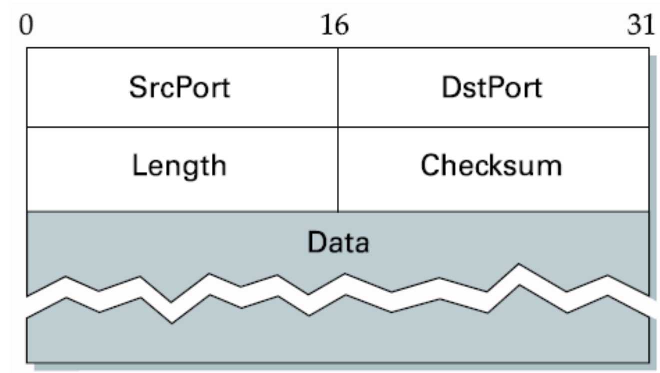
Simple Demultiplexer (2)

- How to identify processes on hosts?
  - *<u>Q: why not to use process id?</u>*
  - Introduce concept of "port"
    - Endpoints identified by ports
    - servers have well-known ports
    - see /etc/services on Unix/Linux
    - see C:\WINDOWS\system32\drivers\etc\services on MS Windows



Simple Demultiplexer: UDP

- Adds multiplexing to Internet Protocol
  - Endpoints identified by ports (UDP ports)
  - Demultiplex via ports on hosts
  - Nothing more is added
    - Unreliable and unordered datagram service
    - No flow control
  - User Datagram Protocol (UDP)
    - A process is identified by <host, port>
    - Connectionless model

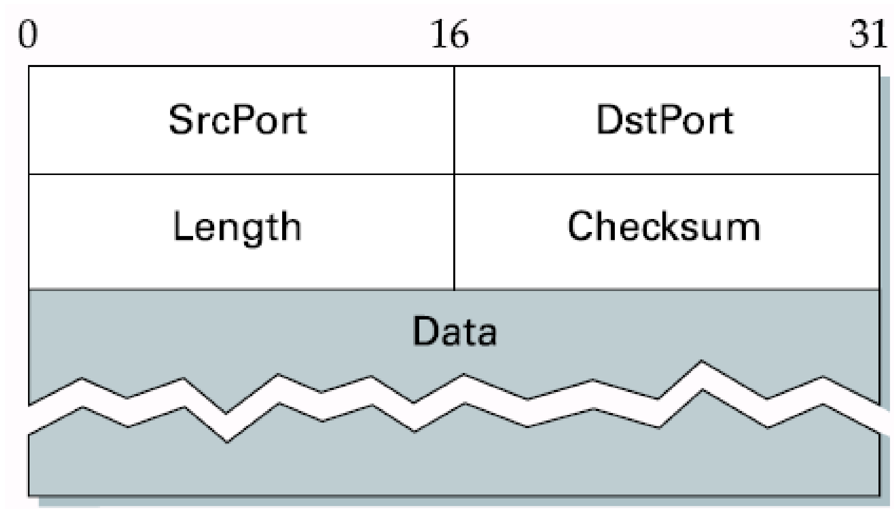


- Header format
  - Optional checksum
    - psuedo header + UDP header + data
    - pseudo header = protocol number + source IP address and destination IP address + UDP length field

From IP header

From UDP header

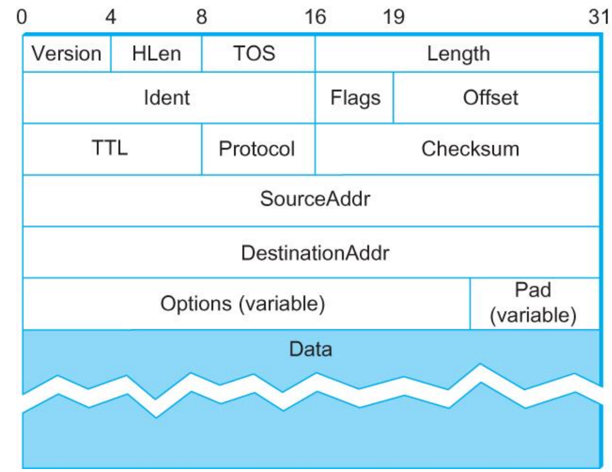
Exercise 1

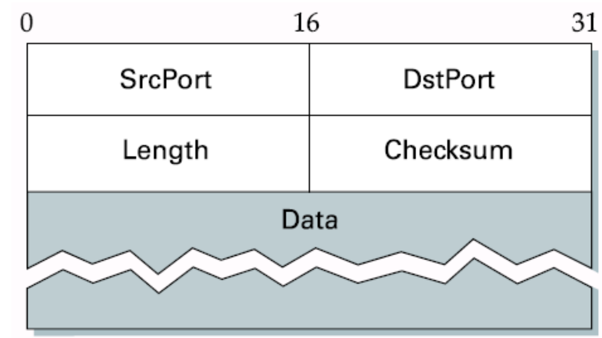


- Q1: How many UDP ports are there?
- Q2: How big are UDP headers?
- Q3: How much data does a UDP datagram can carry?

Exercise 2

- What are these two packets?
- Give fields and field values for the two packets?






```
>>> hexdump(datagram)
WARNING: No IP underlayer to compute checksum. Leaving null.
0000  30 39 D4 31 00 15 00 00 48 65 6C 6C 6F 2C 20 57  09.1....Hello, W
0010  6F 72 6C 64 21                                   orld!
>>> hexdump(packet)
0000  45 00 00 29 00 01 00 00 40 11 88 A3 C0 A8 38 67  E..)....@.....8g
0010  C0 A8 38 68 30 39 D4 31 00 15 C8 0C 48 65 6C 6C  ..8h09.1....Hell
0020  6F 2C 20 57 6F 72 6C 64 21                       o, World!
>>>
```

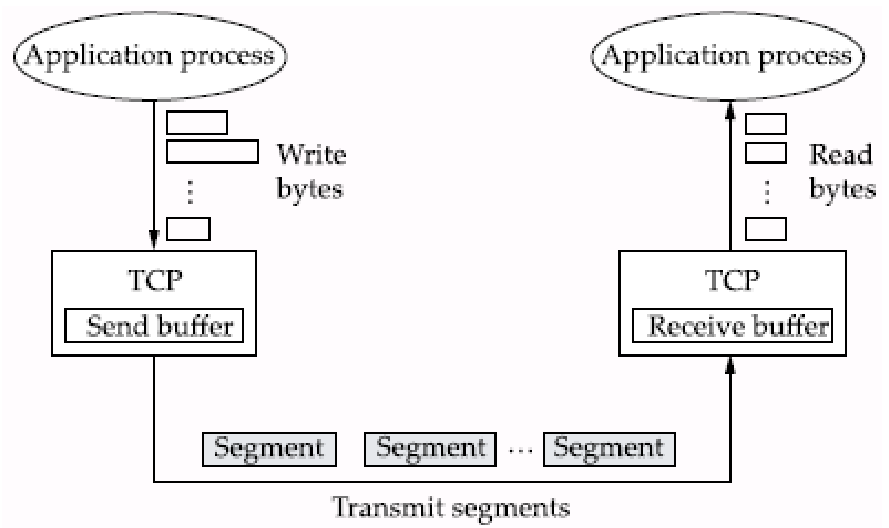
Transmission Control Protocol (TCP)

- Connection-oriented
- Byte-stream
  - applications writes bytes
  - TCP sends segments
  - applications reads bytes
- Full duplex
- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network

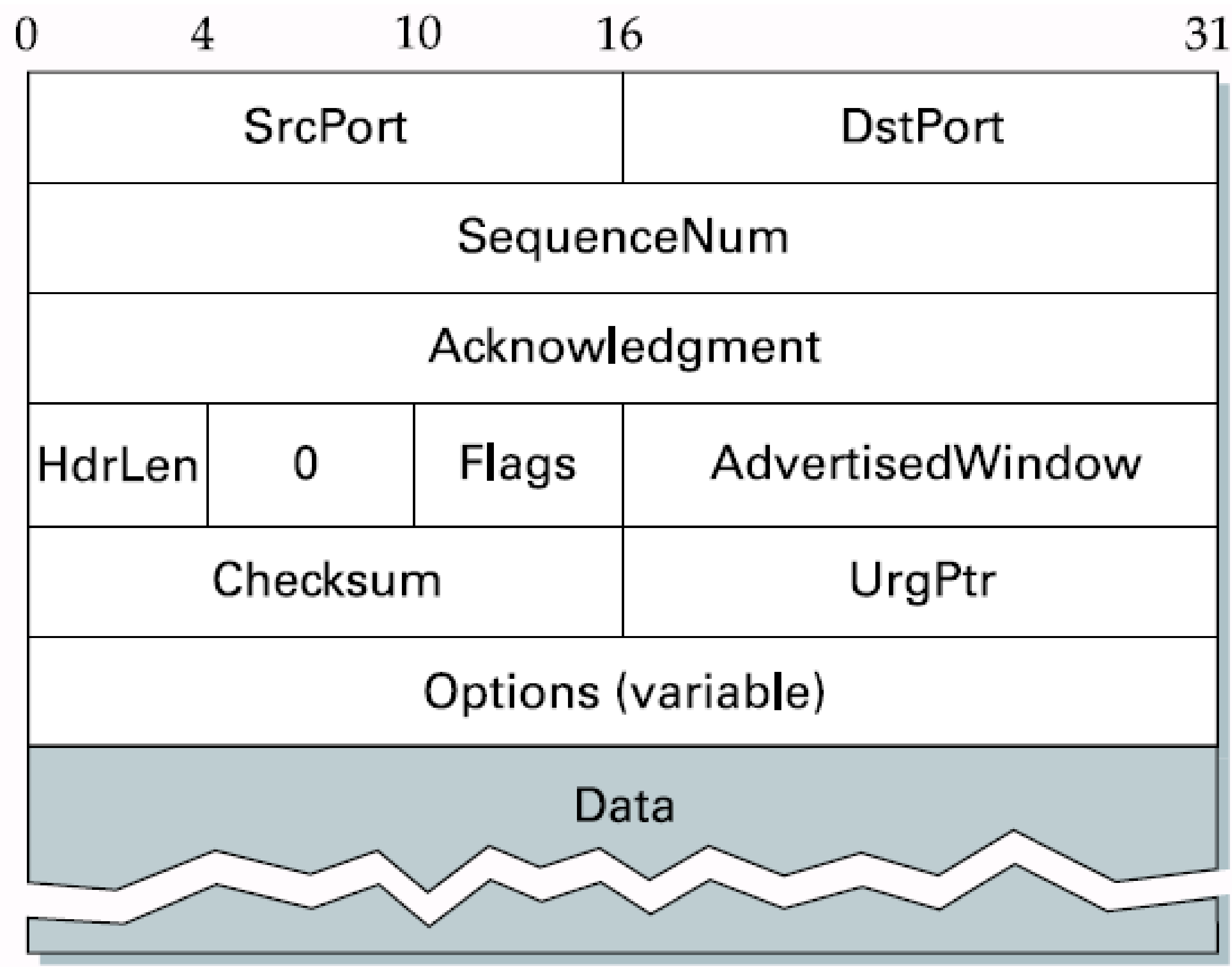
Data Link Versus Transport

- Potentially connects many different hosts
  - need explicit connection establishment and termination
- Potentially different RTT
  - need adaptive timeout mechanism
- Potentially long delay in network
  - need to be prepared for arrival of very old packets

- Potentially different capacity at destination
- need to accommodate different node capacity
- Potentially different network capacity
- need to be prepared for network congestion

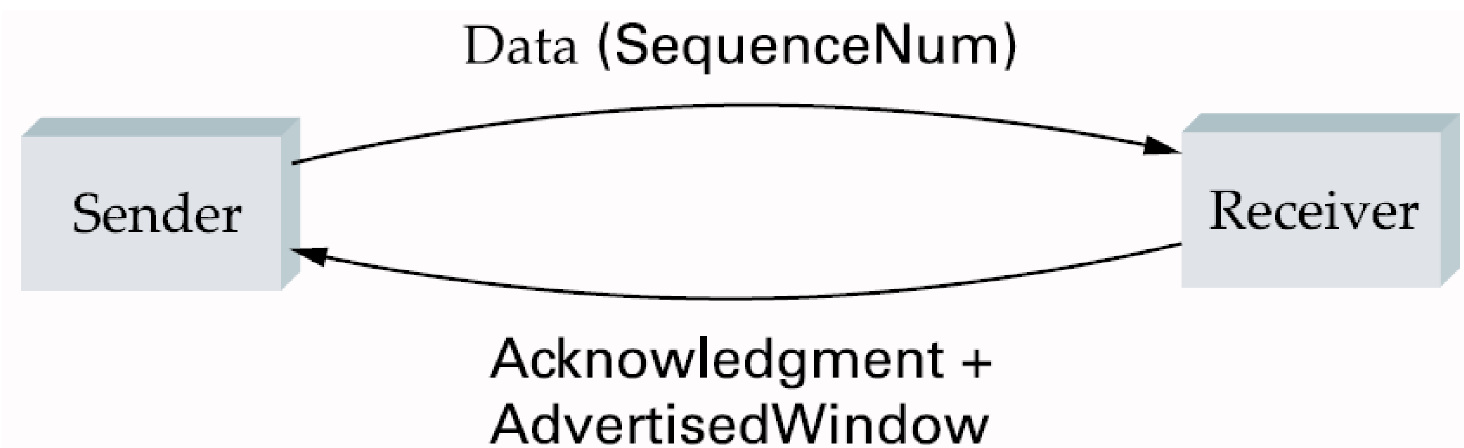


Segment Format (1)



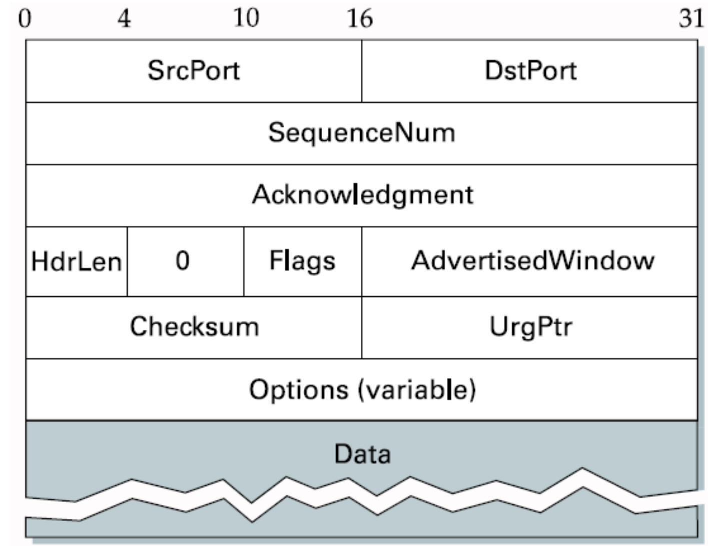
Segment Format (2)

- Each connection identified with 4-tuple:
  - (SrcPort, SrcIPAddr, DsrPort, DstIPAddr)
- Sliding window + flow control
  - acknowledgment, SequenceNum, AdvertisedWinow
- Flags
  - SYN, FIN, RESET, PUSH, URG, ACK
- Checksum
  - pseudo header + TCP header + data



Exercise 3

- What are these packets? What are the fields and their values?




```
>>> hexdump(packet1)
0000  45 00 00 28 00 01 00 00 40 06 64 A9 0A 01 01 03  E..(....@.d.....
0010  0A 01 01 22 C3 50 C3 51 00 00 00 64 00 00 00 64  ...".P.Q...d...d
0020  50 02 20 00 F2 51 00 00                          P. ..Q..
>>> hexdump(packet2)
0000  45 00 00 28 00 01 00 00 40 06 64 A9 0A 01 01 03  E..(....@.d.....
0010  0A 01 01 22 C3 50 C3 51 00 00 00 67 00 00 01 4F  ...".P.Q...g...O
0020  50 10 20 00 F1 55 00 00                          P. ..U..
>>> hexdump(packet3)
0000  45 00 00 3A 00 01 00 00 40 06 64 97 0A 01 01 03  E..:....@.d.....
0010  0A 01 01 22 C3 50 C3 51 00 00 00 67 00 00 01 4F  ...".P.Q...g...O
0020  50 18 20 00 12 9B 00 00 47 45 54 20 2F 20 48 54  P. .....GET / HT
0030  54 50 2F 31 2E 31 0D 0A 0D 0A                    TP/1.1....
```

Sequence and Acknowledgement Numbers (1)

- Host A sends a file of 500,000 bytes over a TCP connection with Maximum Segment Size (MSS) as 1,000 bytes to host B
  - How many segments? 500,000/1,000 = 500
  - Sequence number assignments
    - Sequence number of 1st segment? 0
    - Sequence number of 2nd segment? 1,000
    - Sequence number of 3rd segment? 2,000
    - ……

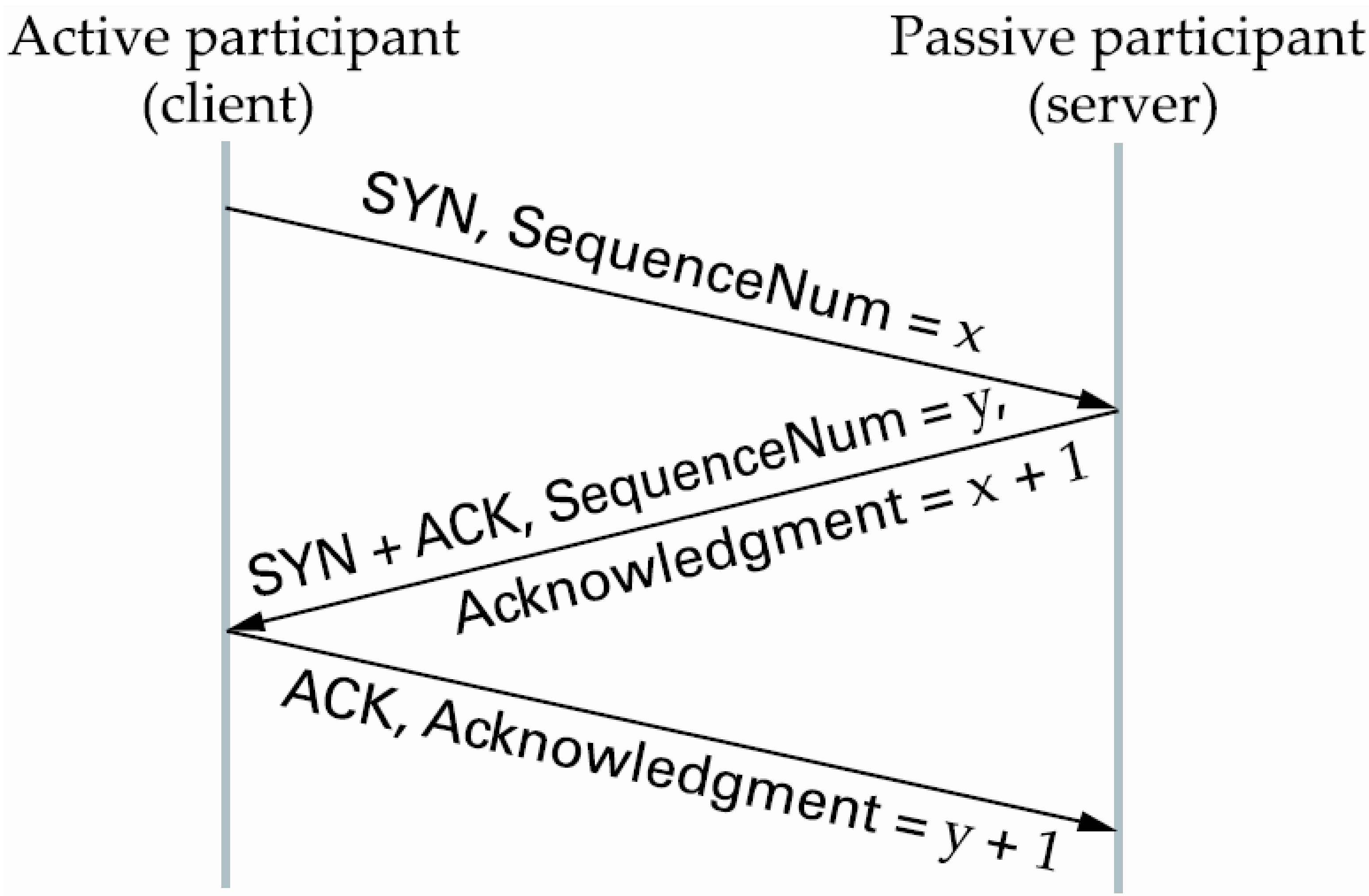
Sequence and Acknowledgement Numbers (2)

- Scenario 1
  - Host B received all bytes numbered 0 to 1,999 from host A
  - What would host B put in the acknowledgement number field of the segment it sends to A?
    - 2,000: the sequence number of the next byte host B is expecting
- Scenario 2
  - Host B received two segments containing bytes from 0-999, and 2,000-2,999, respectively?
  - What would host B put in the acknowledgement number field of the segment it sends to A?
    - 1000: TCP only acknowledges bytes up to the first missing byte in the stream, and it is the next byte host B is expecting
- Scenario 3
  - Host B received 1st segment containing bytes from 0-999. Somehow, next it received 3rd segment containing bytes from 2,000-2,999.
  - What does host B in this case that the segments arrive out of order?
    - TCP does not specify how to deal with this situation. Hence, it is up to the implementation.
      - Option 1: Host B immediately discards out-of-order segment → simple receiver design
      - Option 2: Host B keeps the out-of-order segment and waits for missing bytes to fill in the gaps → more efficient on bandwidth utilization → taken in practice

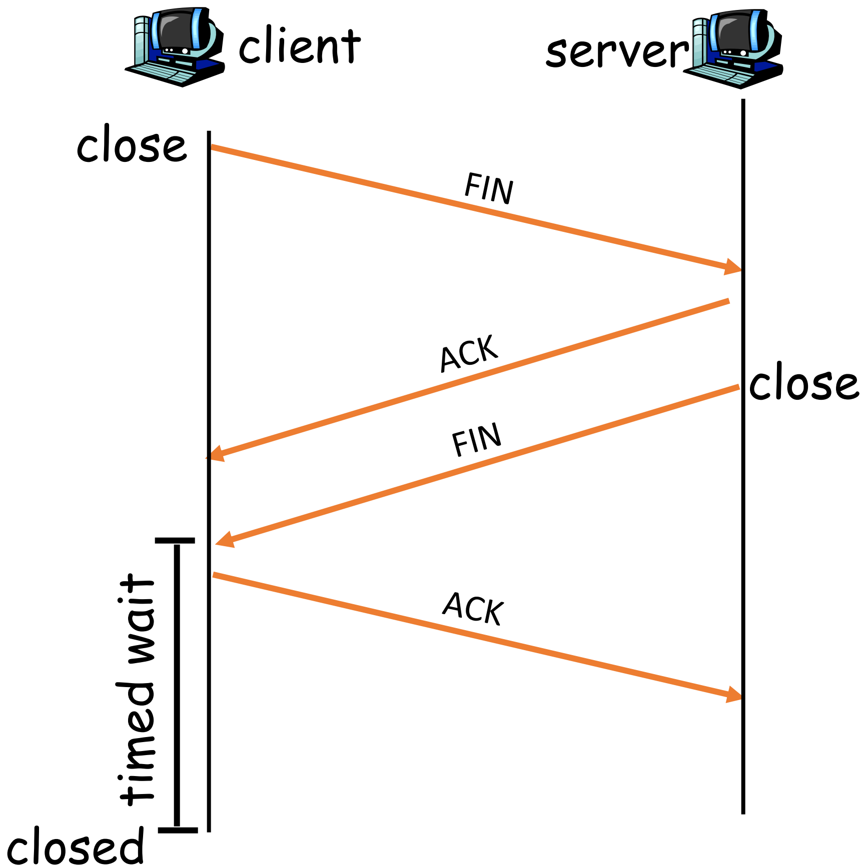
TCP is Connection-Oriented

- Keep track of states of receiver and sender
  - Connection Establishment
  - Connection Termination
  - TCP finite state machine and state transition

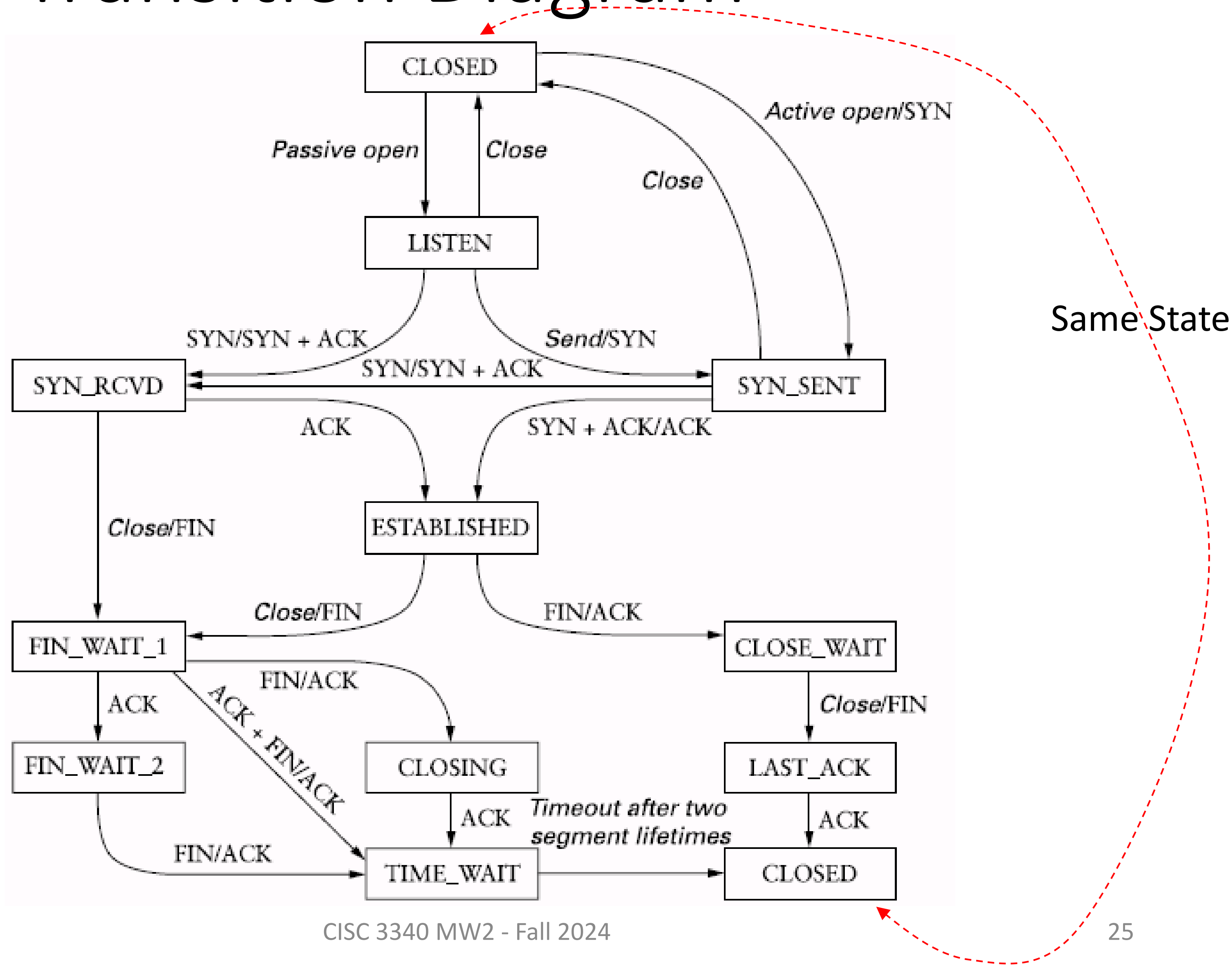
Connection Establishment



Connection Termination

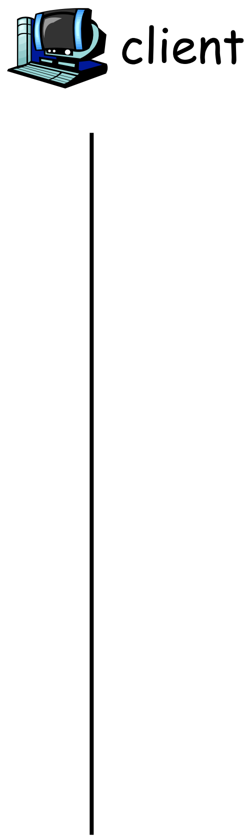


State Transition Diagram

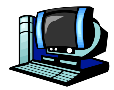


Connection Establishment and State Transition

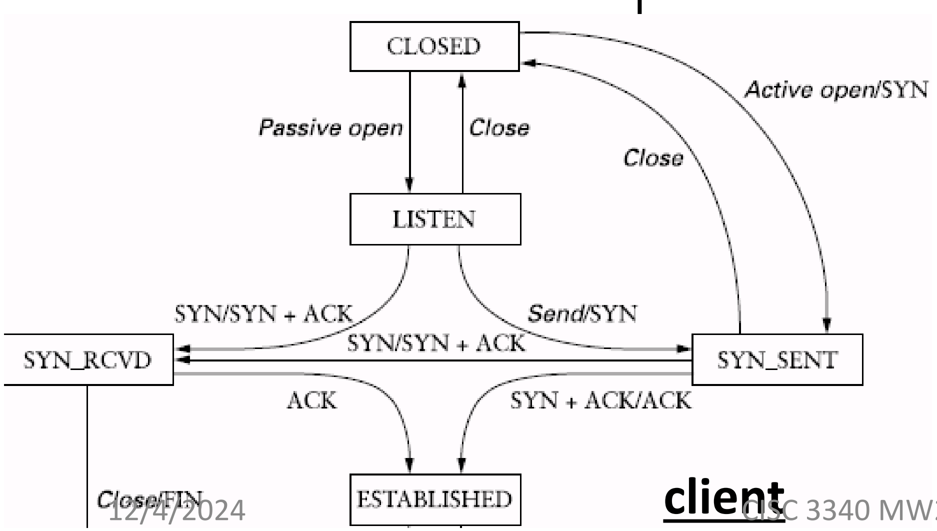
Connection Establishment and State Transition



Connection Establishment and State Transition

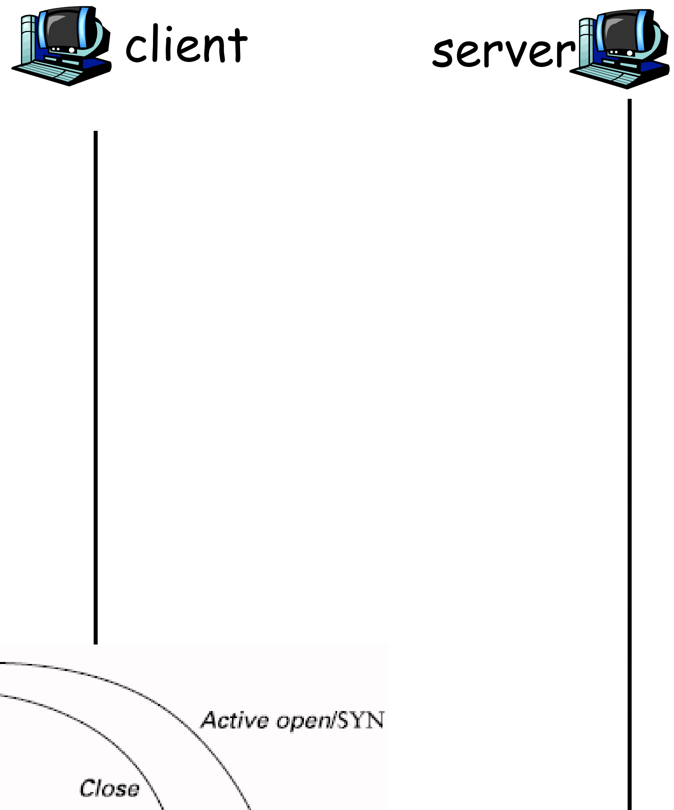


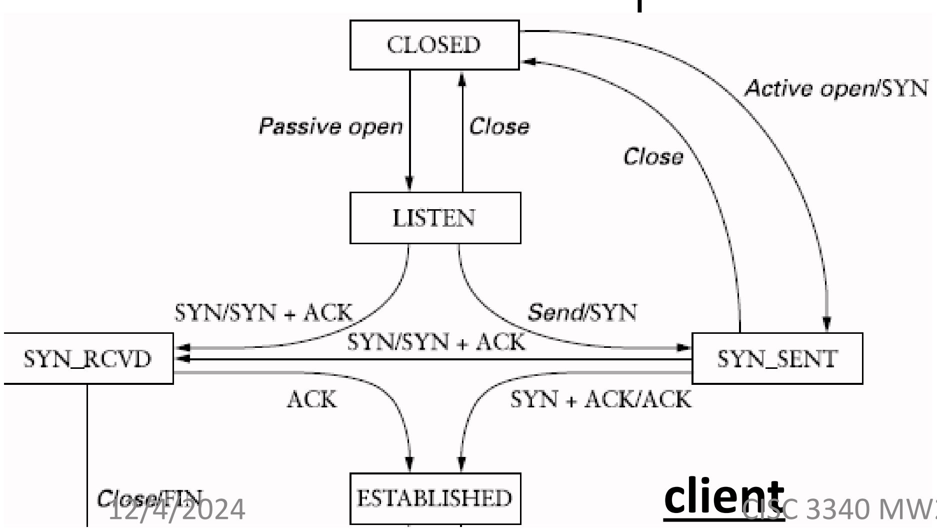
client



client

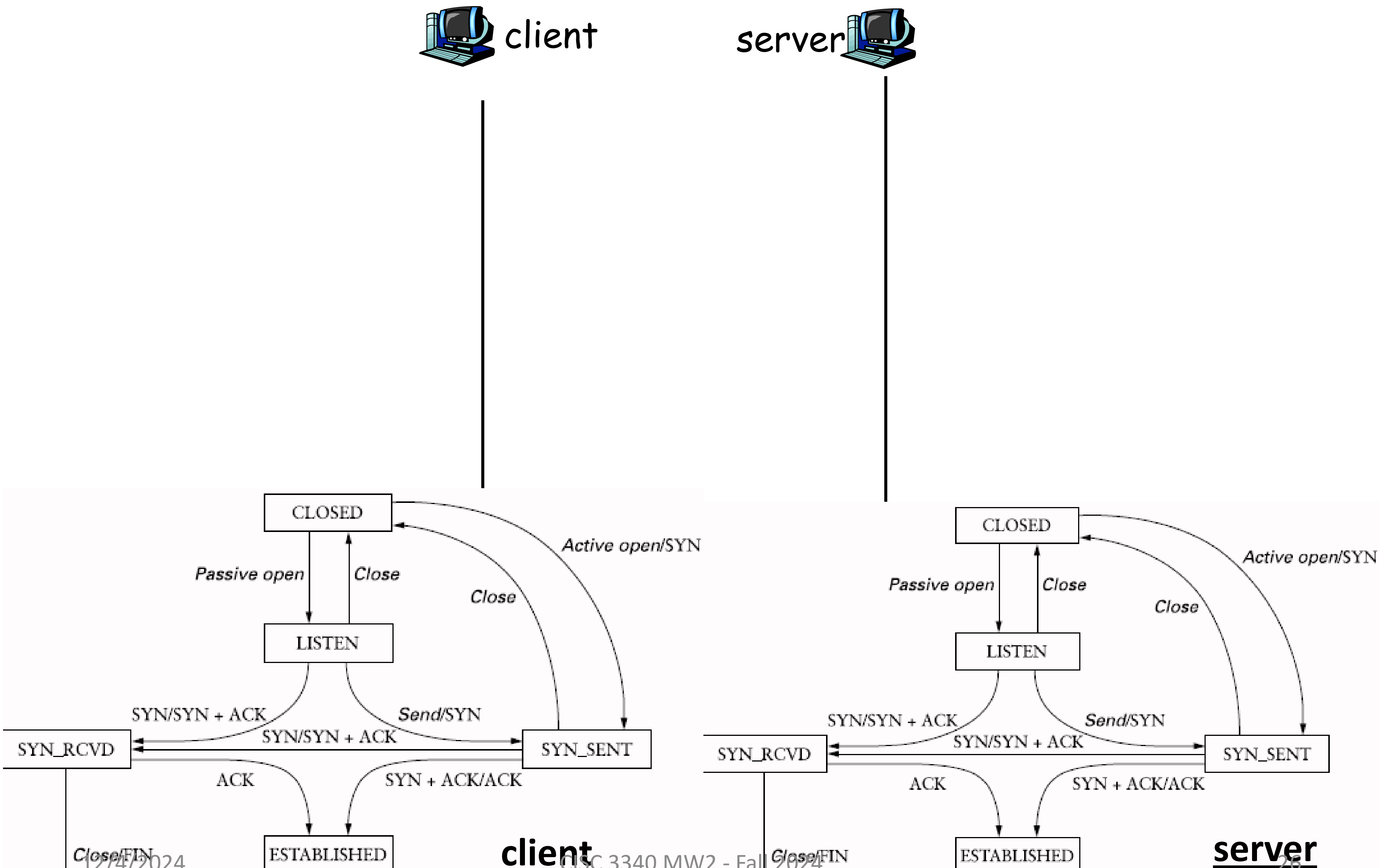
Connection Establishment and State Transition





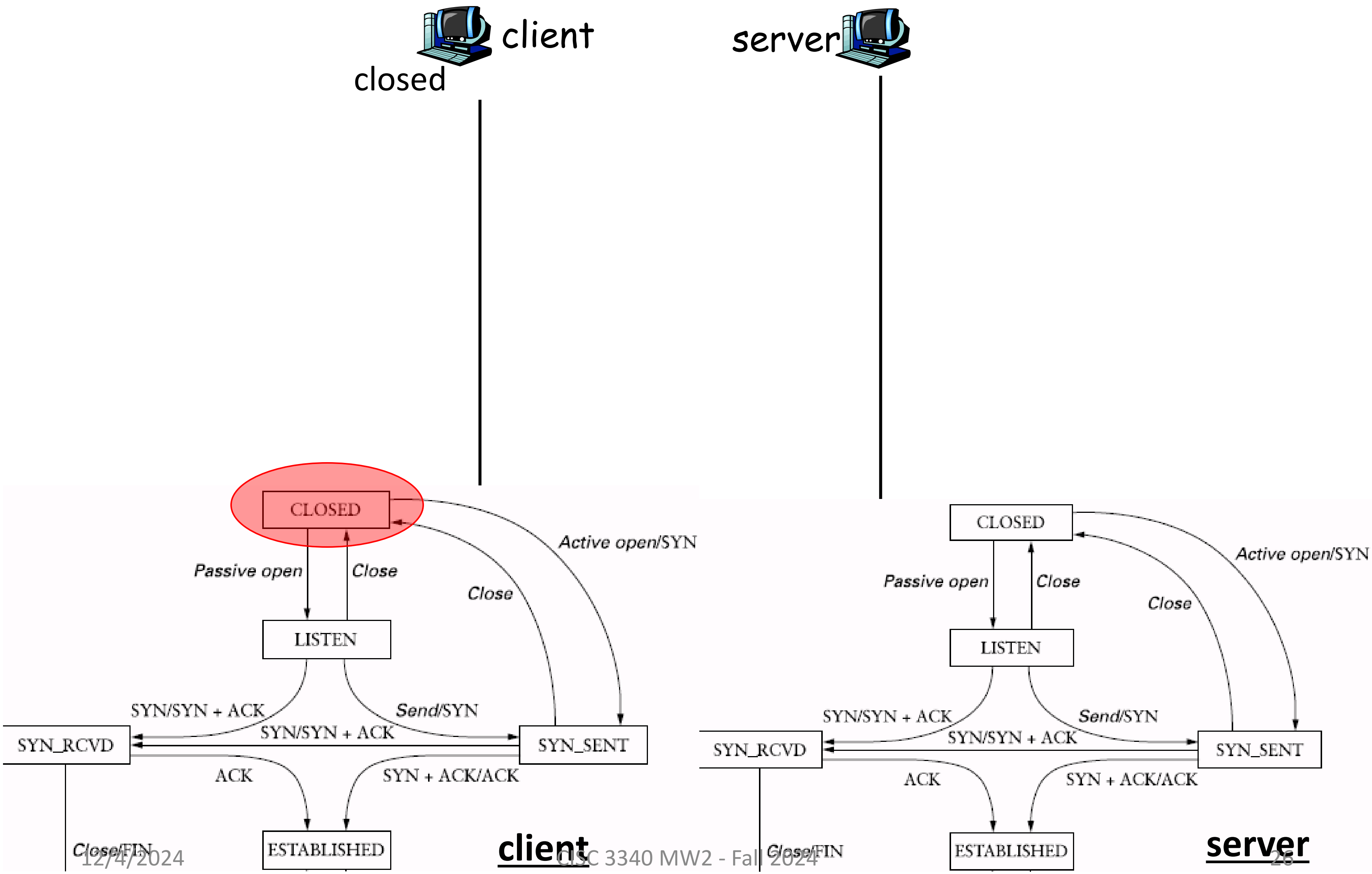
client

Connection Establishment and State Transition

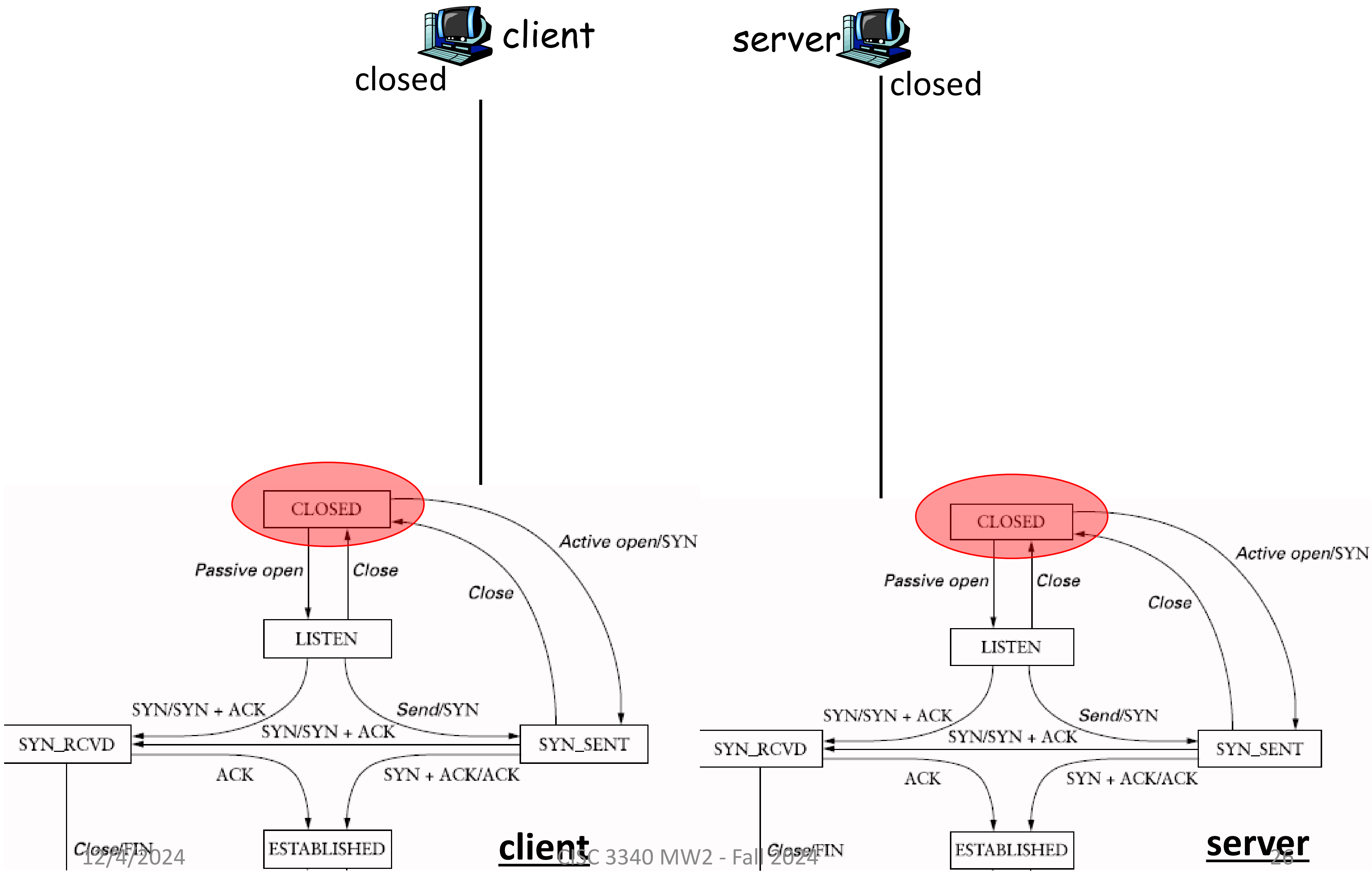


client **server**

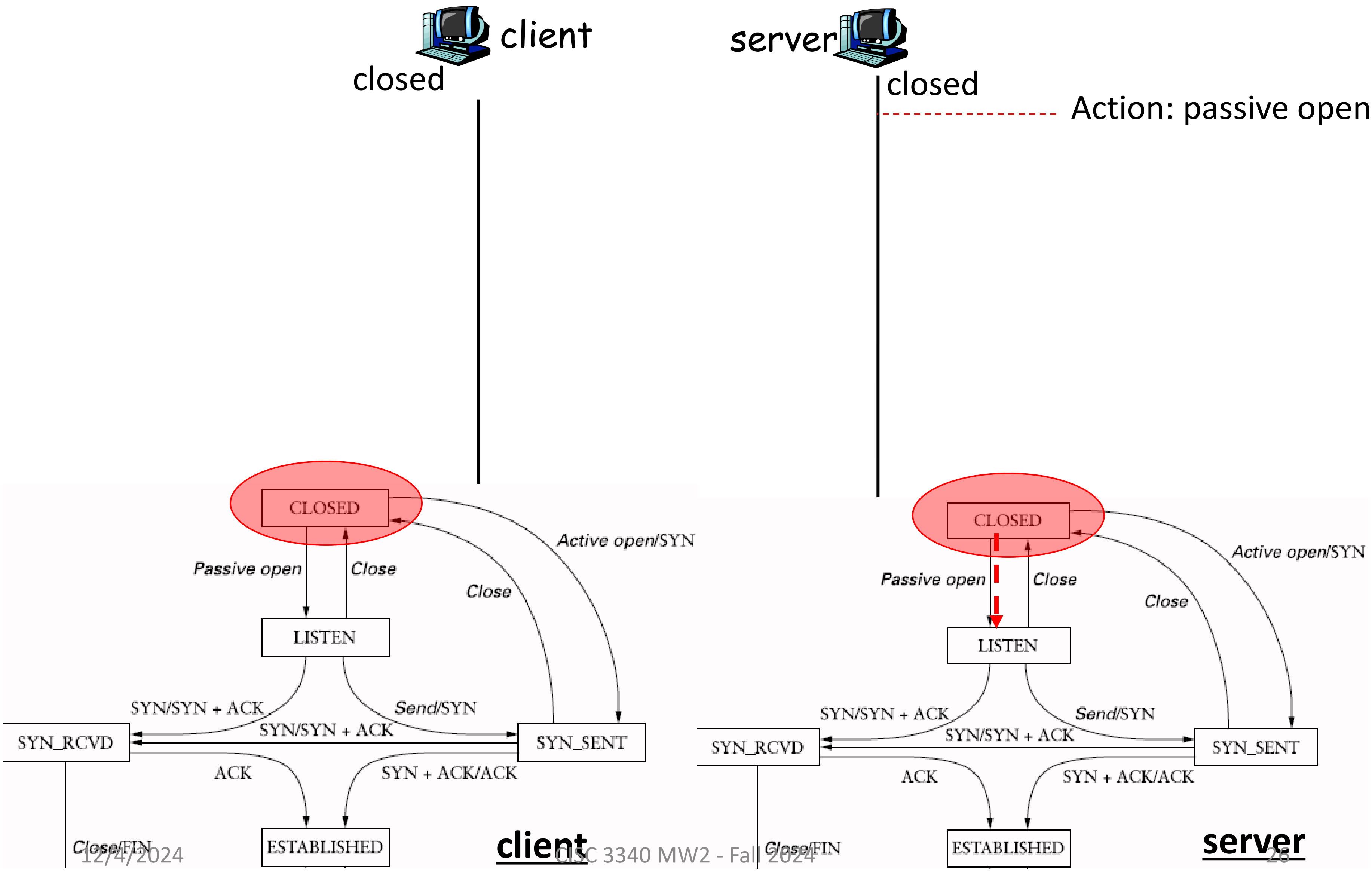
Connection Establishment and State Transition



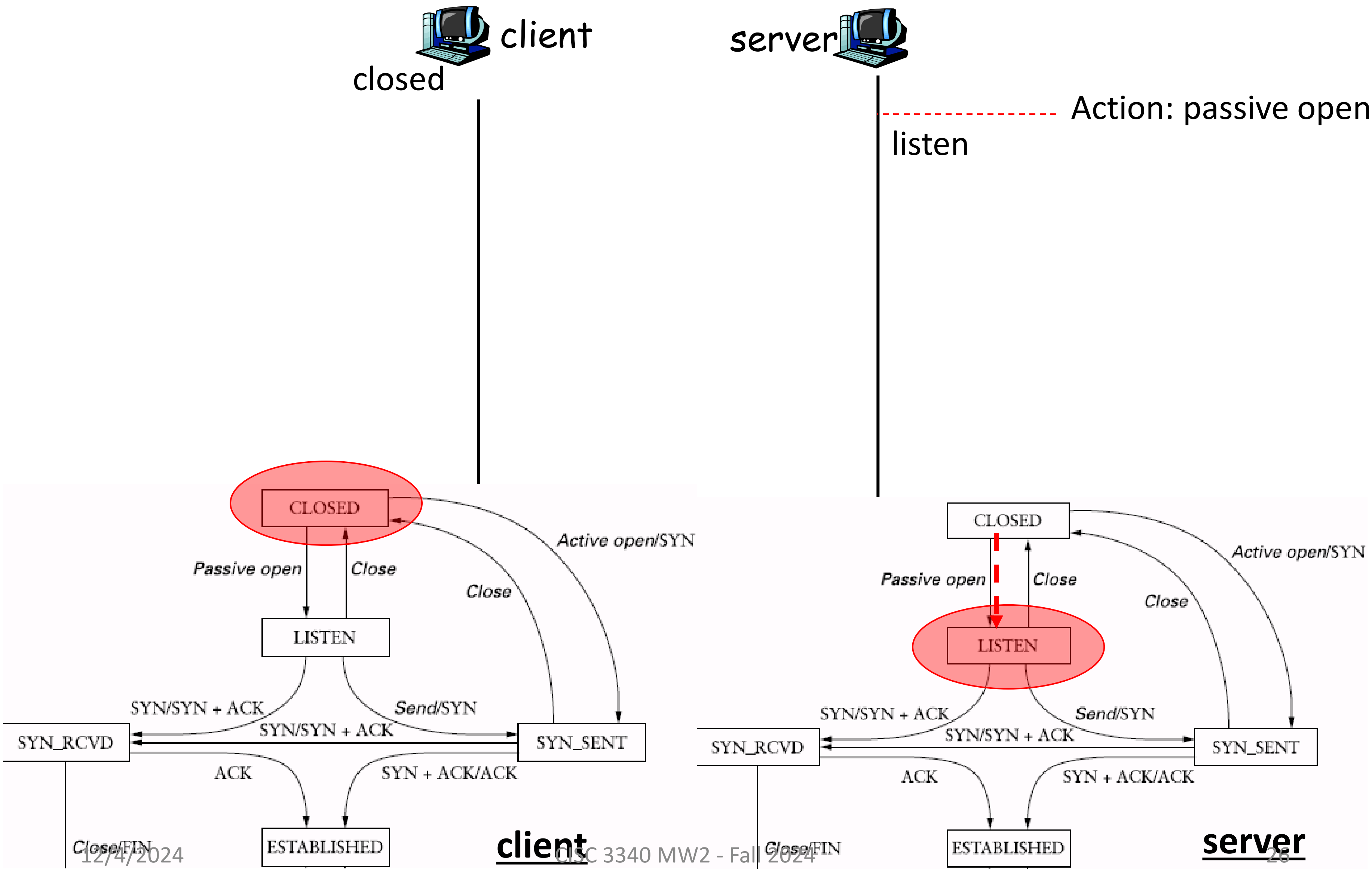
Connection Establishment and State Transition



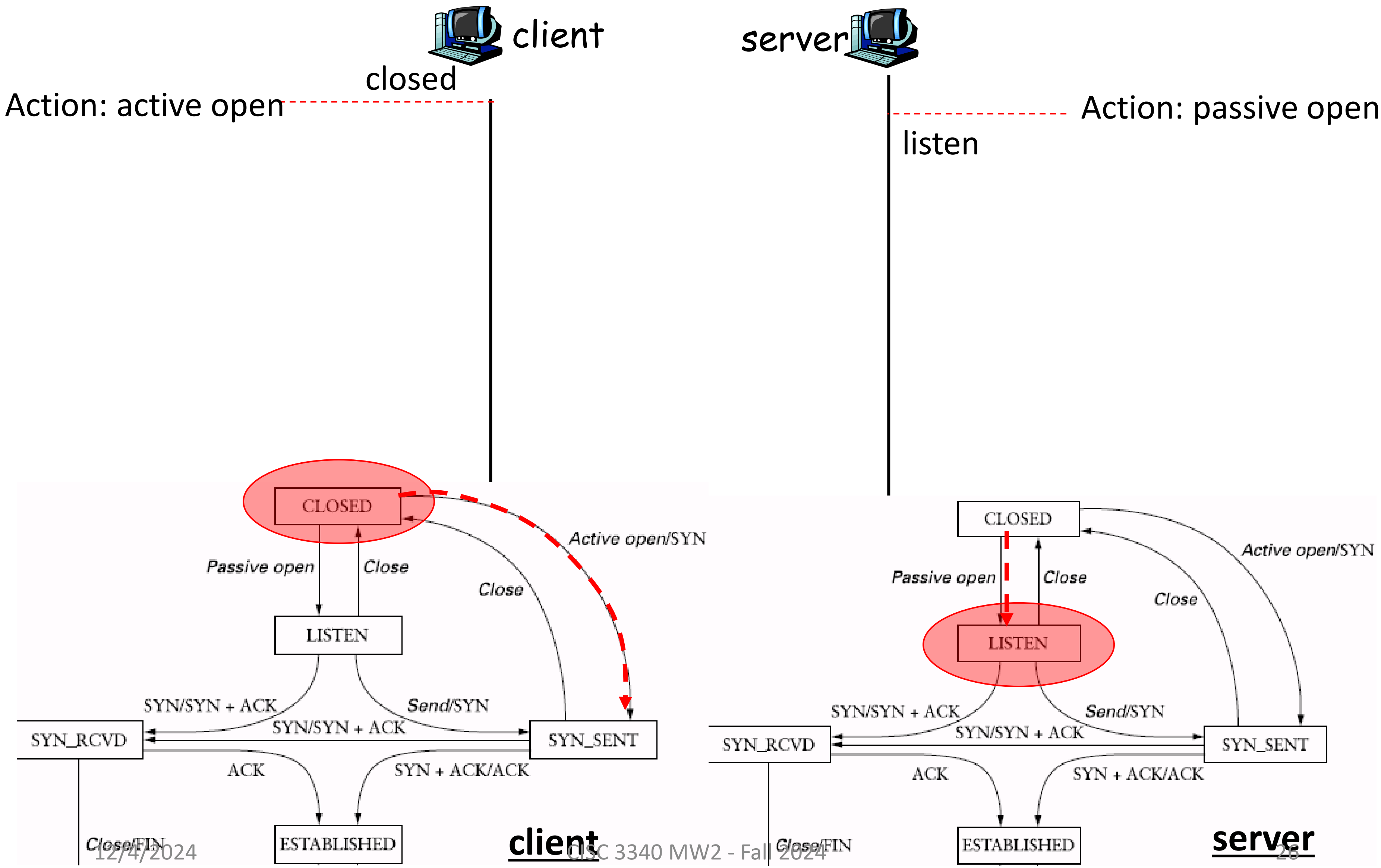
Connection Establishment and State Transition



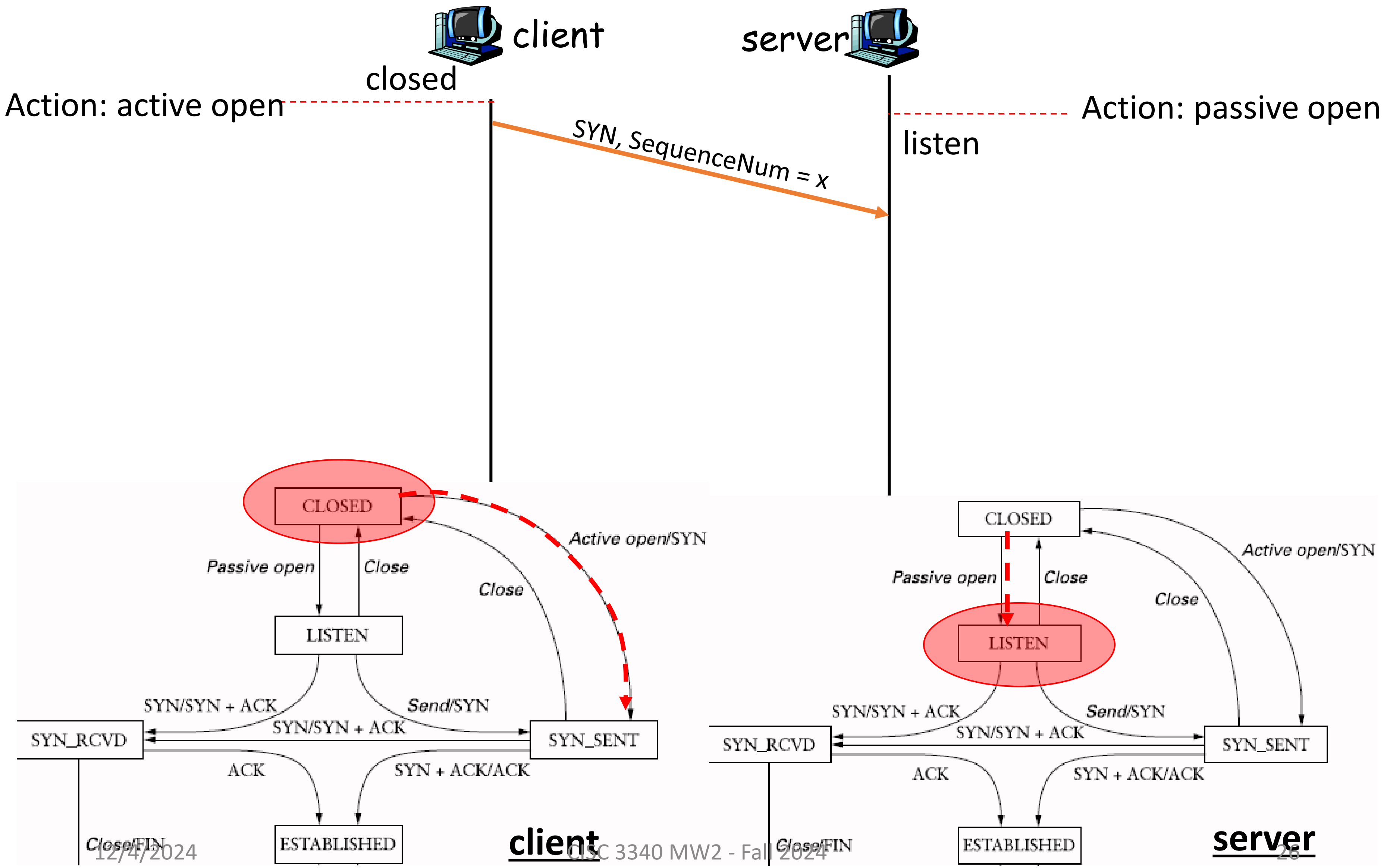
Connection Establishment and State Transition



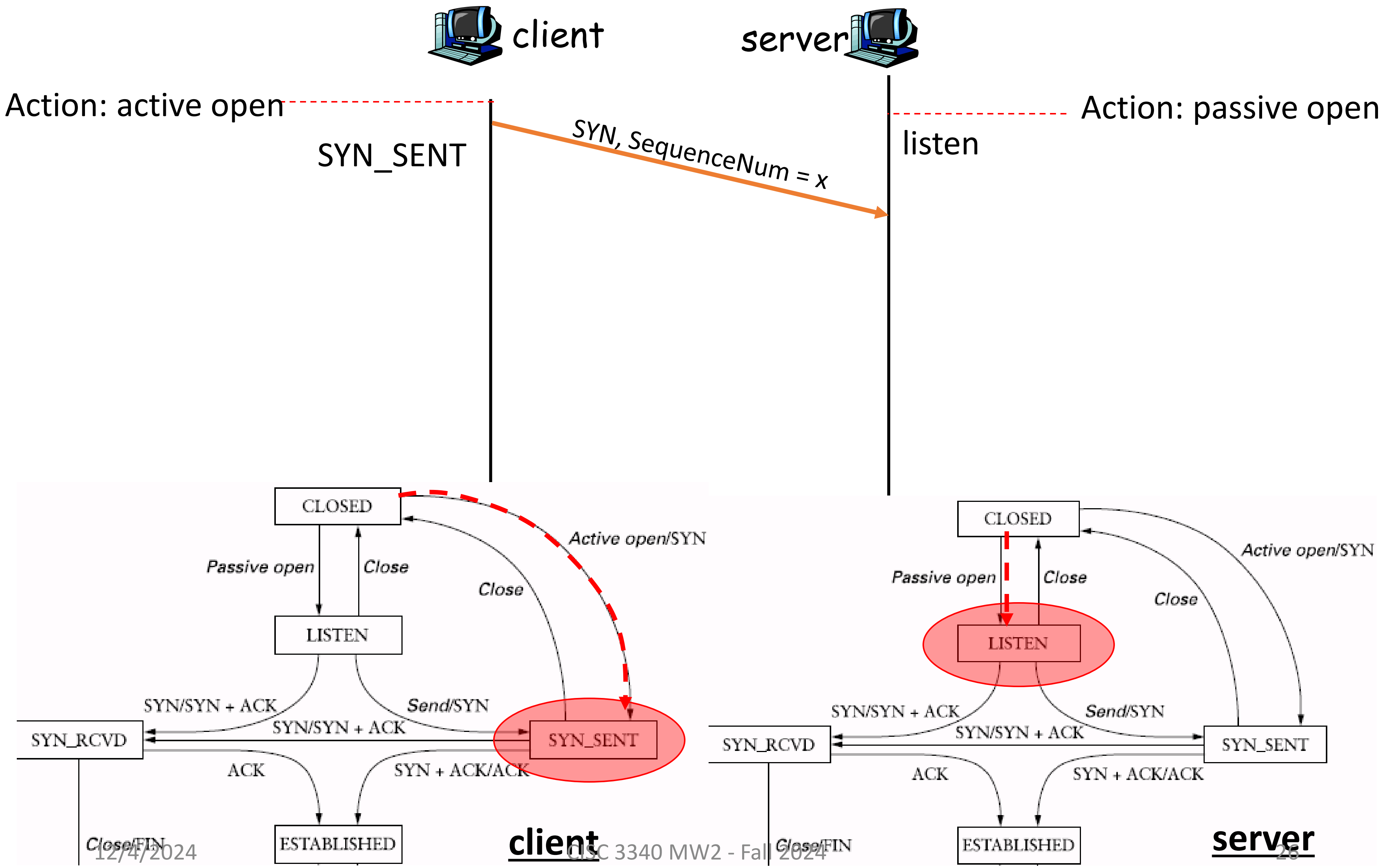
Connection Establishment and State Transition



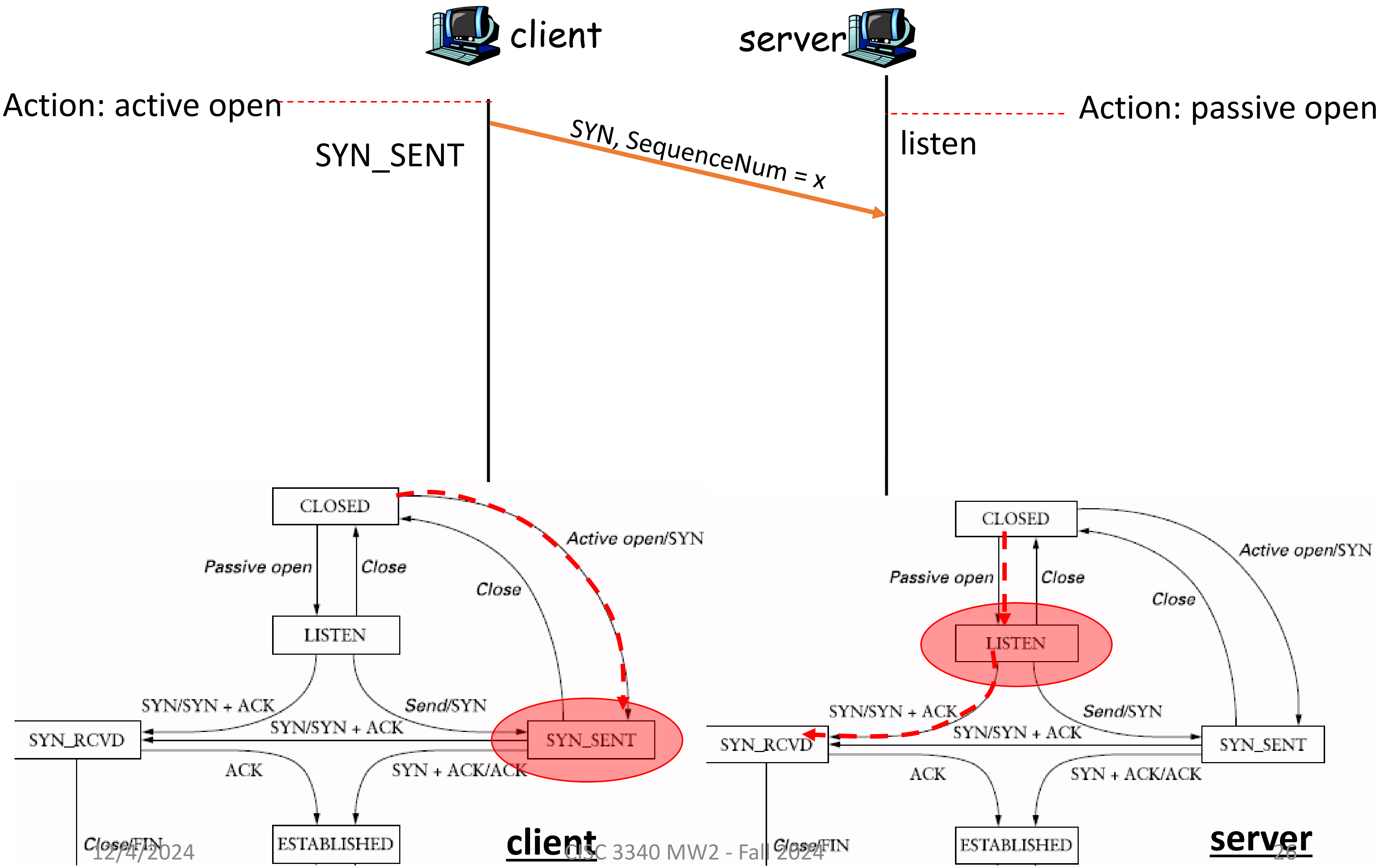
Connection Establishment and State Transition



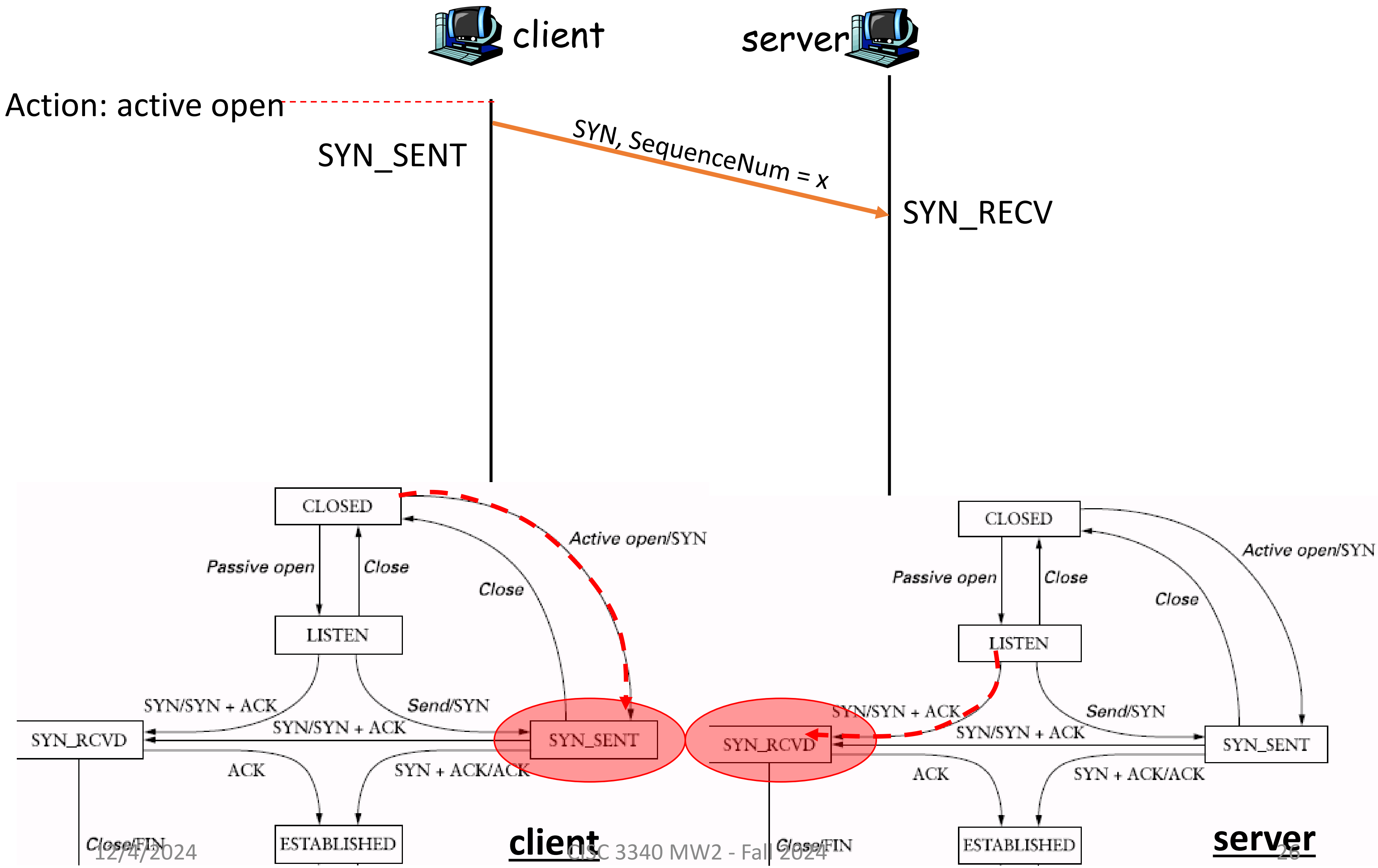
Connection Establishment and State Transition



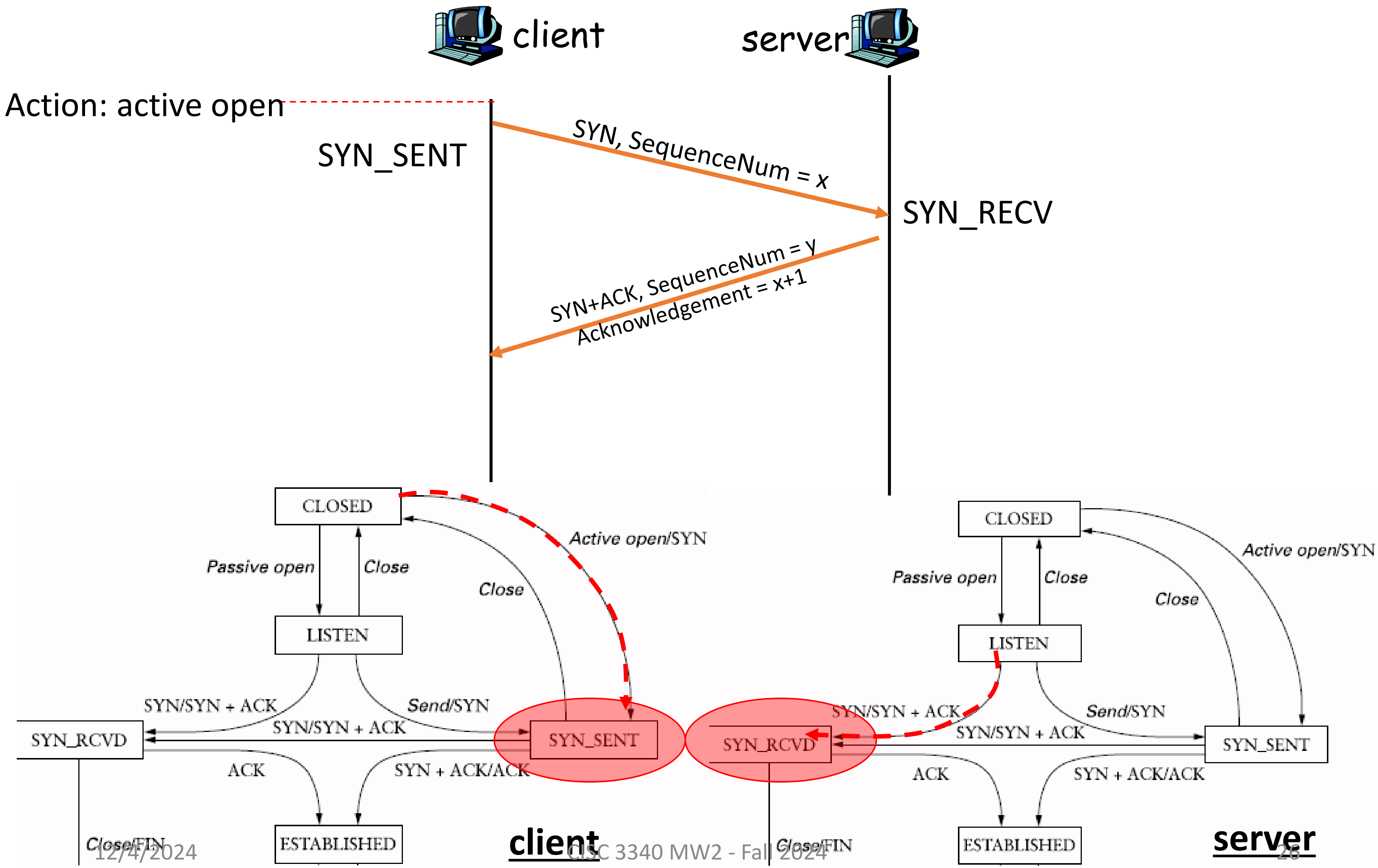
Connection Establishment and State Transition



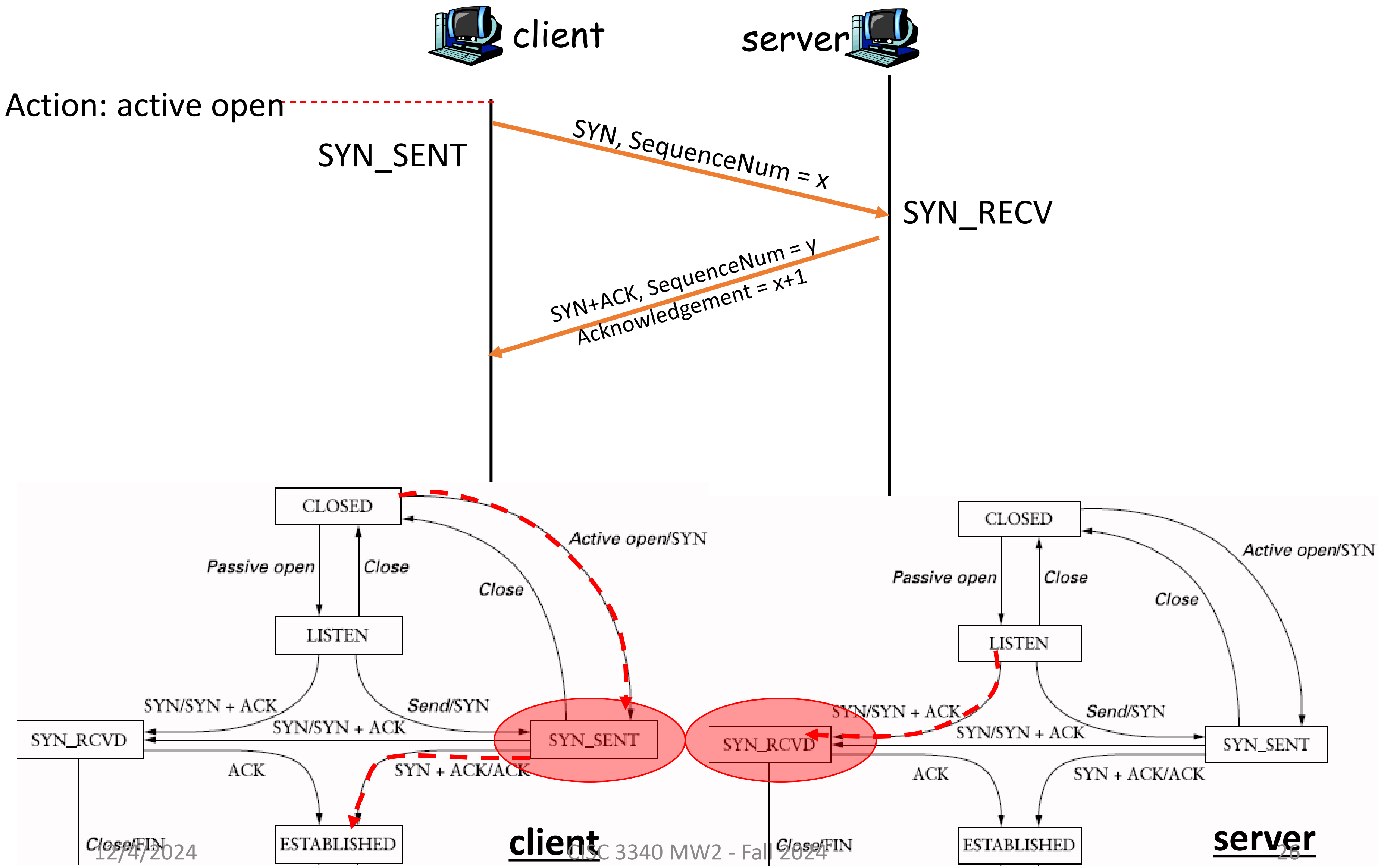
Connection Establishment and State Transition



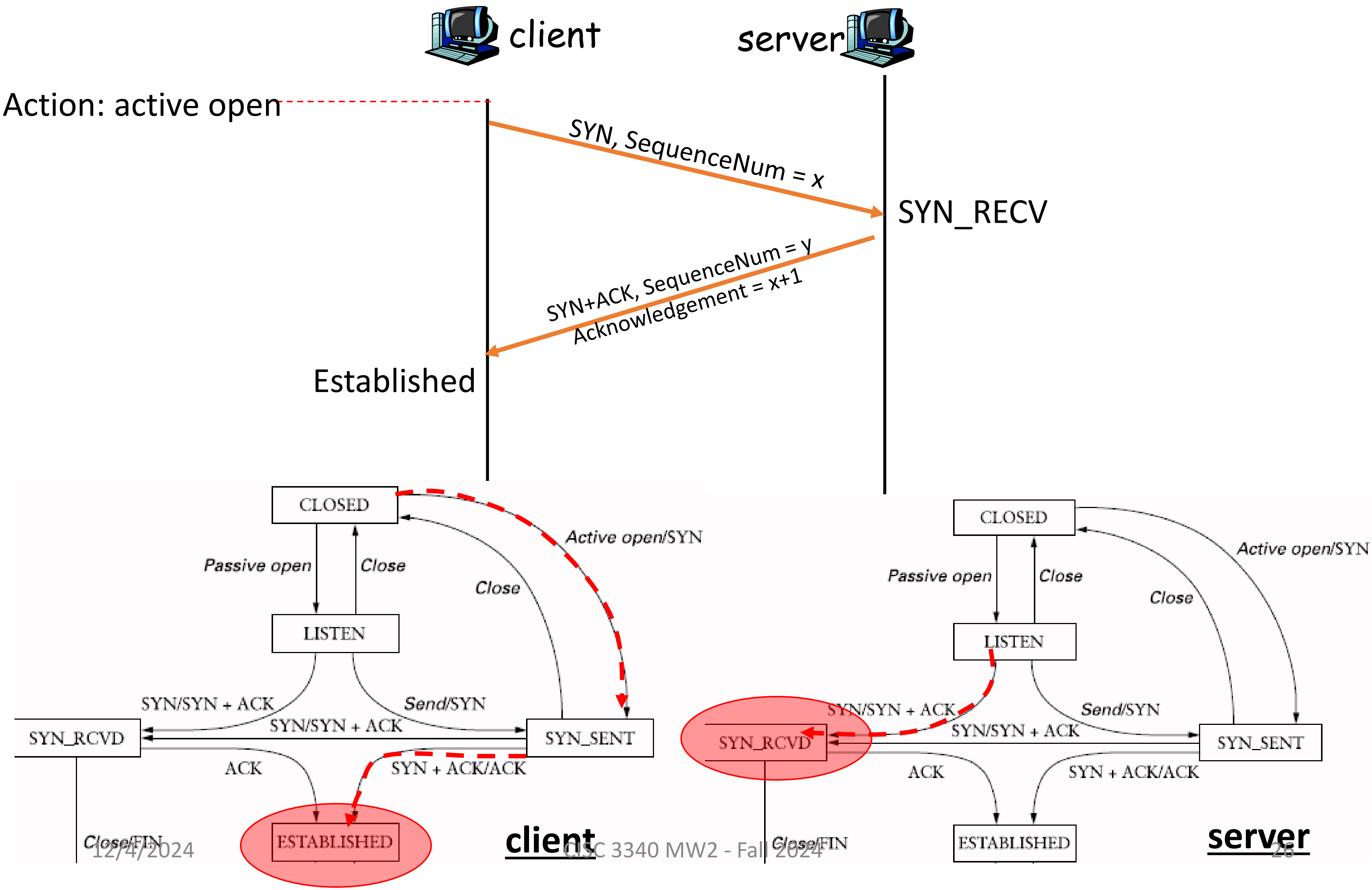
Connection Establishment and State Transition



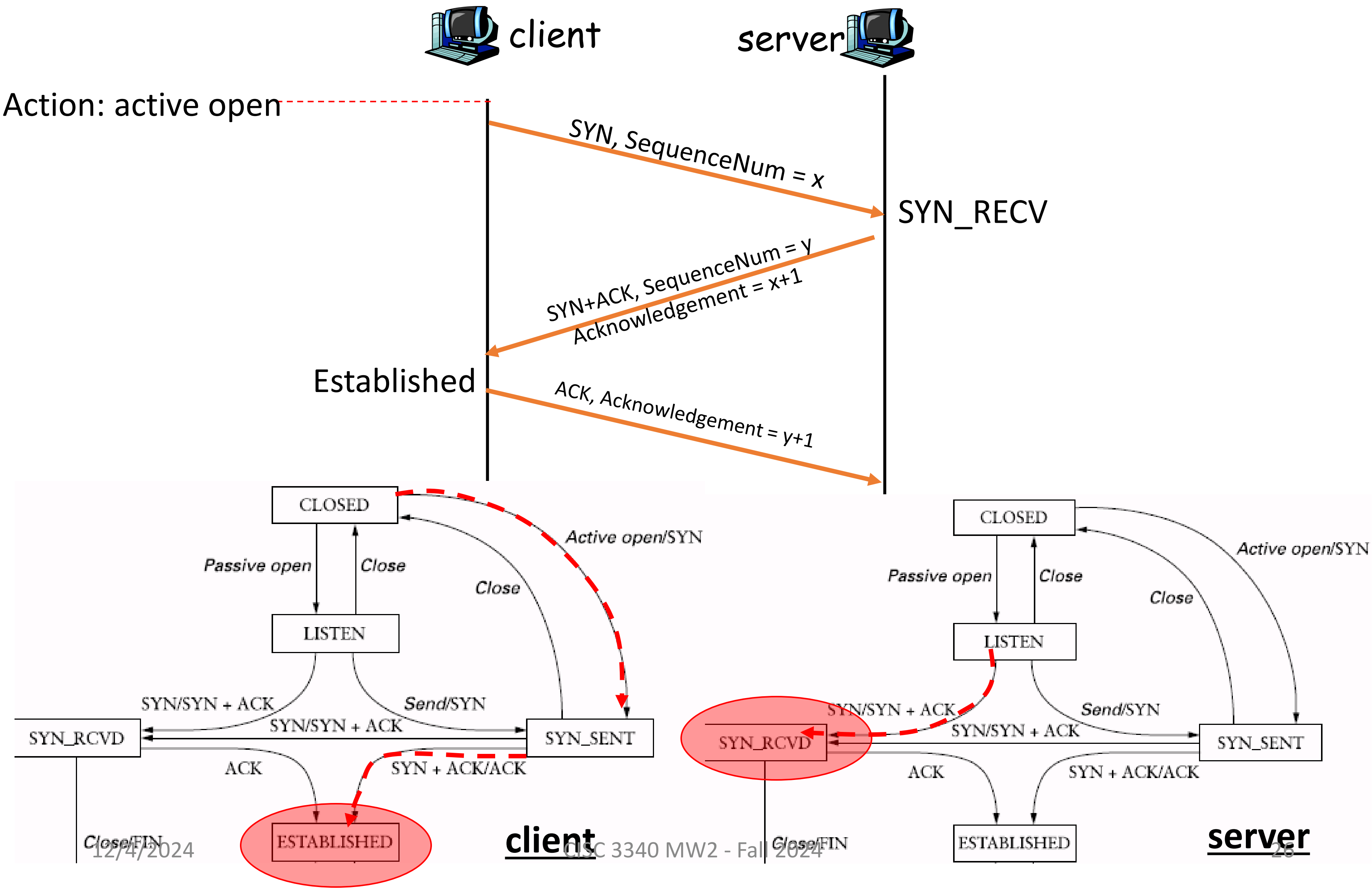
Connection Establishment and State Transition



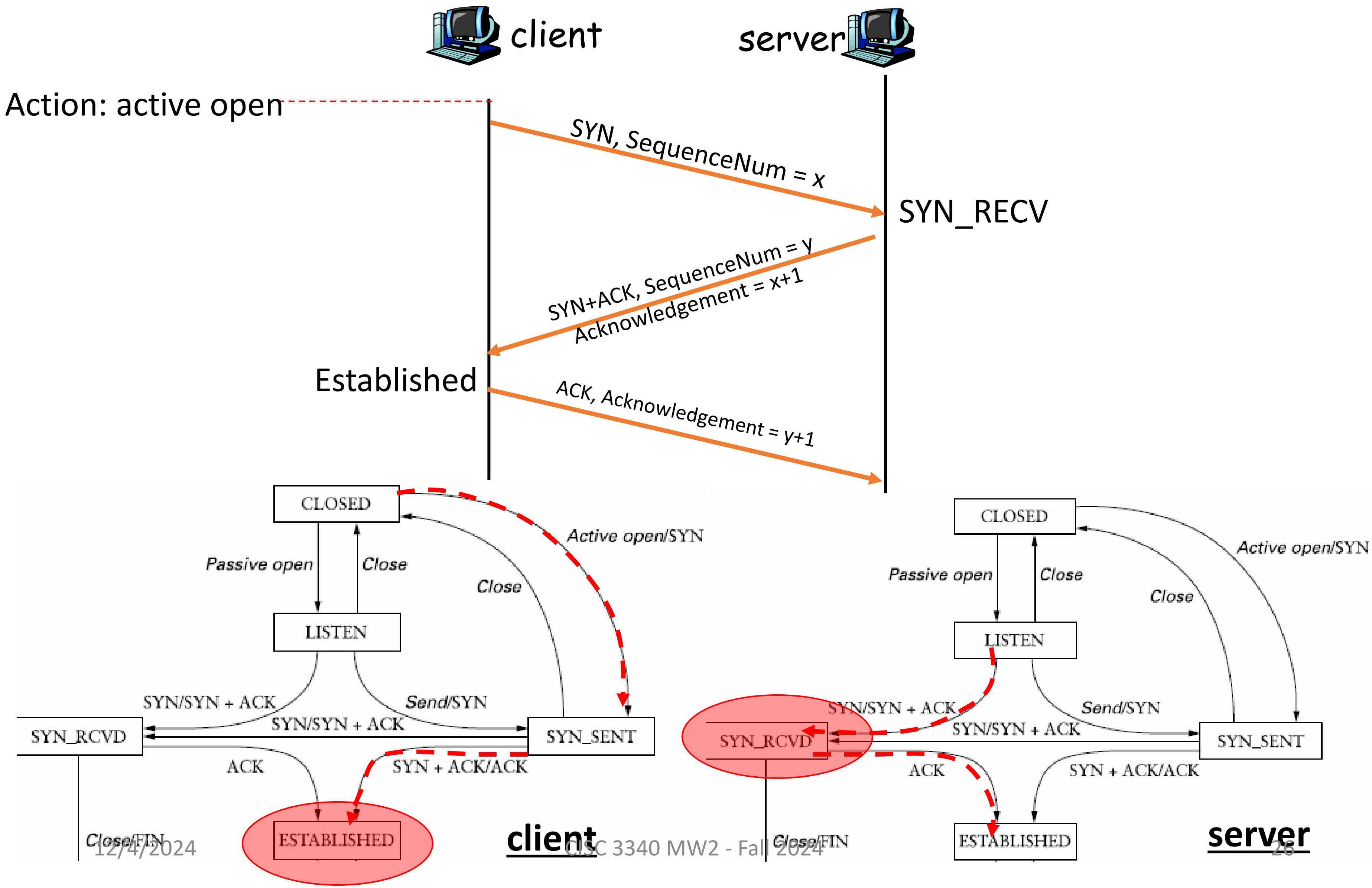
Connection Establishment and State Transition



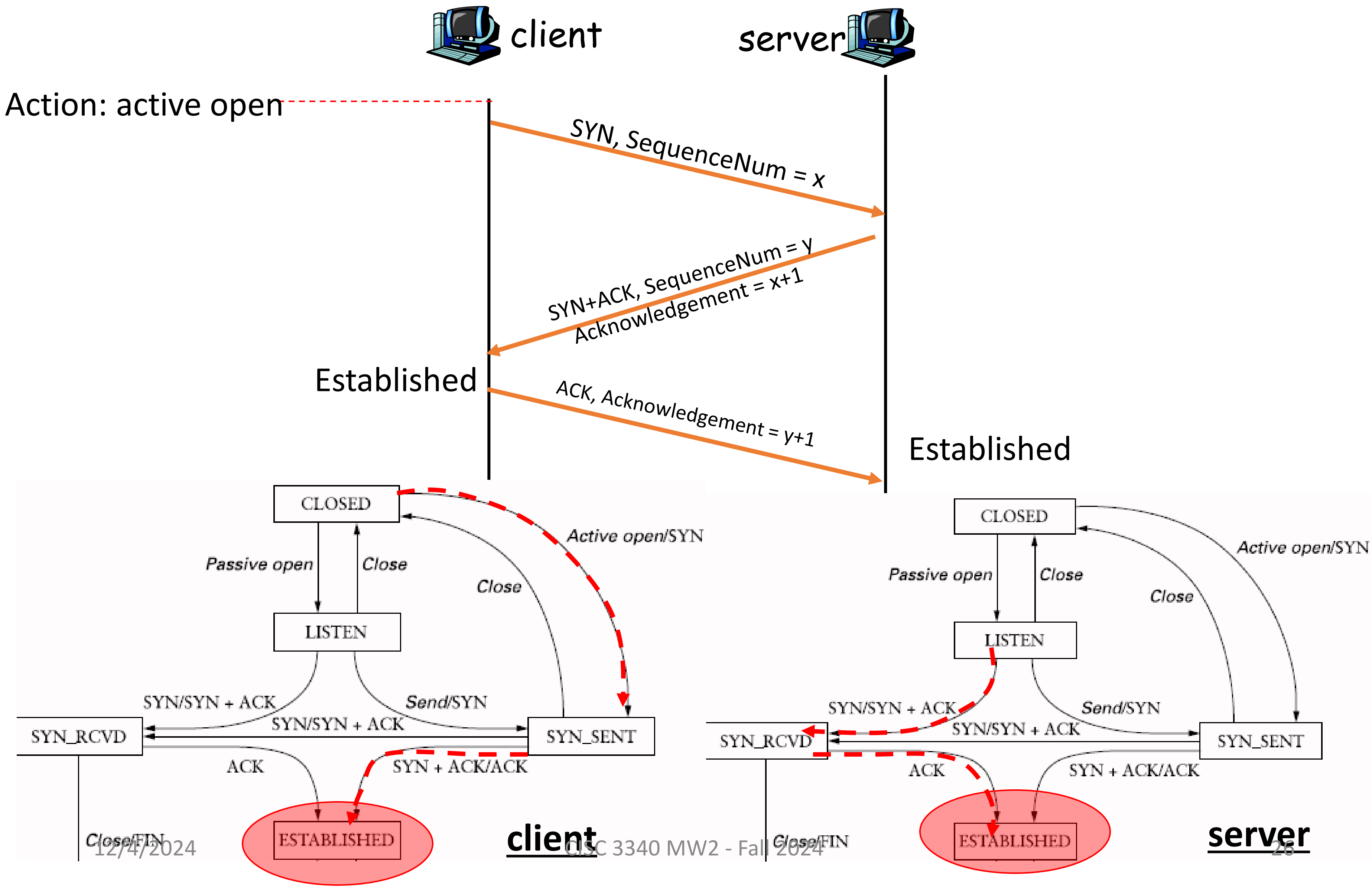
Connection Establishment and State Transition



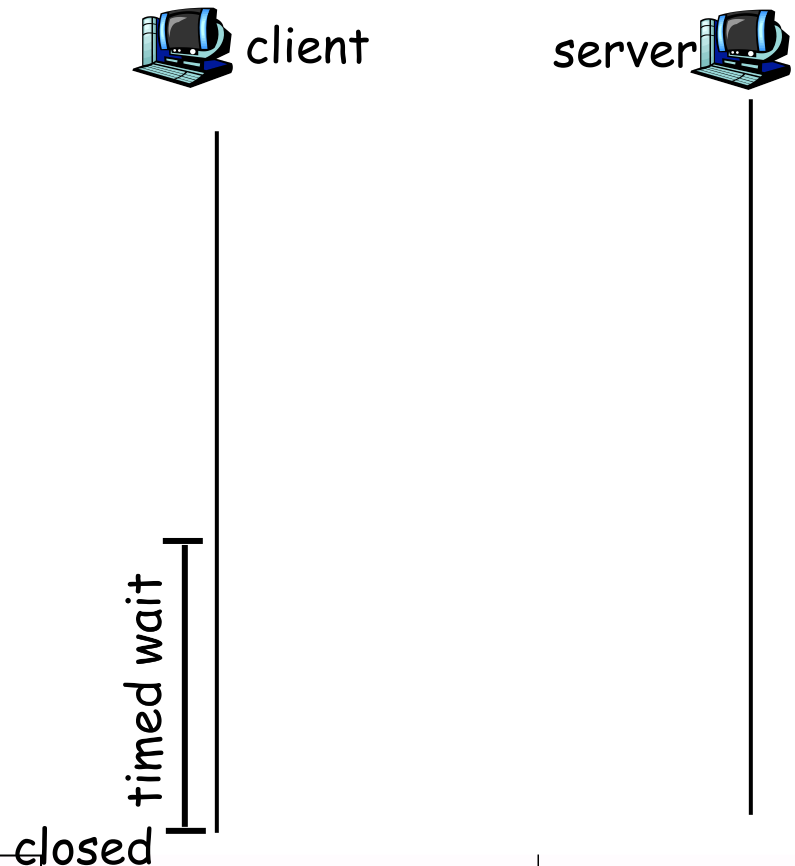
Connection Establishment and State Transition



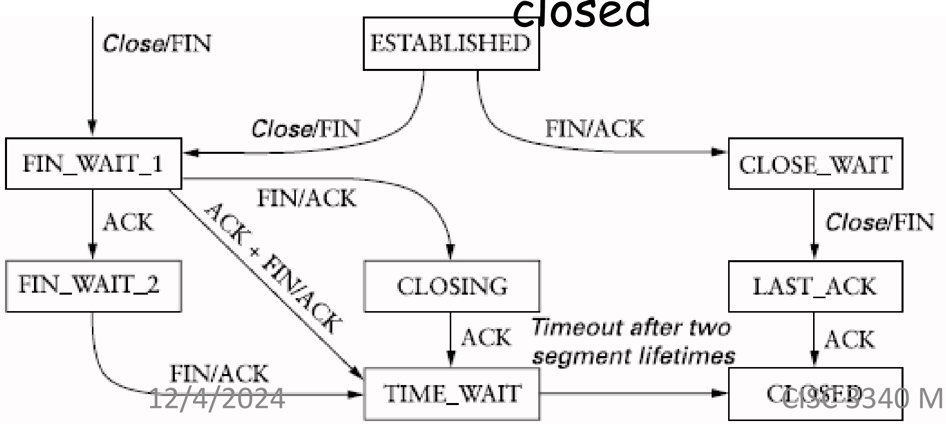
Connection Establishment and State Transition

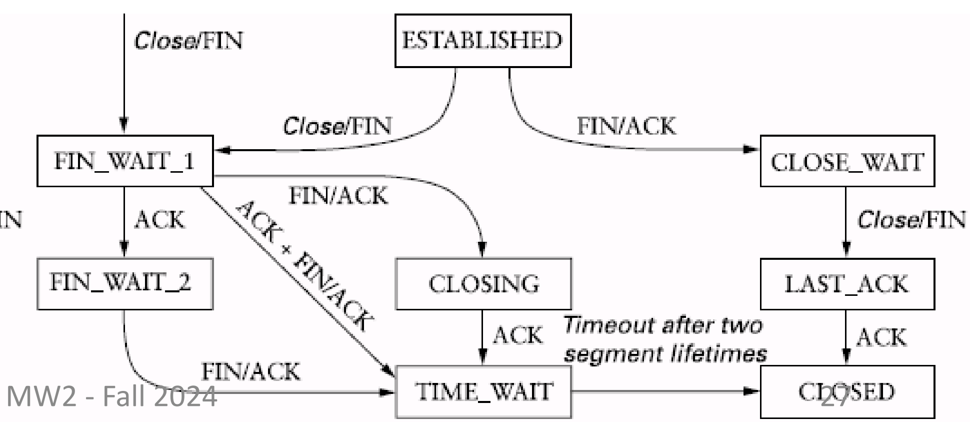


Connection Termination and State Transition (1)

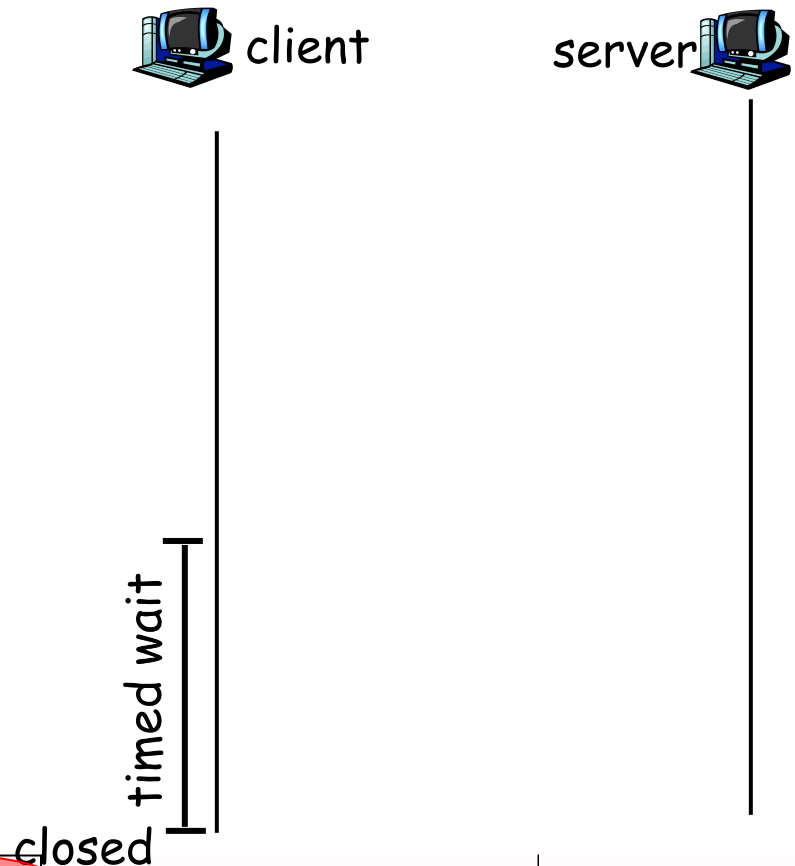


Client closes first

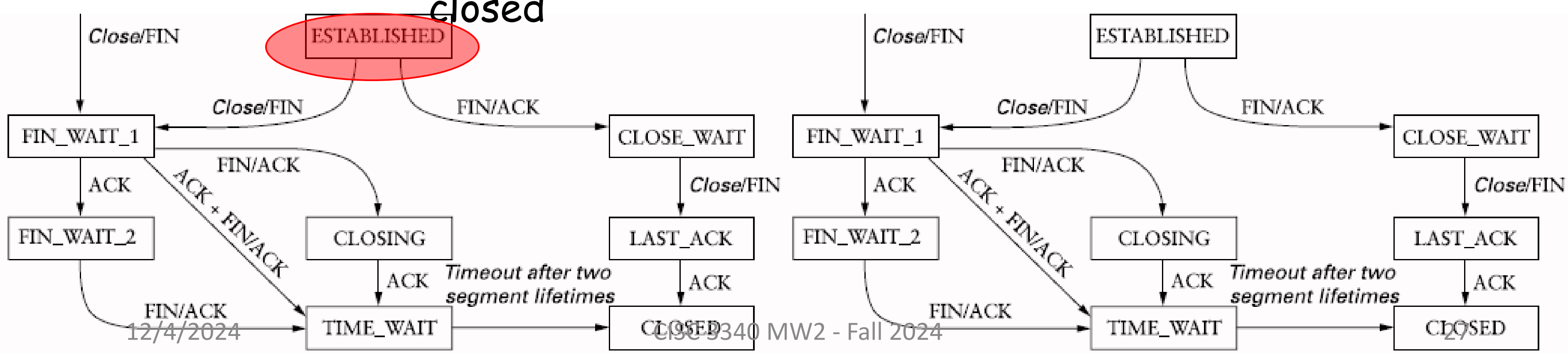




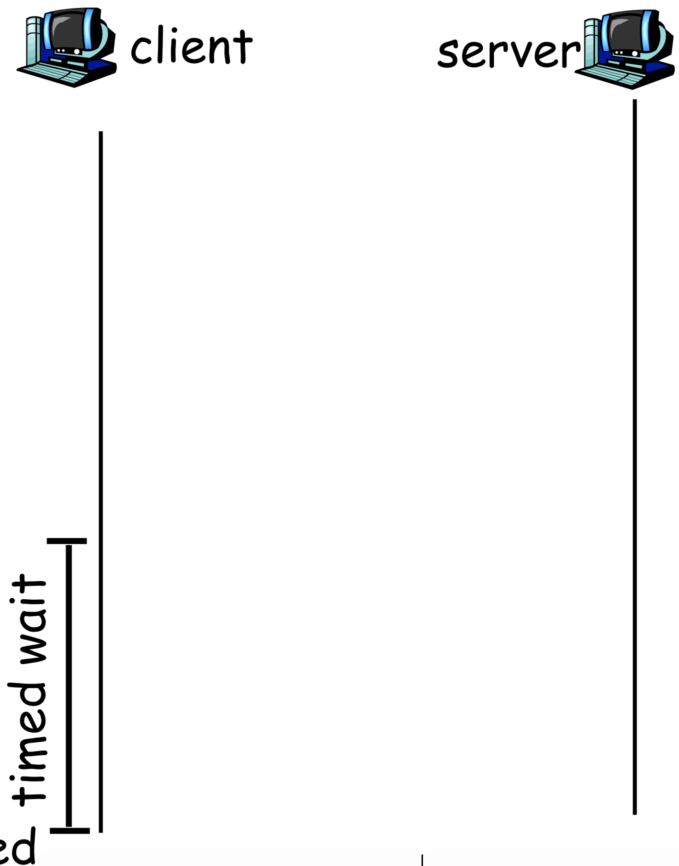
Connection Termination and State Transition (1)



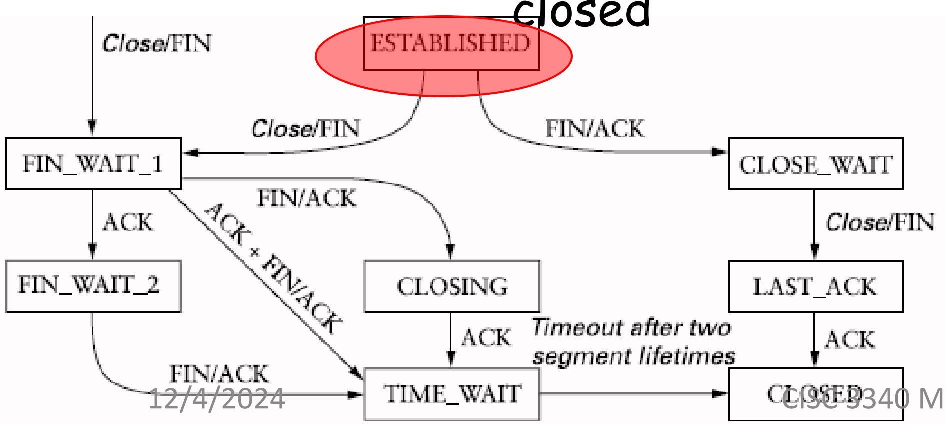
Client closes first

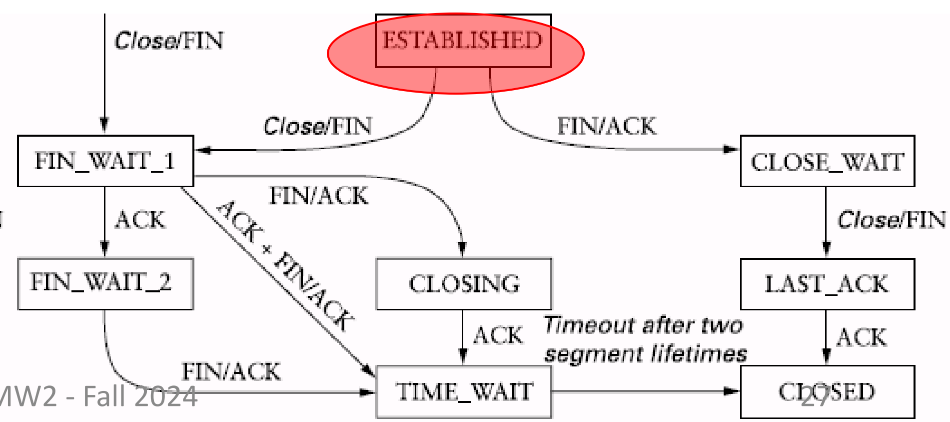


Connection Termination and State Transition (1)

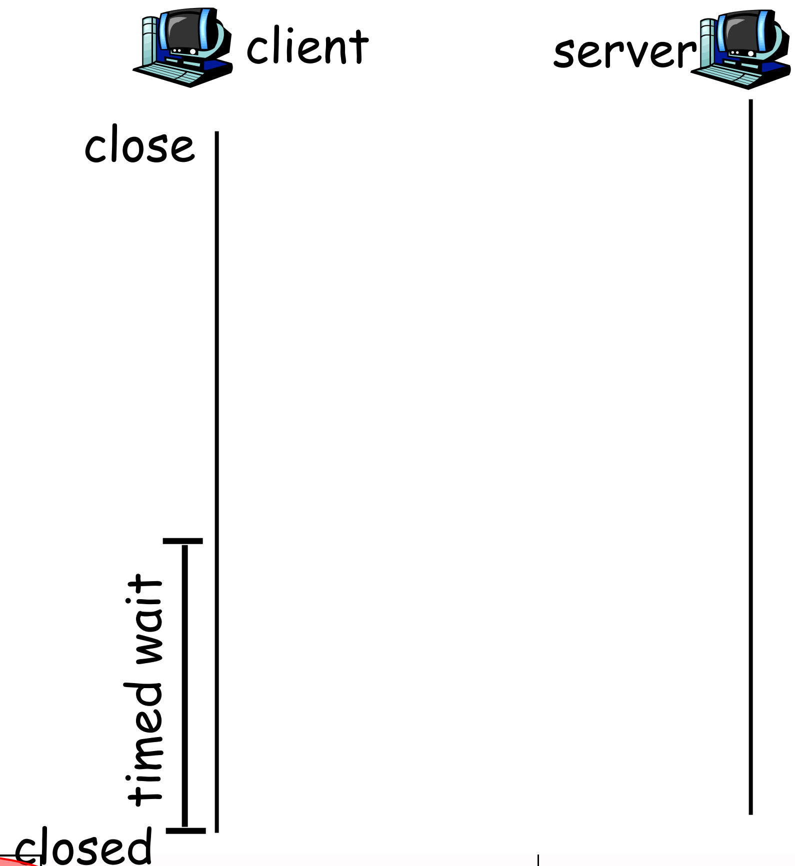


Client closes first

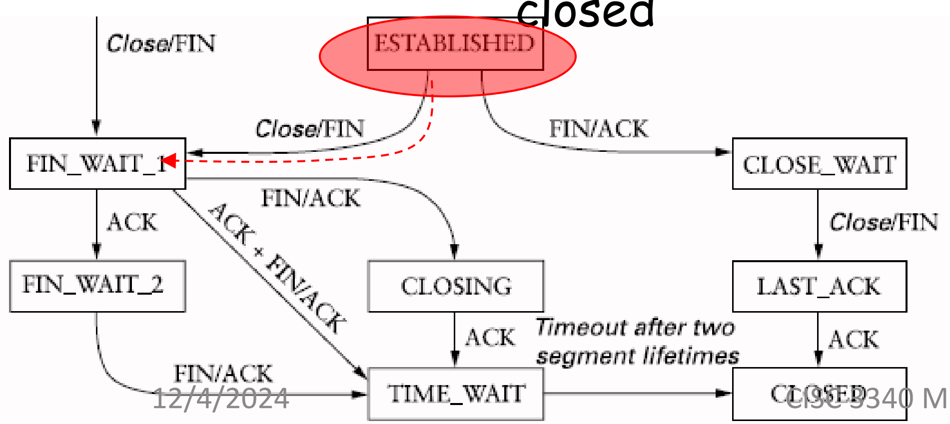


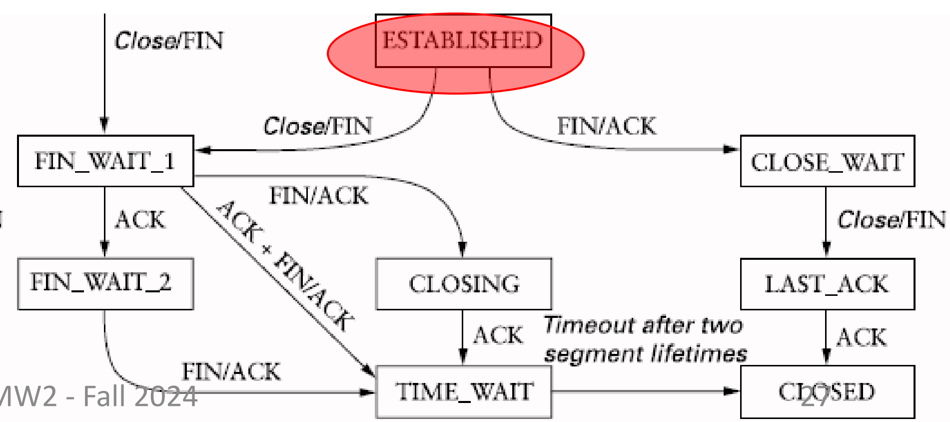


Connection Termination and State Transition (1)

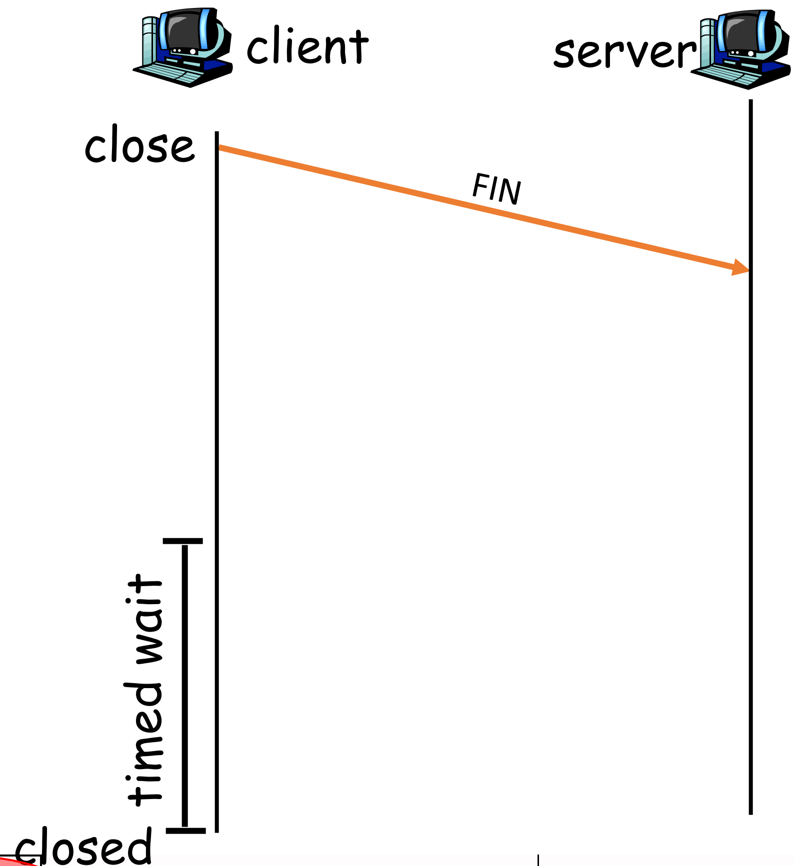


Client closes first

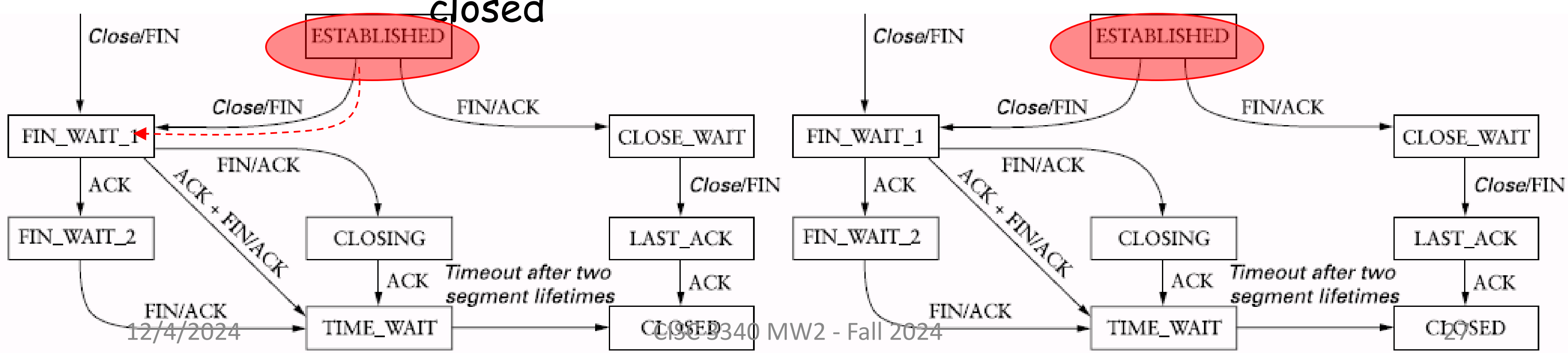




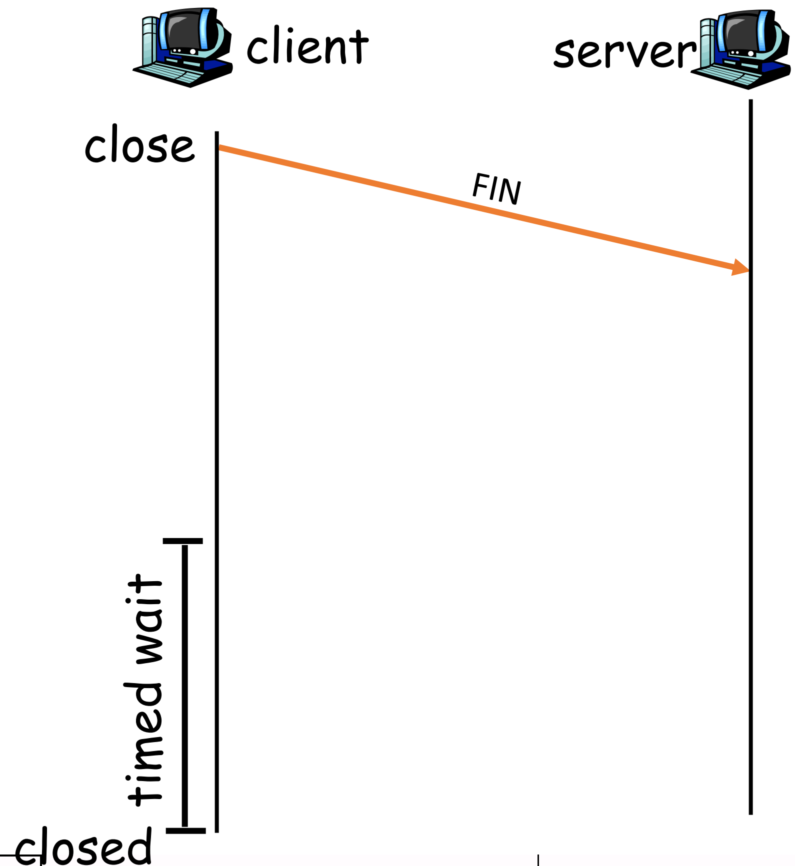
Connection Termination and State Transition (1)



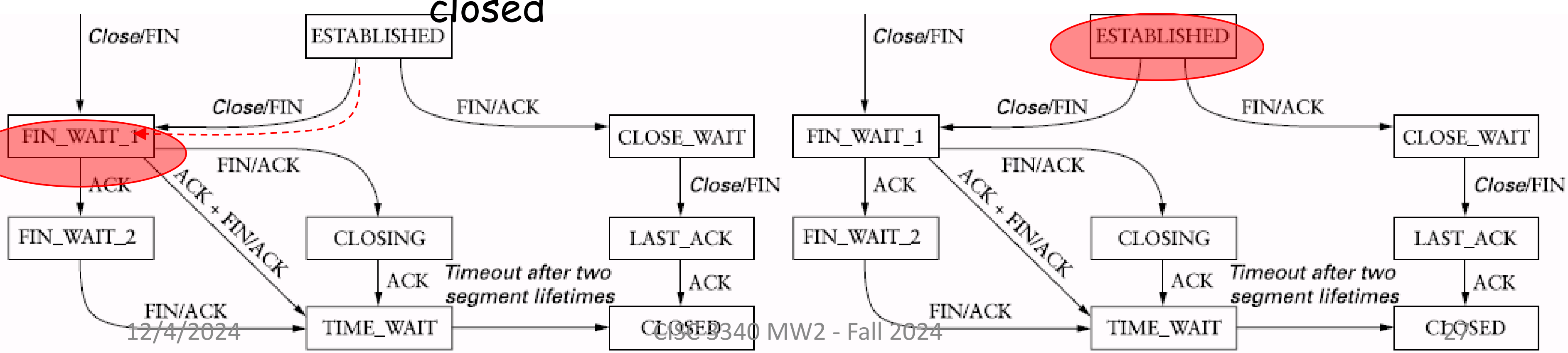
Client closes first



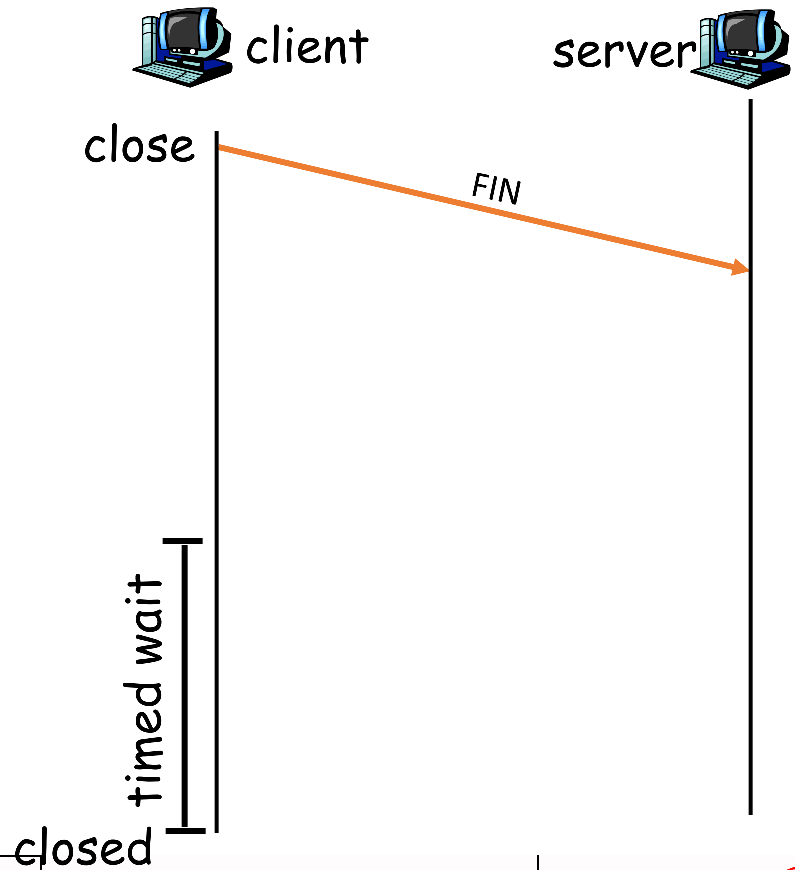
Connection Termination and State Transition (1)



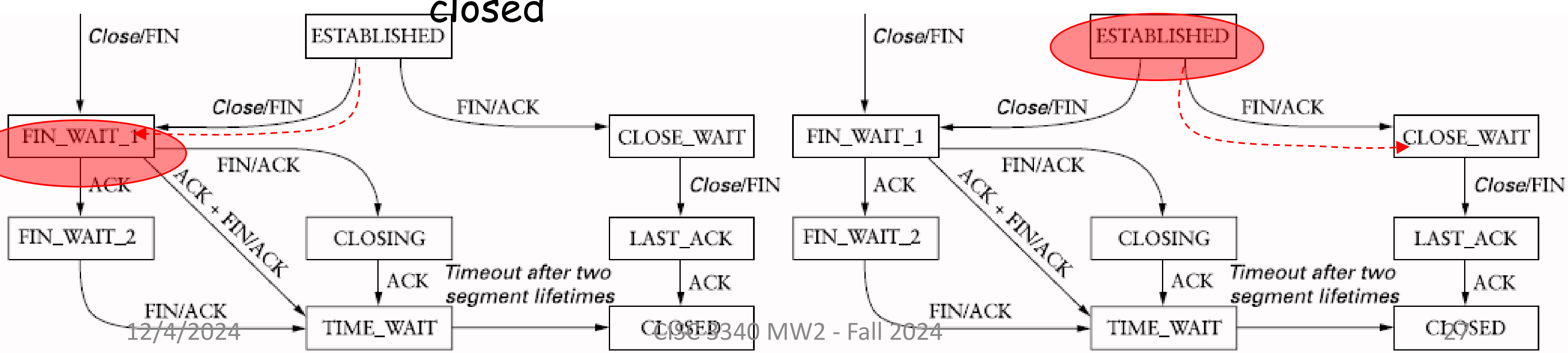
Client closes first



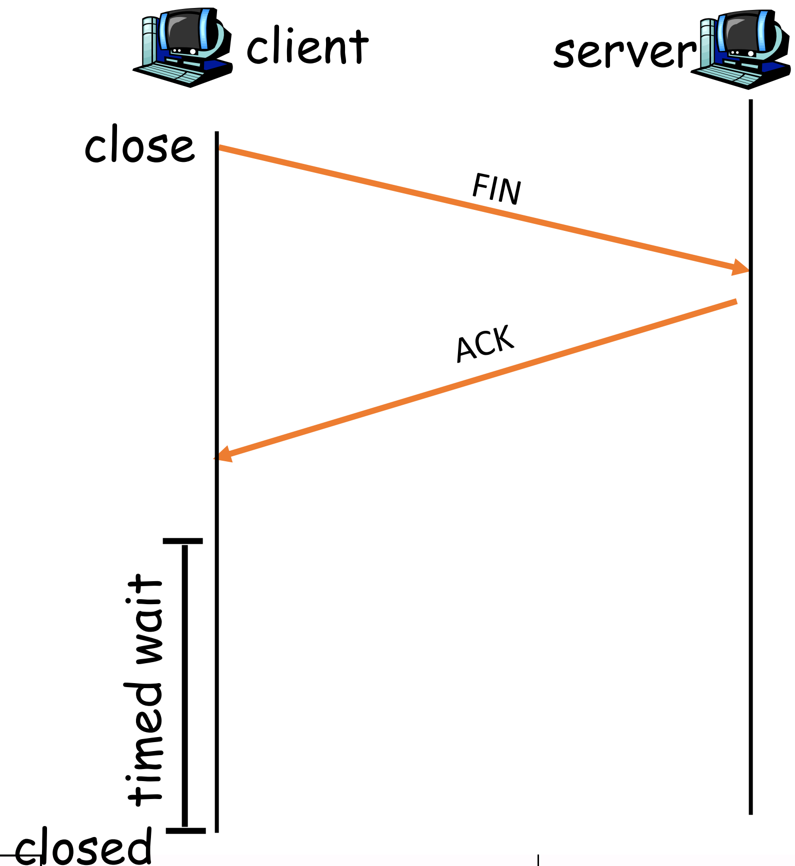
Connection Termination and State Transition (1)



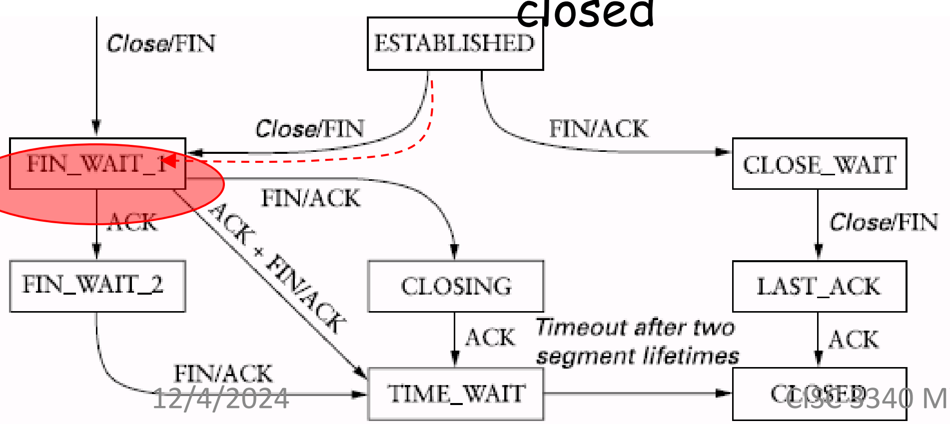
Client closes first

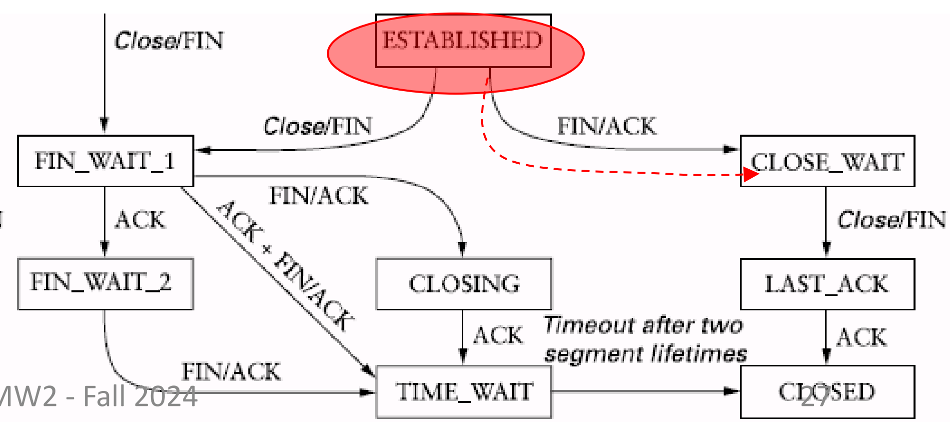


Connection Termination and State Transition (1)

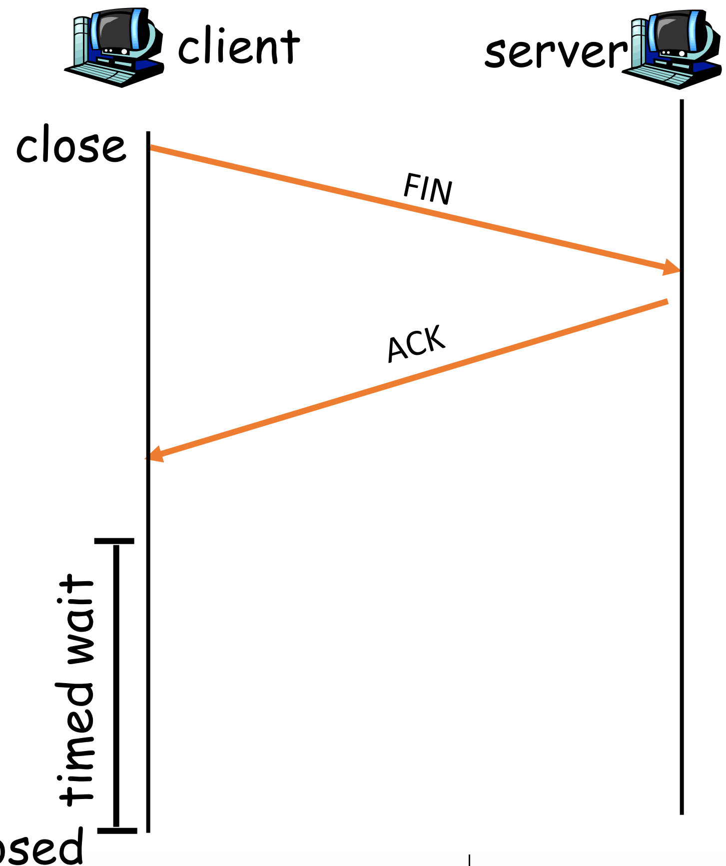


Client closes first

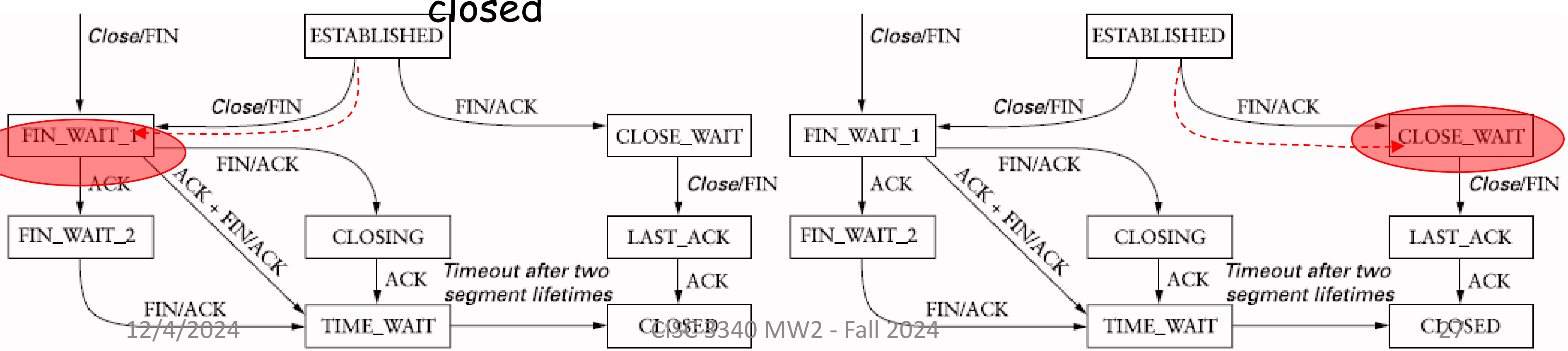




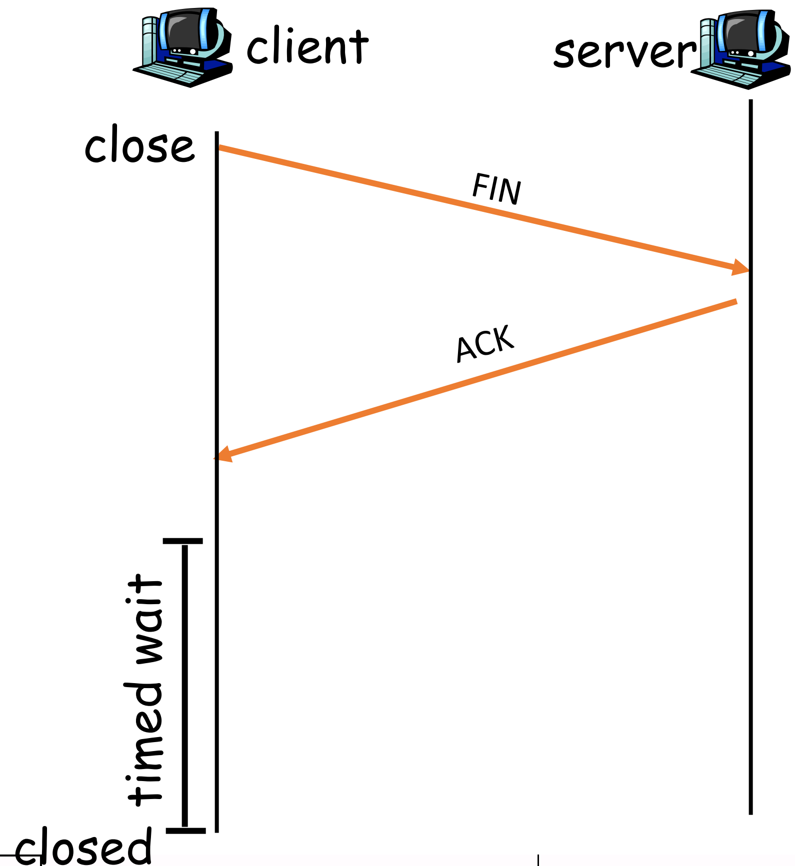
Connection Termination and State Transition (1)



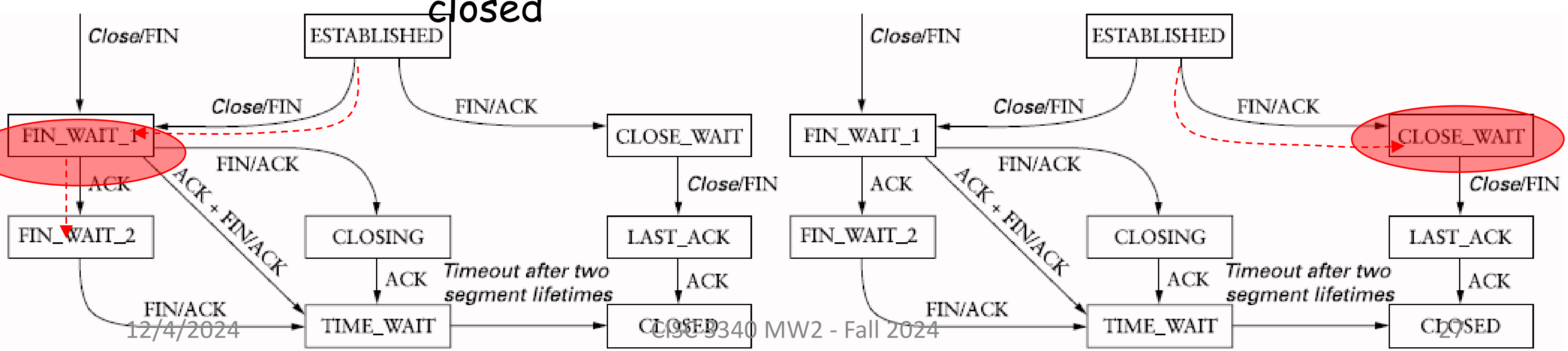
Client closes first



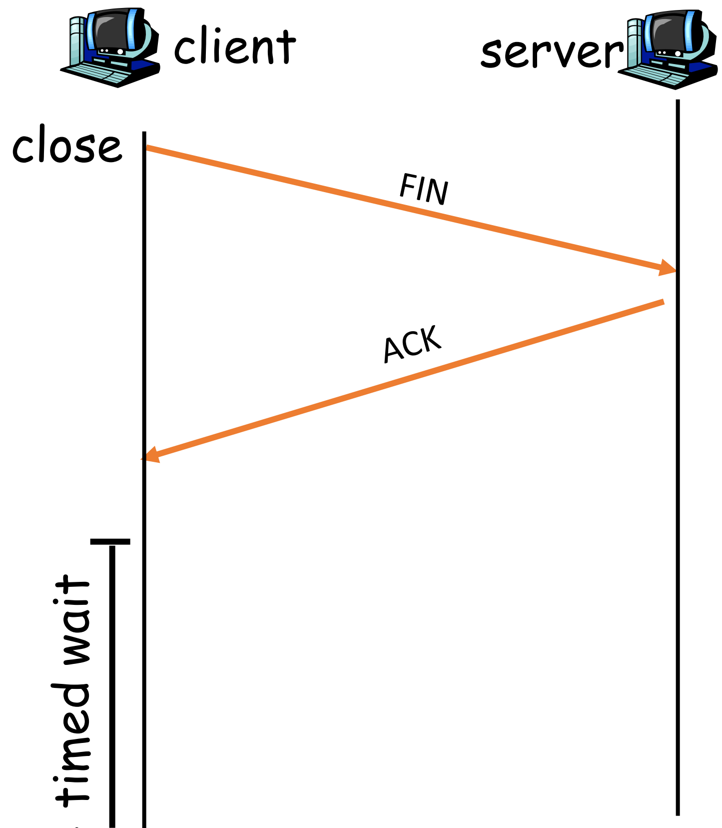
Connection Termination and State Transition (1)



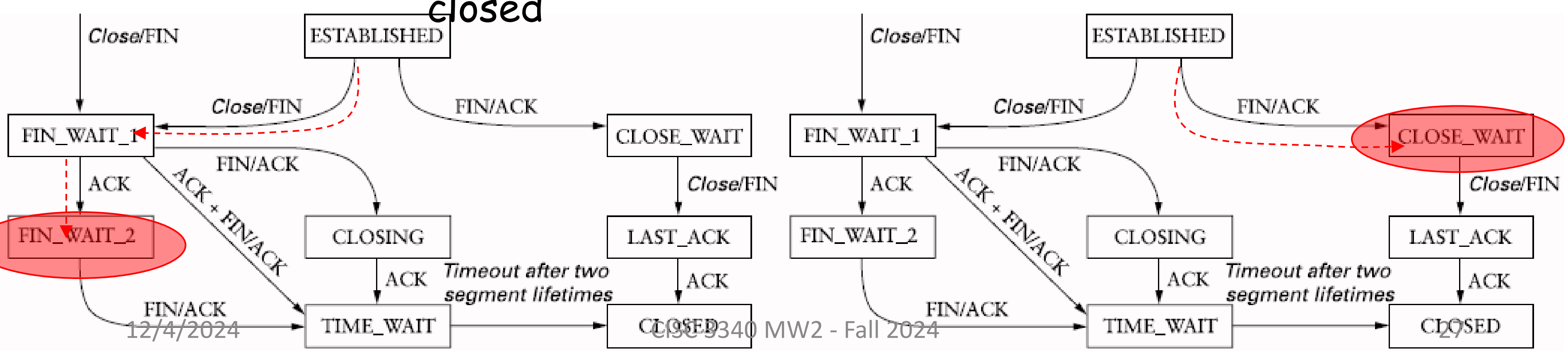
Client closes first



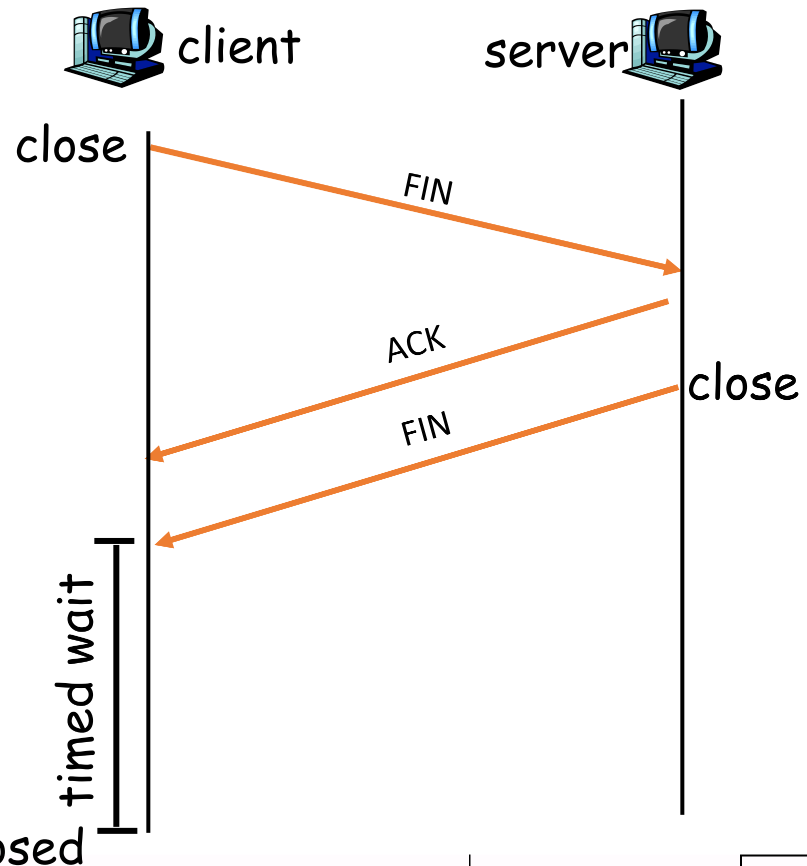
Connection Termination and State Transition (1)



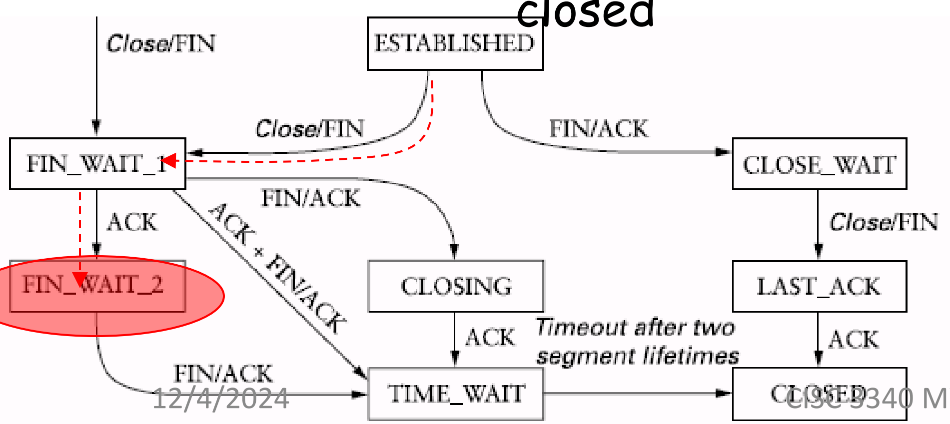
Client closes first

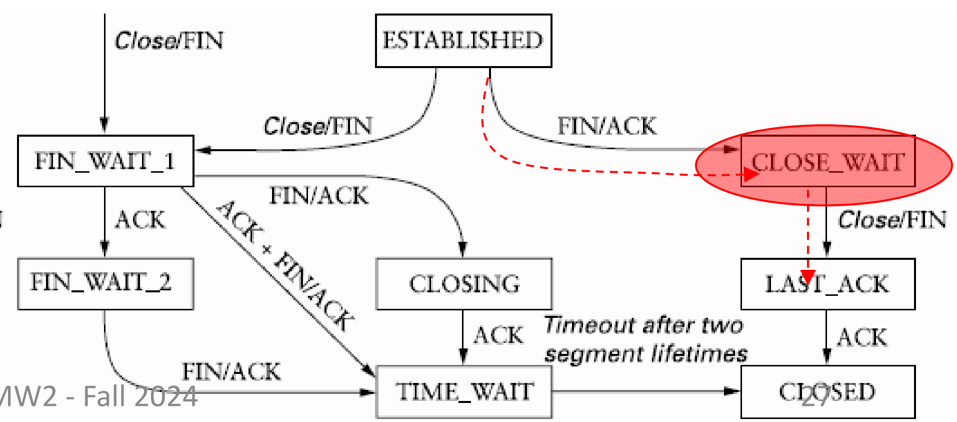


Connection Termination and State Transition (1)

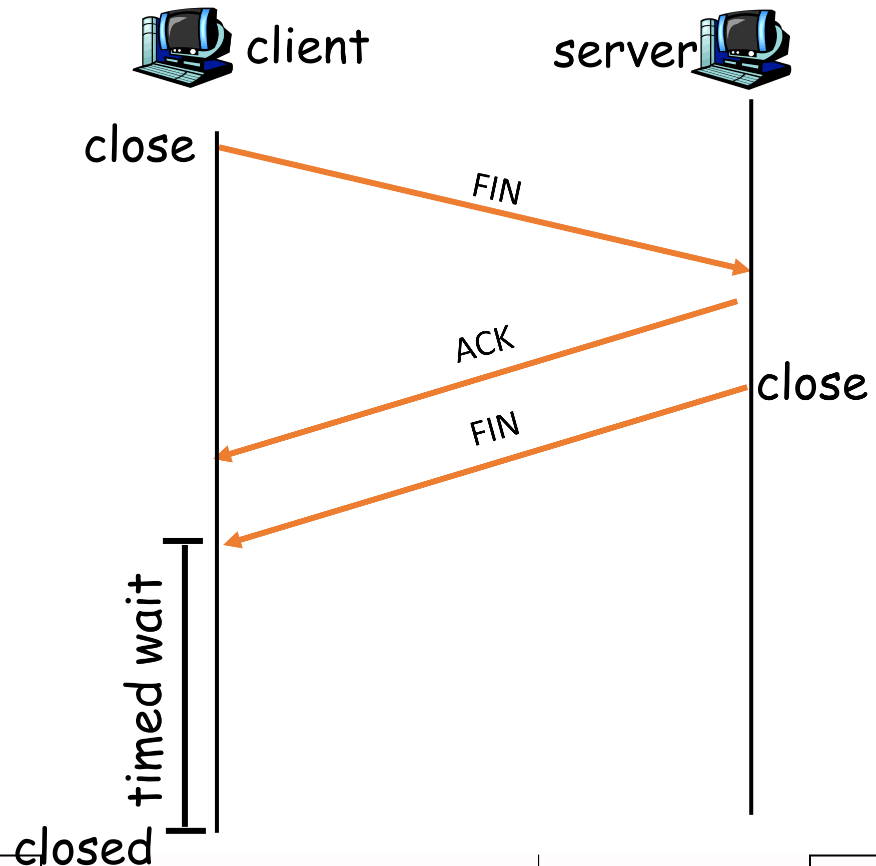


Client closes first

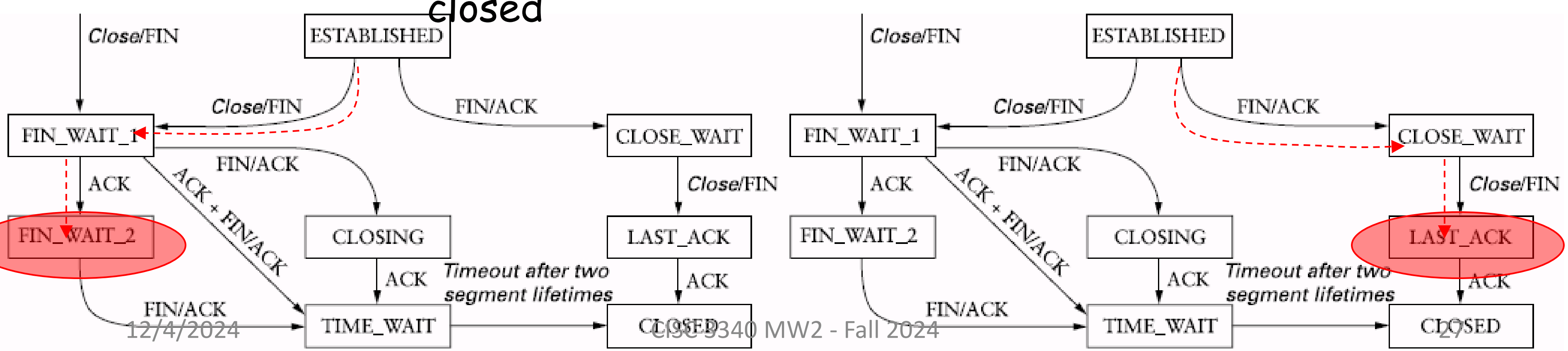




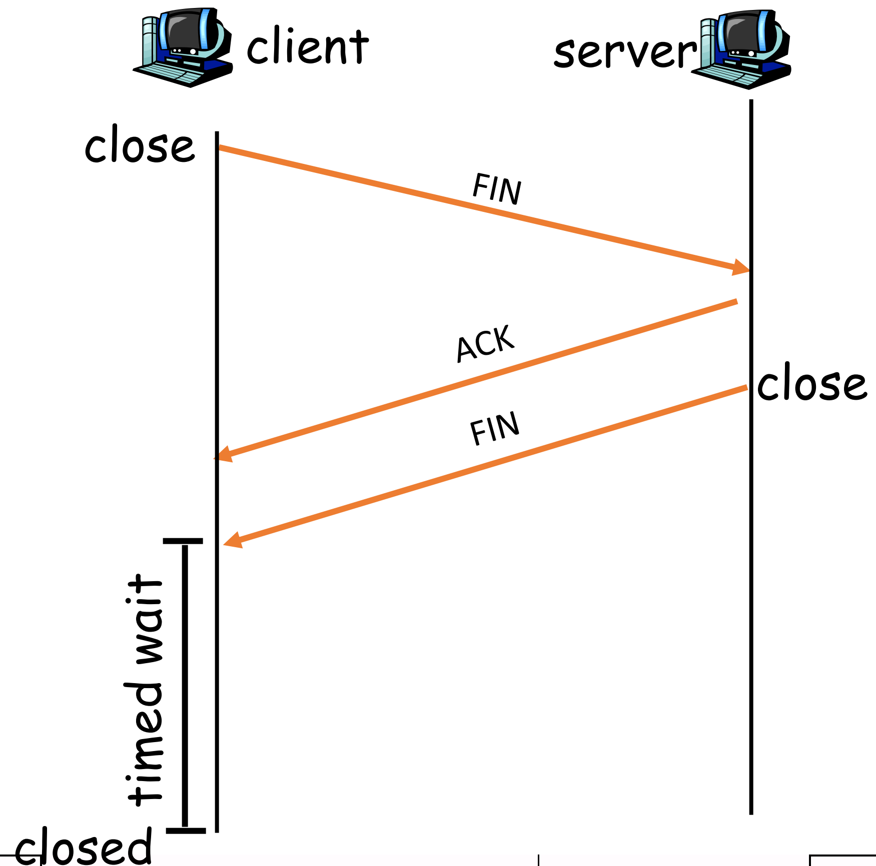
Connection Termination and State Transition (1)



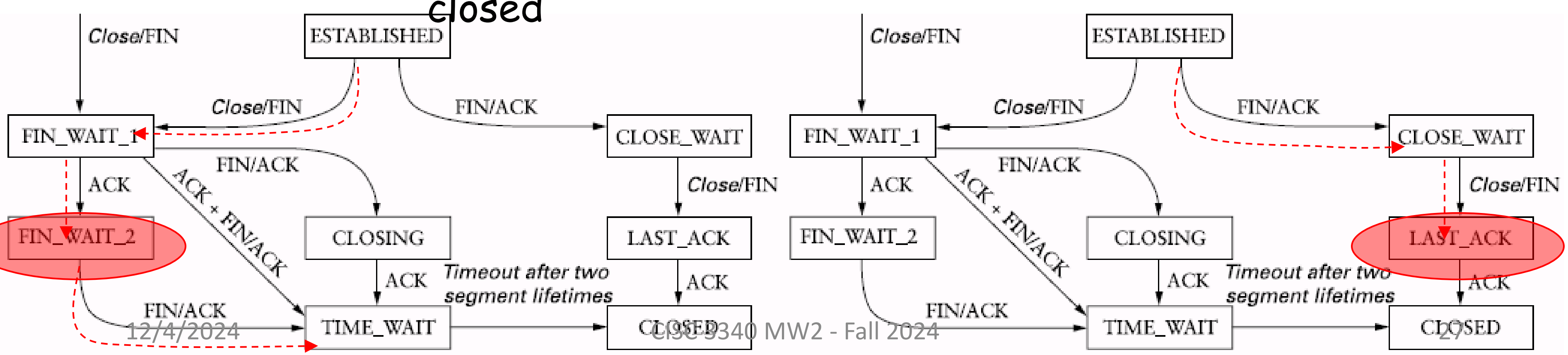
Client closes first



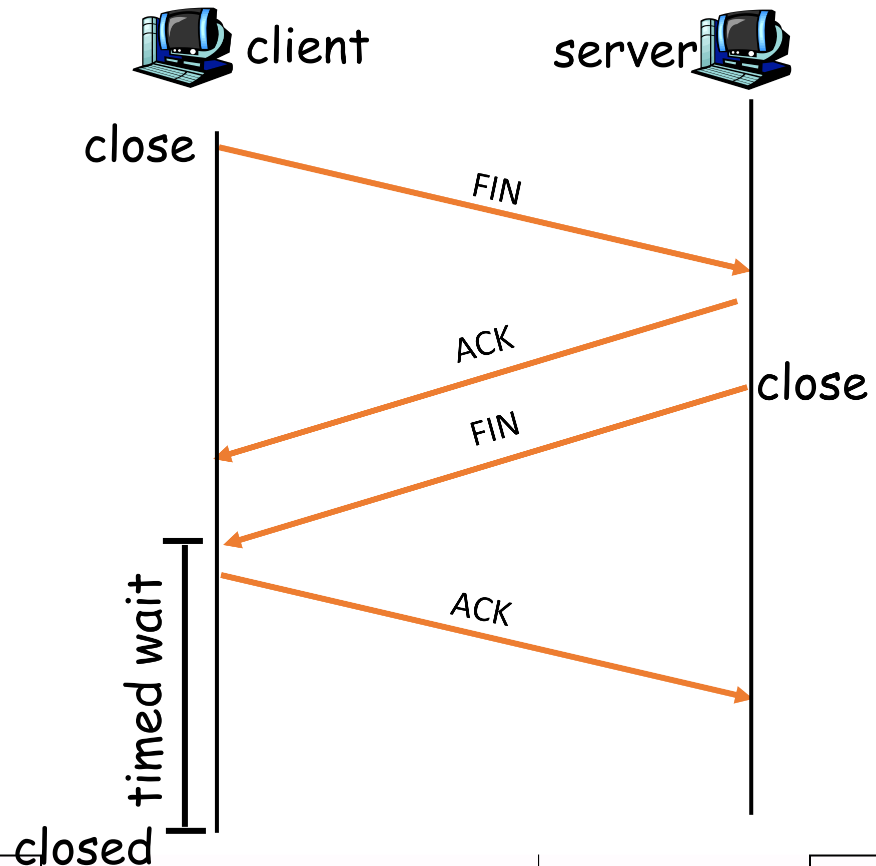
Connection Termination and State Transition (1)



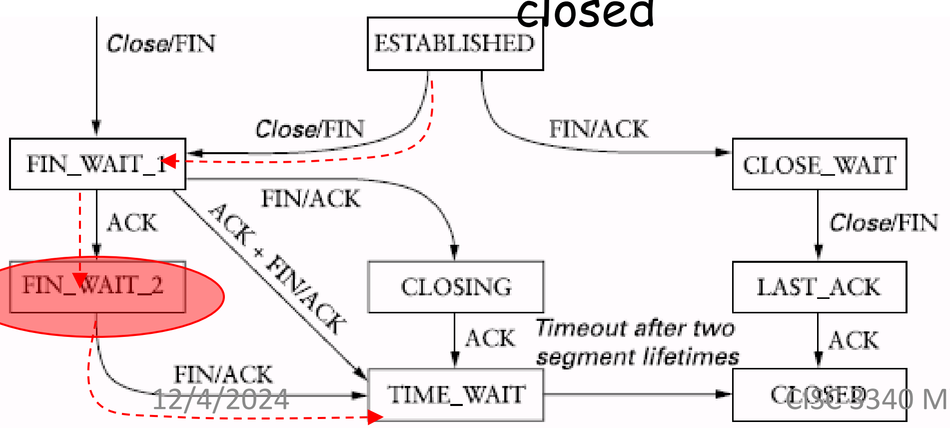
Client closes first

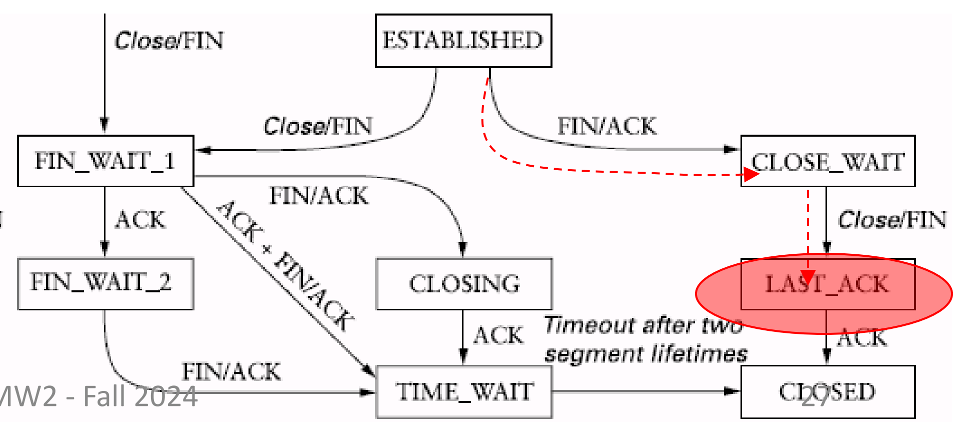


Connection Termination and State Transition (1)

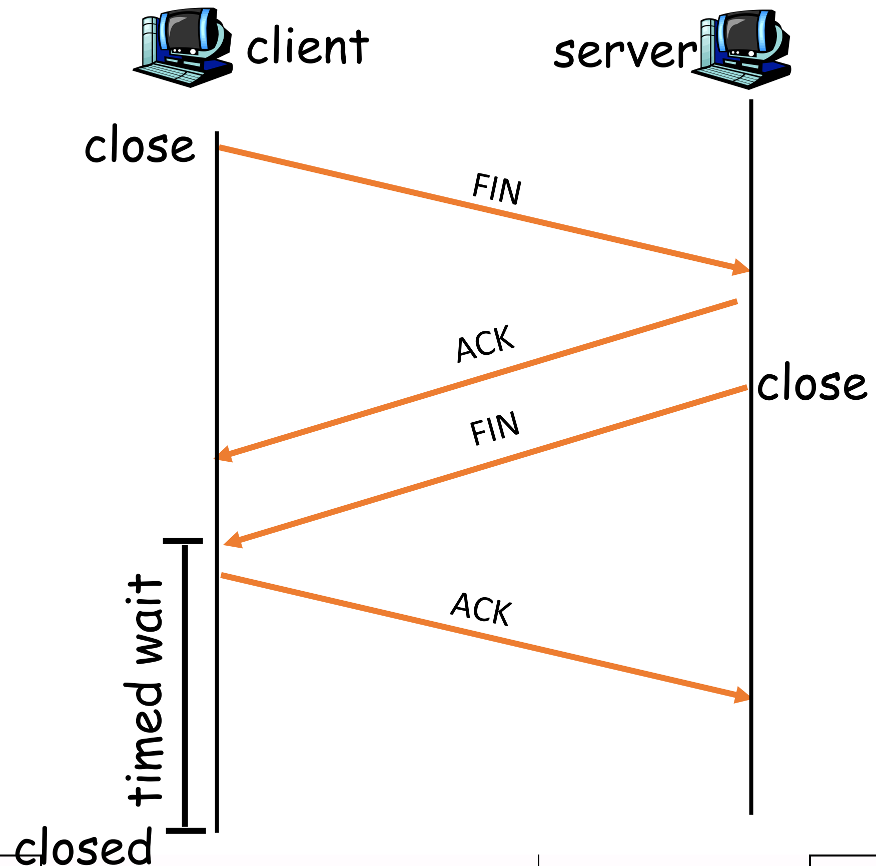


Client closes first

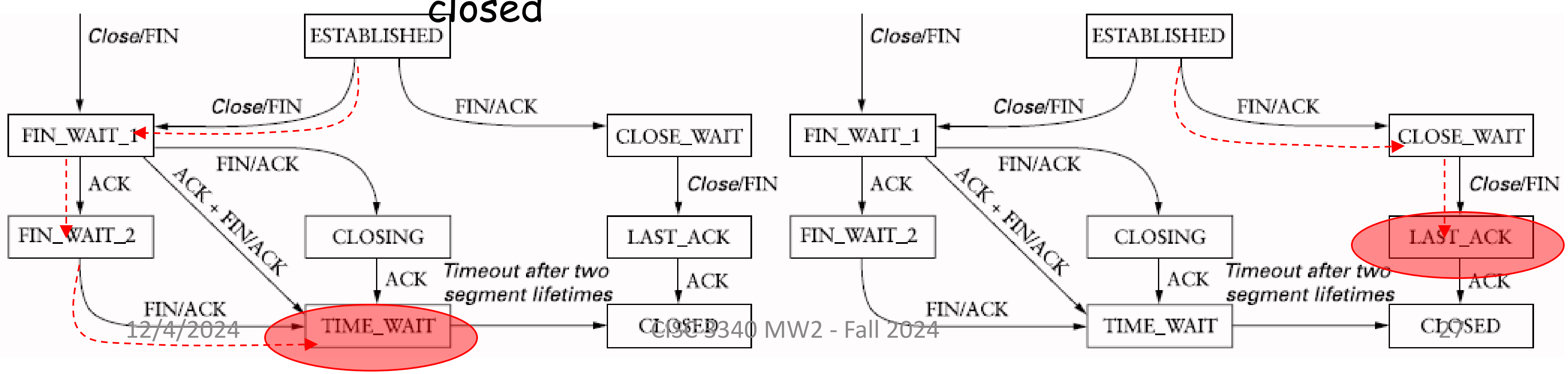




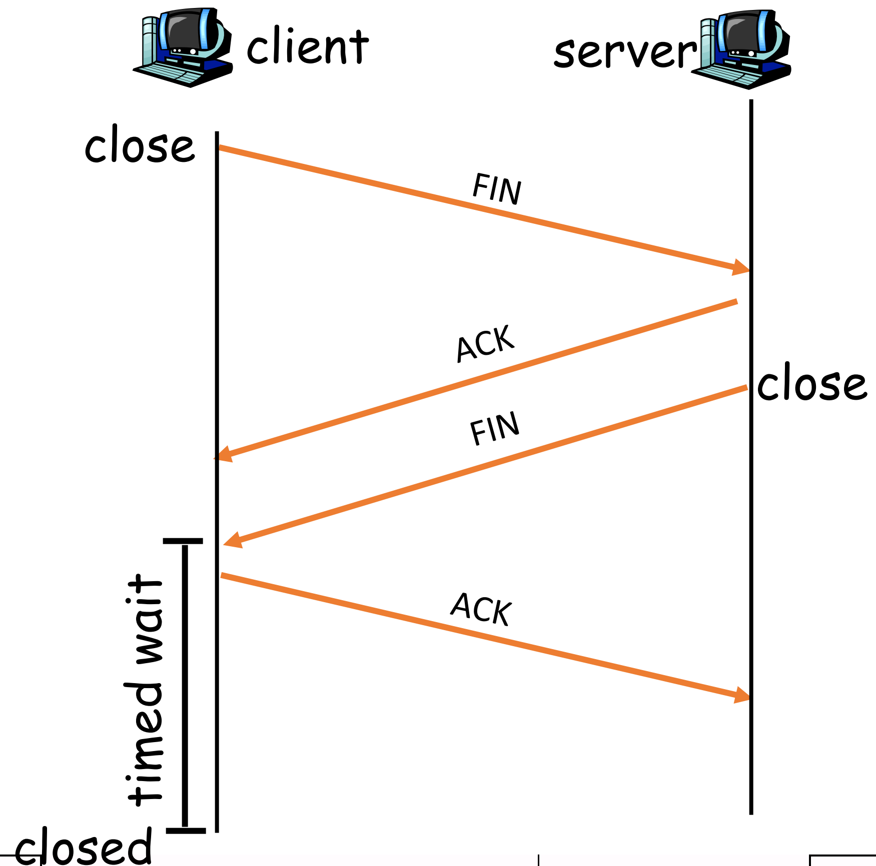
Connection Termination and State Transition (1)



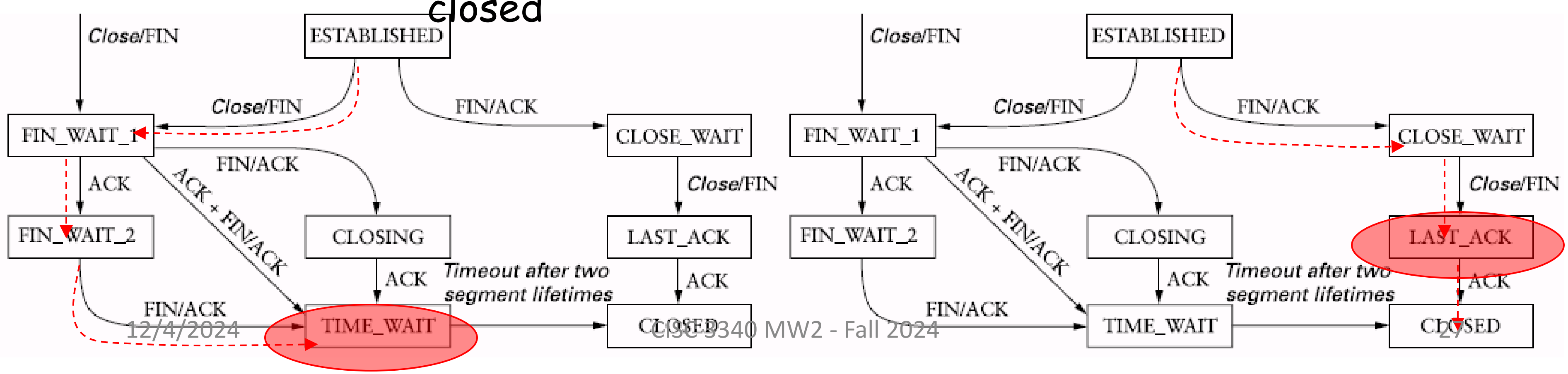
Client closes first



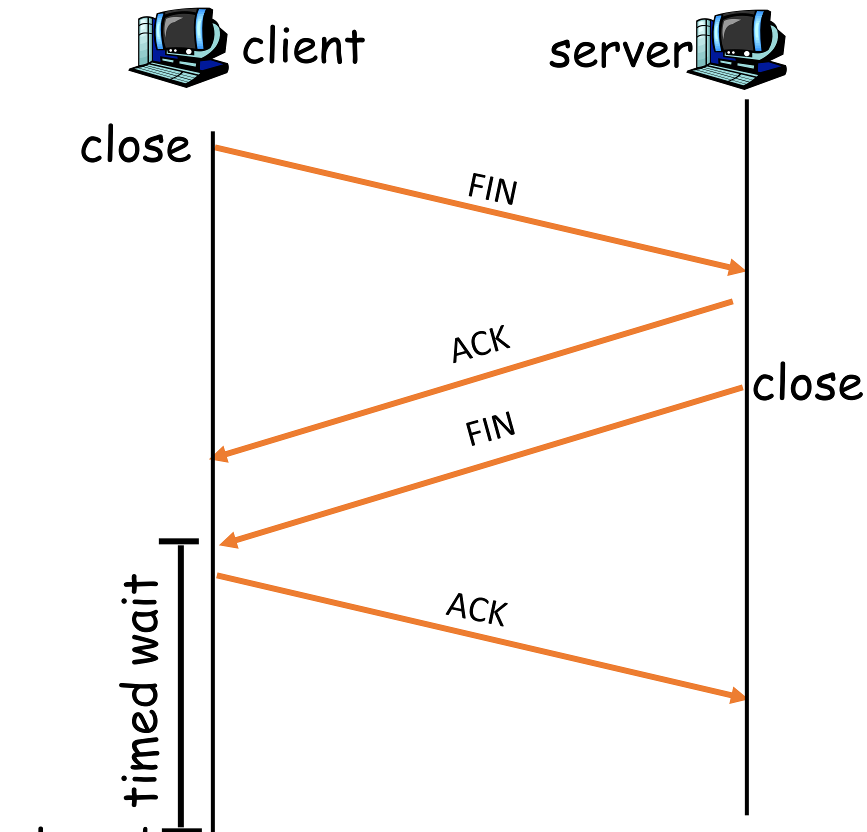
Connection Termination and State Transition (1)



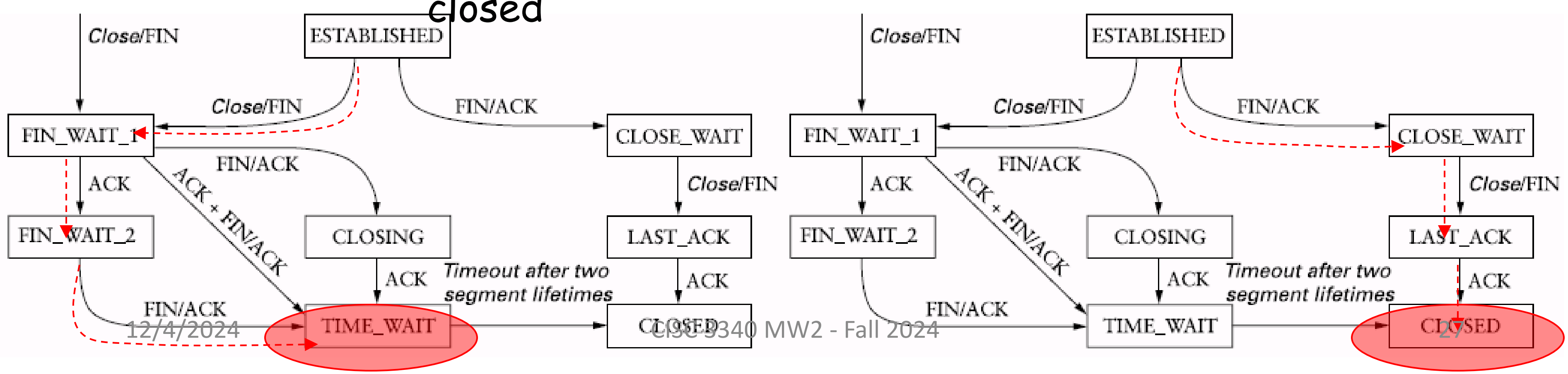
Client closes first



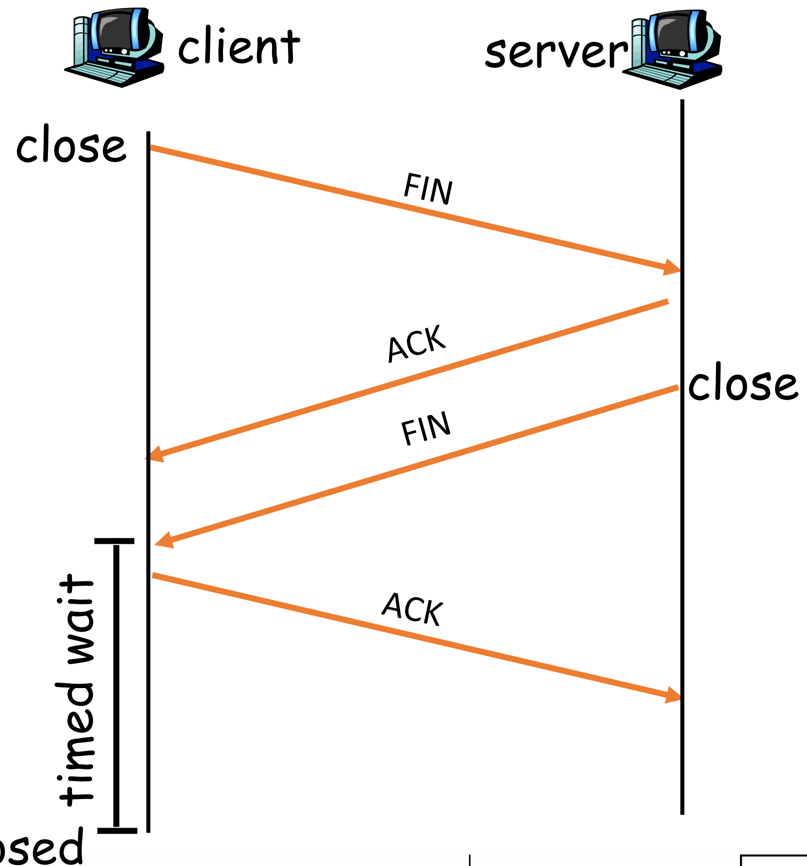
Connection Termination and State Transition (1)



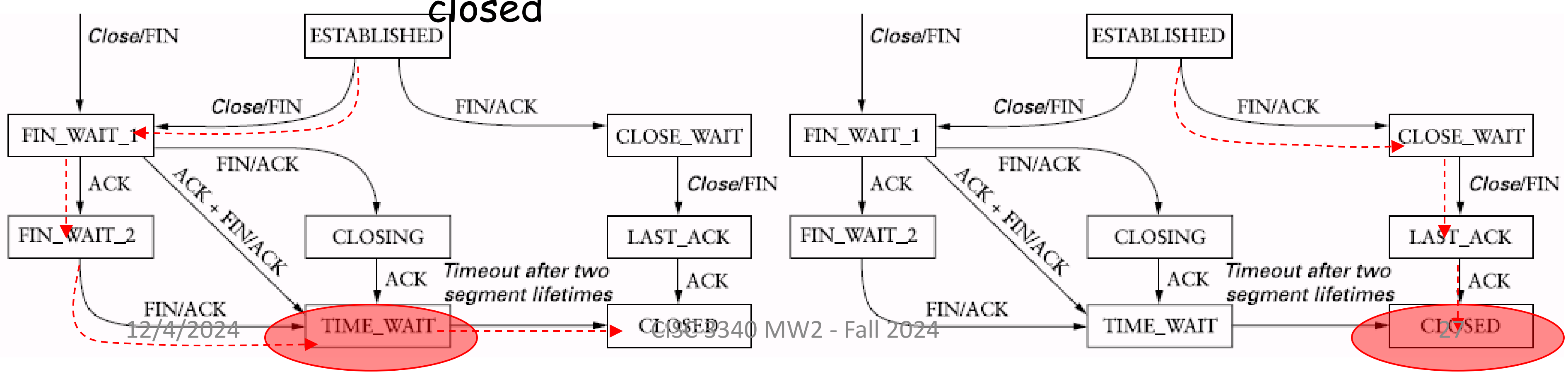
Client closes first



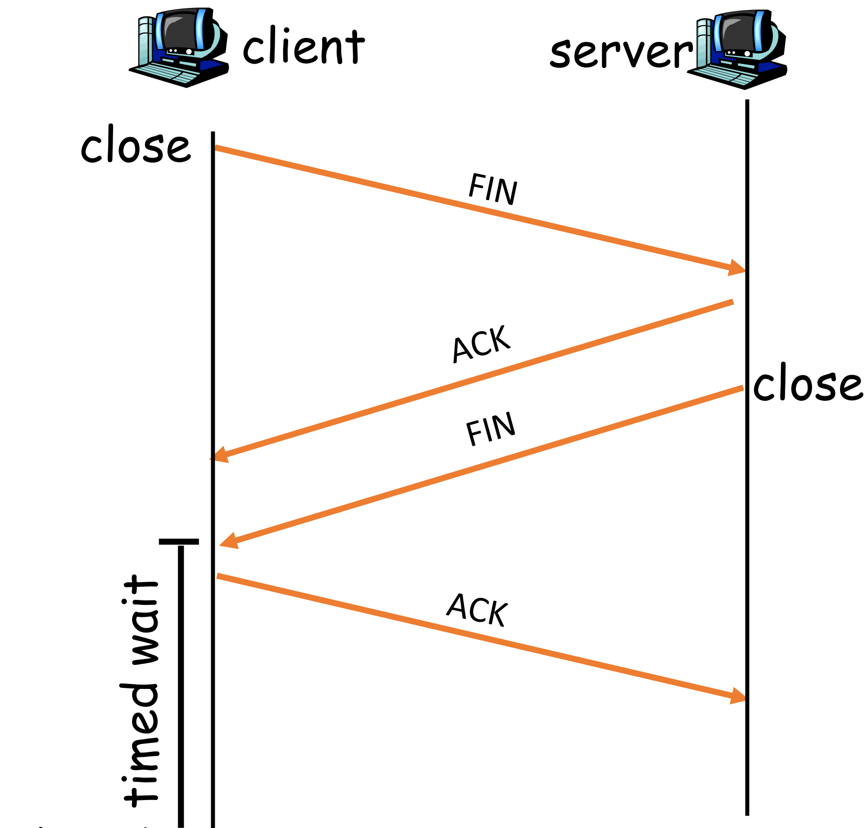
Connection Termination and State Transition (1)



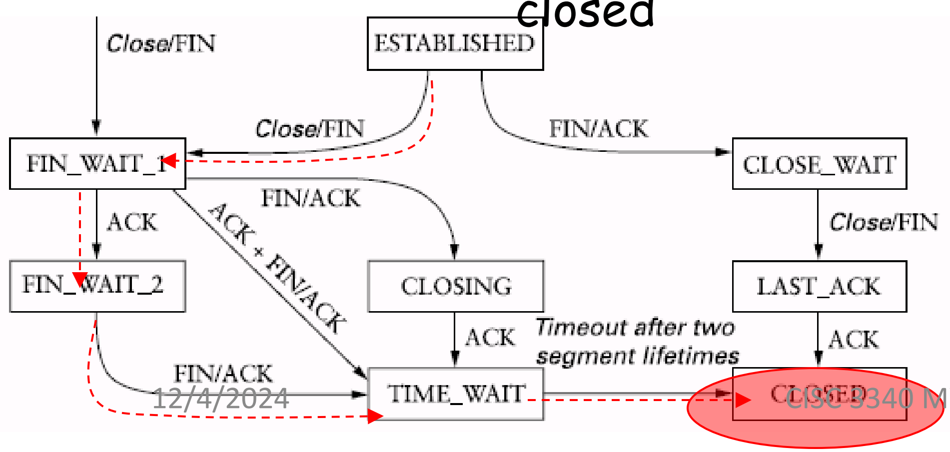
Client closes first

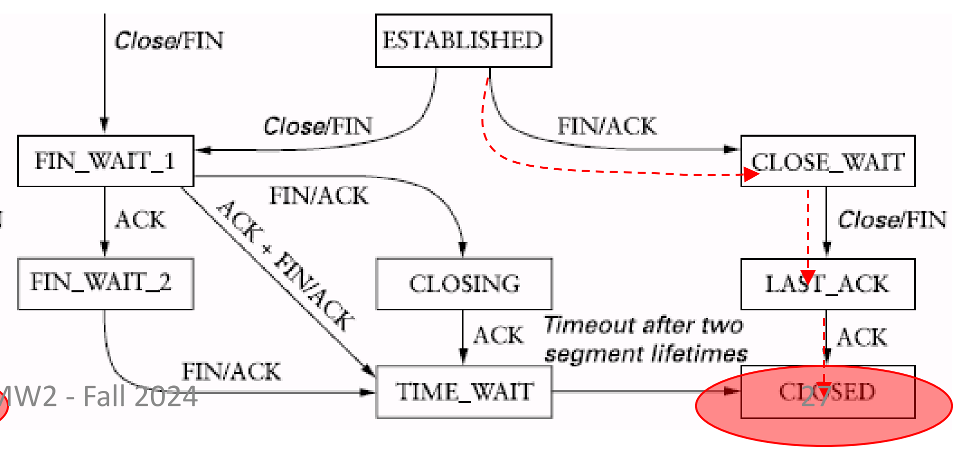


Connection Termination and State Transition (1)



Client closes first



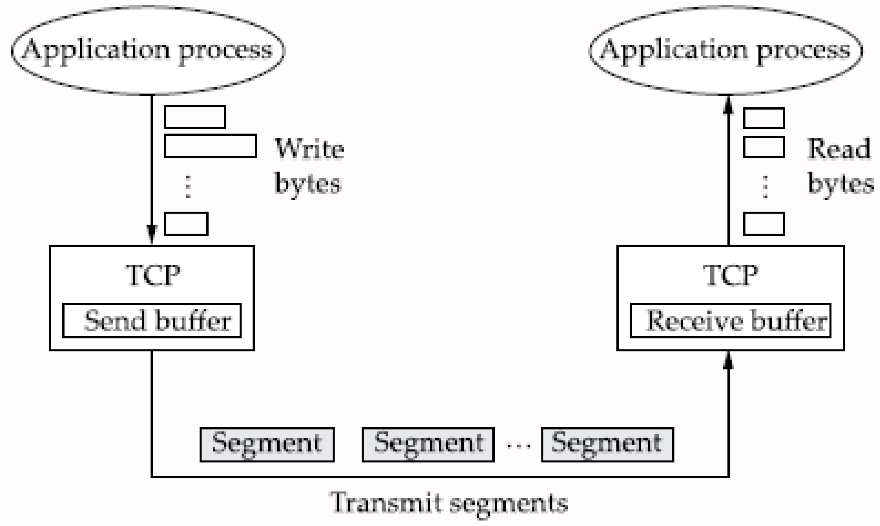


Connection Termination and State Transition (2)

- This side closes first
  - ESTABLISHED → FIN_WAIT_1 → FIN_WAIT_2 → TIME_WAIT
- Other side closes first
  - ESTABLISHED → CLOSE_WAIT → LAST_ACK → CLOSED
- Both sides close at the same time
  - ESTABLISHED → FIN_WAIT_1 → CLOSING → TIME_WAIT → CLOSED

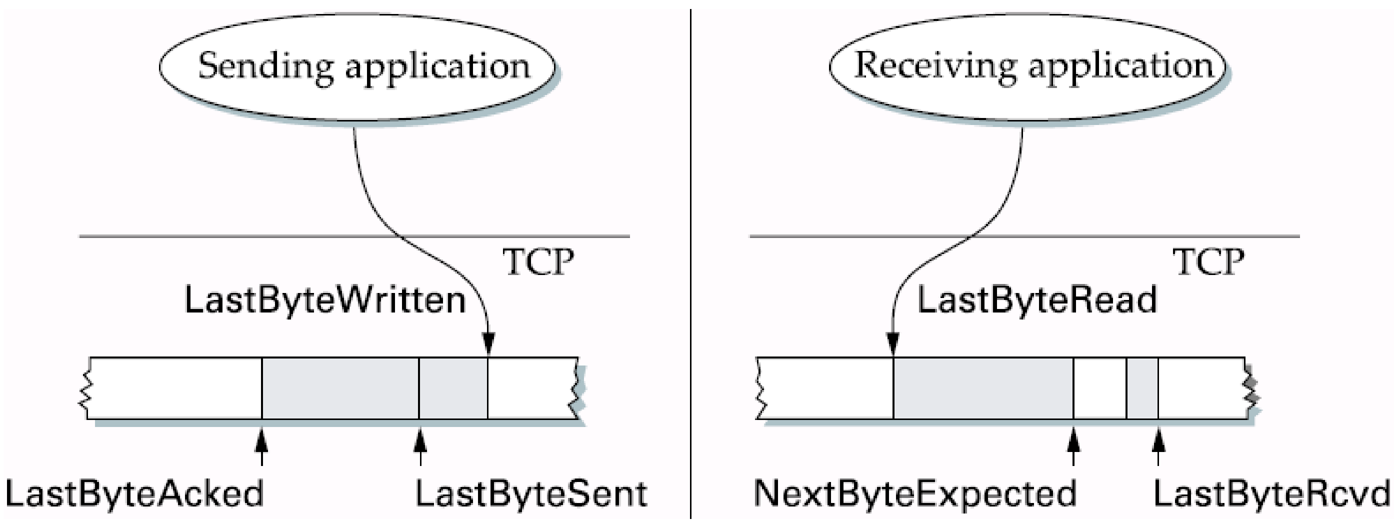
TCP Sliding Window: Why Different?

- Potentially connects many different hosts
  - need explicit connection establishment and termination
- Potentially different RTT
  - need adaptive timeout mechanism
- Potentially long delay in network
  - need to be prepared for arrival of very old packets
- Potentially different capacity at destination
  - need to accommodate different node capacity
- Potentially different network capacity
  - need to be prepared for network congestion



TCP Sliding Window: Reliable and Ordered Delivery

TCP uses cumulative acknowledgements to acknowledge receiving of all the bytes up to the first missing byte



- Sending side
  - LastByteAcked ≤ LastByteSent
  - LastByteSent ≤ LastByteWritten
  - buffer bytes between LastByteAcked and LastByteWritten

Receiving side

LastByteRead < NextByteExpected

NextByteExpected ≤ LastByteRcvd +1

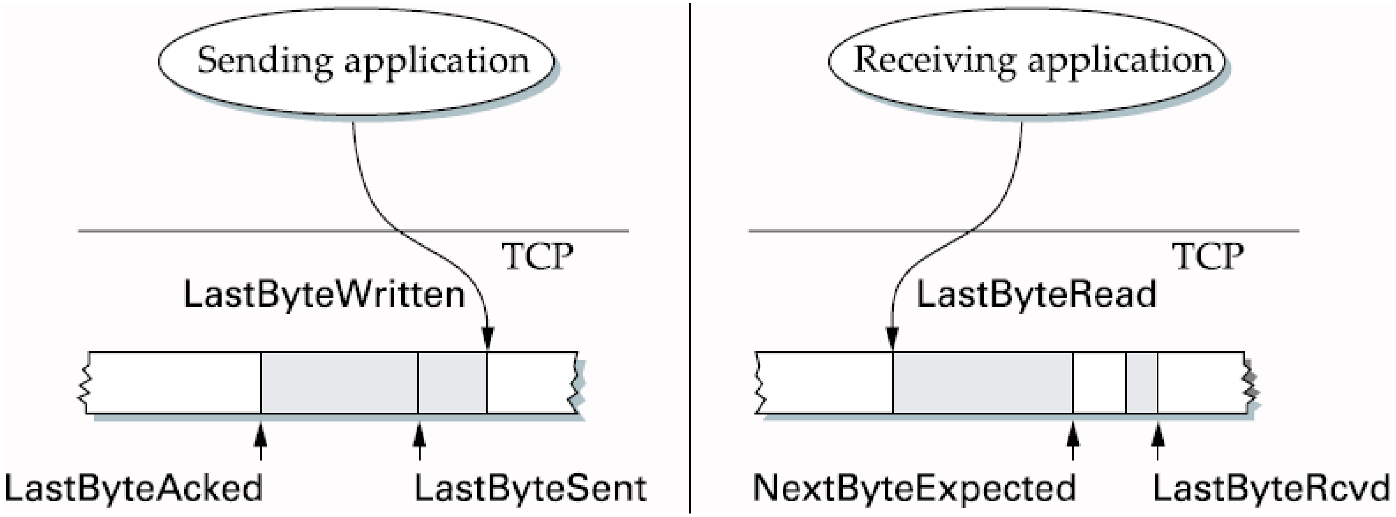
buffer bytes betweenNextByteRead and LastByteRcvd

TCP Flow Control (1)

- receive side of TCP connection has a receive buffer
- app process may be slow at reading from buffer
- speed-matching service: matching the send rate to the receiving app's drain rate

flow control

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



TCP Flow Control (2)

- Send buffer size: MaxSendBuffer
- Receive buffer size: MaxRcvBuffer
- Receiving side
  - LastByteRcvd - LastByteRead ≤ MaxRcvBuffer
  - AdvertisedWindow = MaxRcvBuffer – ((NextByteExpected -1) - LastByteRead)) → maximum possible free space remaining in the buffer
- Sending side
  - LastByteSent - LastByteAcked ≤ AdvertisedWindow
    - LastByteSent – LastByteAcked: unacknowledged bytes sender has put in TCP
    - Otherwise, the sender may overrun the receiver
  - EffectiveWindow = AdvertisedWindow - (LastByteSent -LastByteAcked) → how much data it can sent
  - LastByteWritten - LastByteAcked ≤ MaxSendBuffer
  - If the sender tries to write y bytes to TCP
    - block sender if (LastByteWritten - LastByteAcked) + y > MaxSenderBuffer
- Always send ACK in response to arriving data segment
- Persist when AdvertisedWindow = 0

Flow Control and Buffering (3)

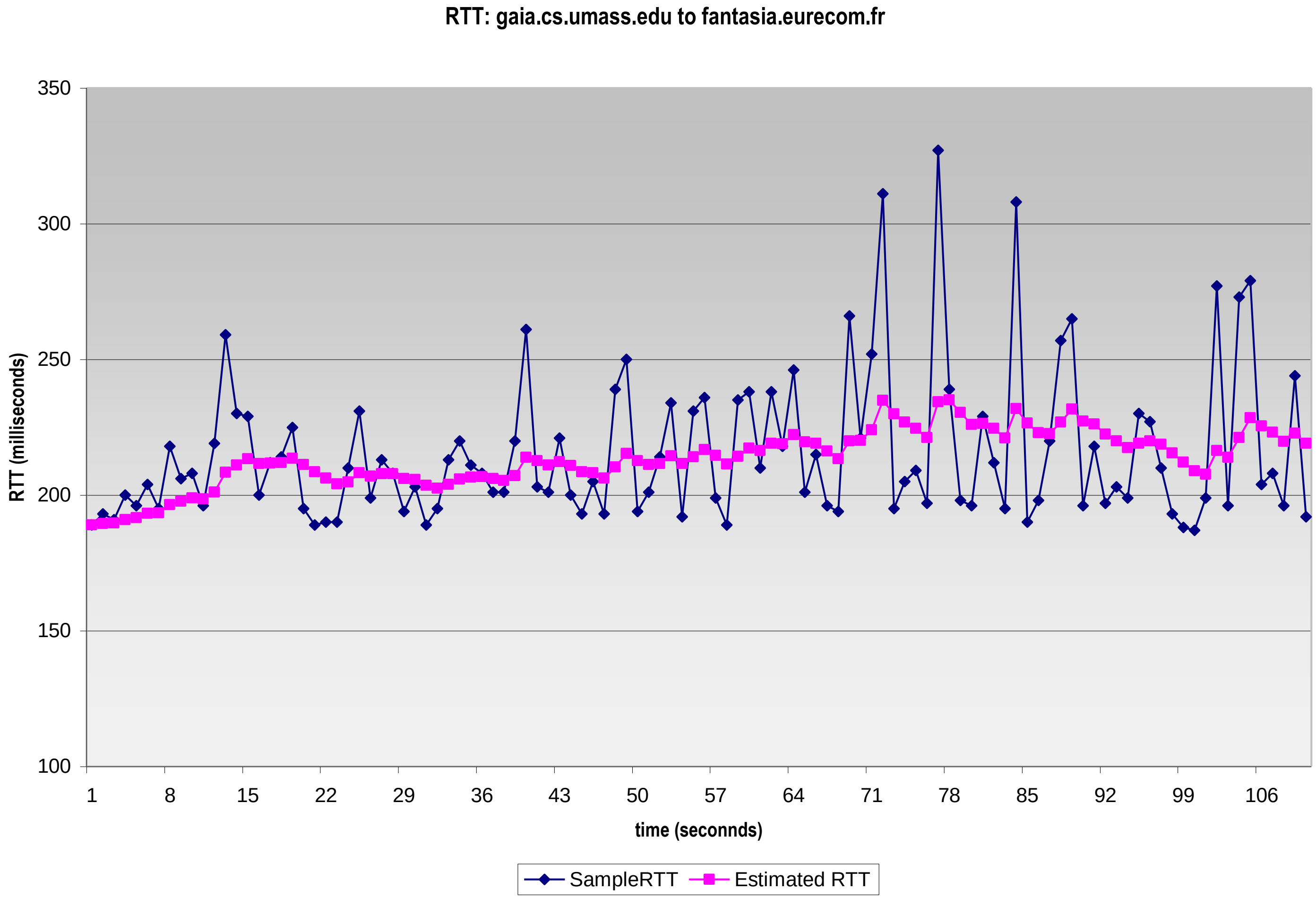
	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	• • •	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3>	←	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3>	→	A has 1 buffer left
8	→	<seq = 4, data = m4>	→	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2>	→	A times out and retransmits
10	←	<ack = 4, buf = 0>	←	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1>	←	A may now send 5
12	←	<ack = 4, buf = 2>	←	B found a new buffer somewhere
13	→	<seq = 5, data = m5>	→	A has 1 buffer left
14	→	<seq = 6, data = m6>	→	A is now blocked again
15	←	<ack = 6, buf = 0>	←	A is still blocked
16	• • •	<ack = 6, buf = 4>	←	Potential deadlock

Dynamic buffer allocation. The arrows show the direction of transmission. An ellipsis (…) indicates a lost TCP segment

Adaptive Retransmission: Original Algorithm

- Measure SampleRTT for each segment/ACK pair
- Compute weighted average of RTT
  - EstimatedRTT = α x EstimatedRTT + β x SampleRTT
  - where α **+** β **= 1**
    - α between 0.8 and 0.9
    - β between 0.1 and 0.2
  - Set timeout based on EstimatedRTT
    - TimeOut = 2 x EstimatedRTT

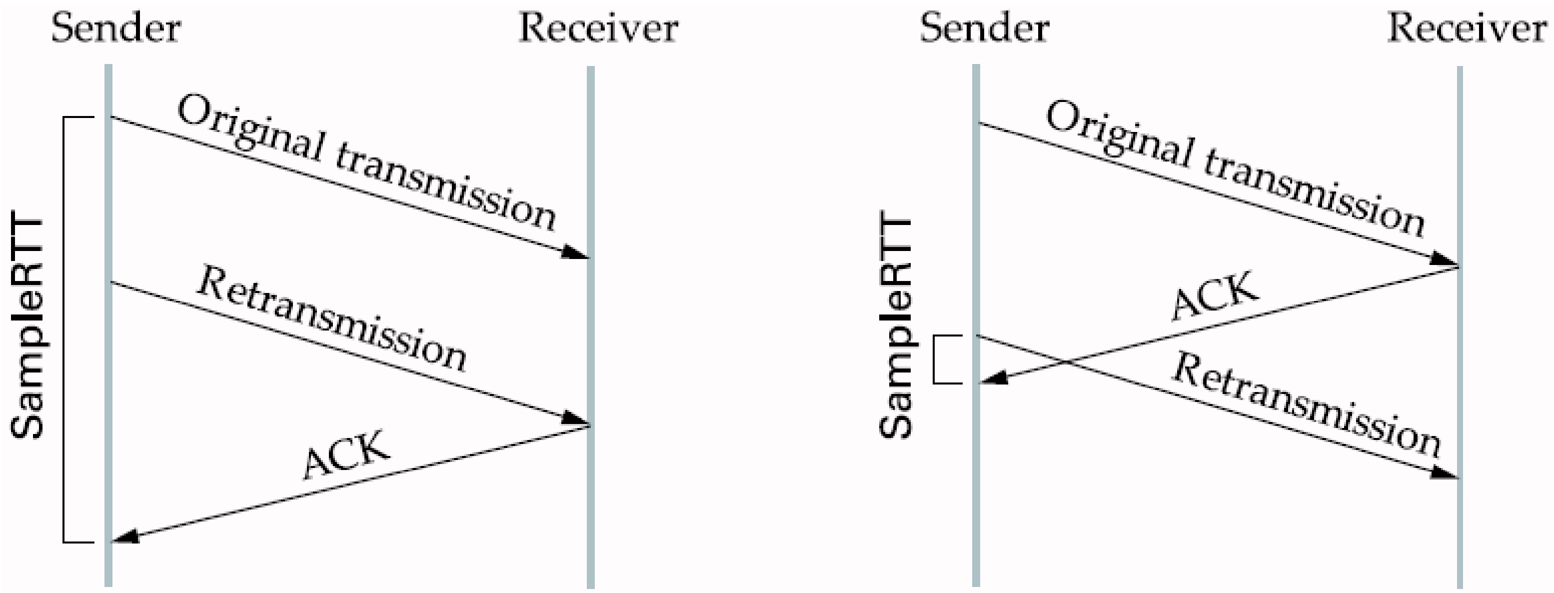
Example RTT estimation:



Adaptive Retransmission: Karn/Partridge Algorithm

Problem with original algorithm

ACK does not really acknowledge a transmission, it acknowledges the receipt of data → can not distinguish an ACK is for which transmission/retransmission of a segment



- Do not sample RTT when retransmitting
- **Double timeout after each retransmission**
  - Congestion is the most likely cause of lost segments → TCP should not react too aggressively to a timeout

Jacobson/ Karels Algorithm

- Previous approaches did not take the variance of the sample RTT into account
  - If no variance, Estimated RTT is good enough, 2 $\times$ Estimated RTT is too pessimistic
  - If variance large, timeout value should not be too dependent on Estimated RTT
- New Calculations for average RTT
  - Difference = SampleRTT – EstimtaedRTT
  - EstimatedRTT = EstimatedRTT + (δ x Difference)
  - Deviation = Deviation + δ(|Difference| - Deviation)
    - where δ is a factor between 0 and 1
  - Consider variance when setting timeout value
    - TimeOut = μ x EstimatedRTT + ϕ x Deviation
    - where μ = 1 and ϕ = 4
- Notes
  - algorithm only as good as granularity of clock (500ms on Unix)
  - accurate timeout mechanism important to congestion control

TCP: Sequence Number Wrap Around

Bandwidth	Time until Wraparound
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
Fast Ethernet (100 Mbps)	6 minutes
OC-3 (155 Mbps)	4 minutes
OC-12 (622 Mbps)	55 seconds
OC-48 (2.5 Gbps)	14 seconds

Time until 32-bit sequence number space wraps around

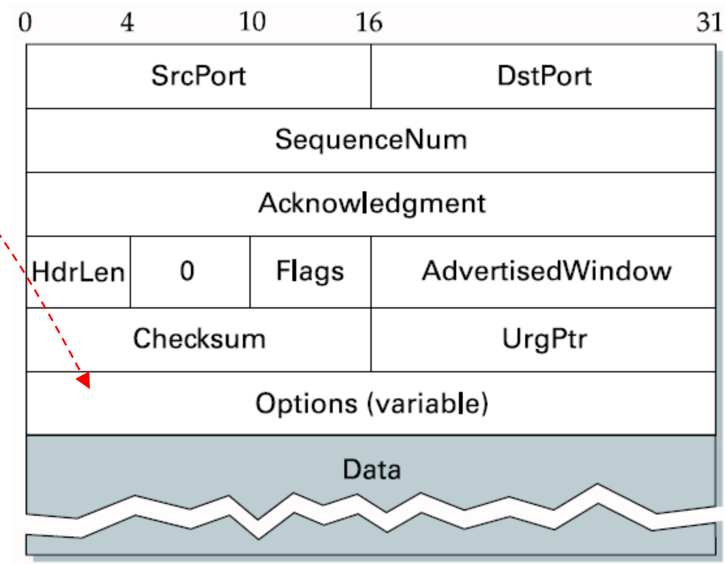
TCP: Can Keep Pipe Full?

Bandwidth	Delay × Bandwidth Product
T1 (1.5 Mbps)	18 KB
Ethernet (10 Mbps)	122 KB
T3 (45 Mbps)	549 KB
Fast Ethernet (100 Mbps)	1.2 MB
OC-3 (155 Mbps)	1.8 MB
OC-12 (622 Mbps)	7.4 MB
OC-48 (2.5 Gbps)	29.6 MB

Required window size for 100-ms RTT.

Solution: TCP Extensions

- Implemented as header options
- Store timestamp in outgoing segments → measure RTT
- Extend sequence space with 32-bit timestamp → protected against sequence number wrap-around
- Shift (scale) advertised window → keep the pipe full
- Selective acknowledgement (SAC) → acknowledge any additional (out-of-order) blocks of received data



TCP Extensions for High Performance
http://tools.ietf.org/html/rfc1323

Summary

- User Datagram Protocol
  - Multiplexer/Demultiplexer for IP
- Transmission Control Protocol
  - Reliable Byte Stream
    - Connection-oriented
      - Connection establishment
      - Connection termination
    - Automatics Repeated-Request: ACKs and NACKs
    - Flow-control
    - Timeout value estimation
    - Extensions
- *Congestion control (another classs?)*