Ch. 2 (contd.) Applications and Layered Architectures
Ch. 8 TCP/IP

TCP

- Segment format
- Ports and multiplexing/demultiplexing
- Connection management
- Window management; flow and congestion control
- Timer management
- Berkeley Sockets

Transport Protocols

Three important protocols:

1. UDP unreliable, connection-less
2. TCP reliable, connection-oriented we will focus on this
3. RPCs

Some Message Transport Requirements

If required, a transport protocol must be available to

- Guarantee message delivery
- Deliver messages in-order
- Identify and eliminate message duplicates
- Deliver messages regardless of size
- Provide sender-receiver sync
- Implement flow control (recvr → sender)
- Support multiple applications simultaneously on host

A transport protocol ***need not do all*** of those things

We merely ask that, if required, a transport protocol be available to do at least some of those things, and that ..

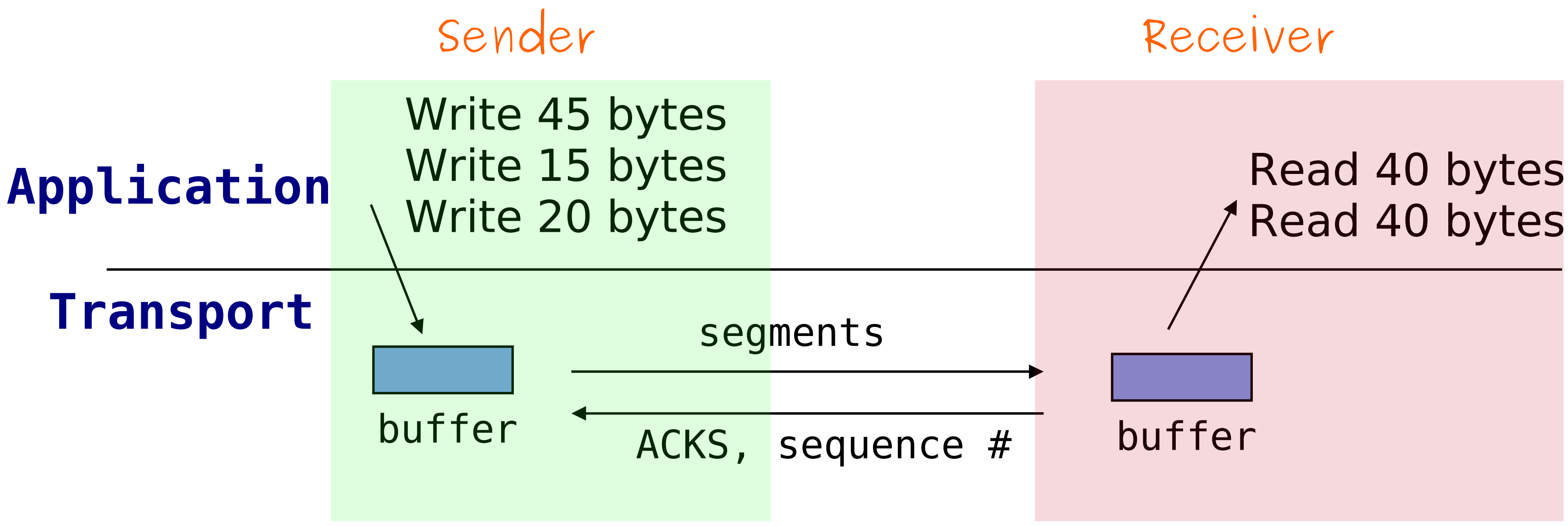
A suite of transport protocols *cover all* of those things *in the aggregate*

The Transmission Control Protocol (TCP)

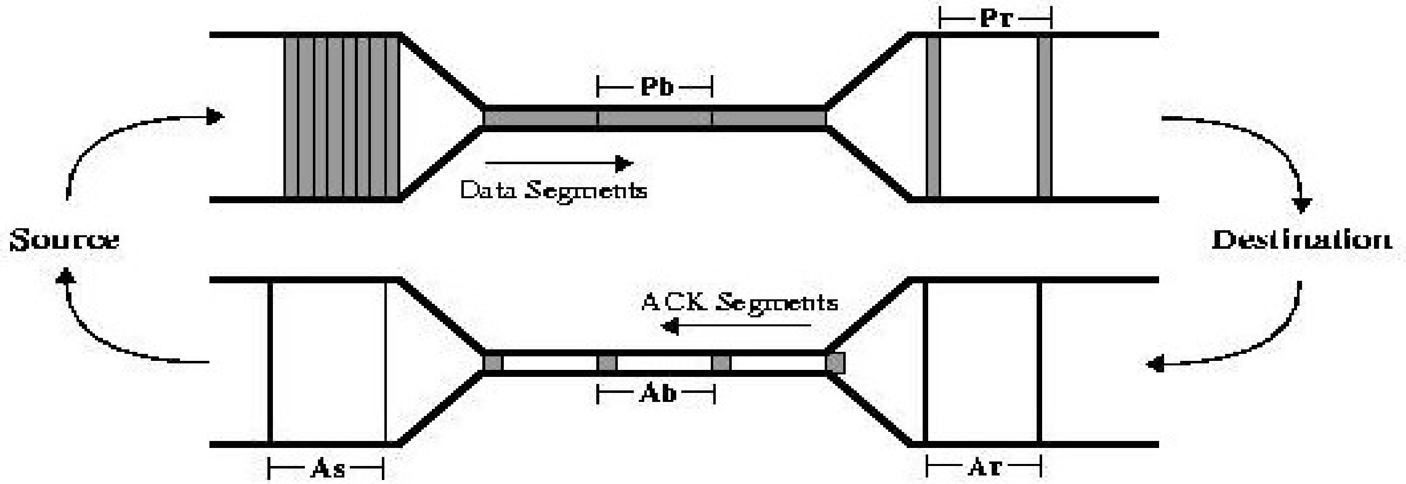
- TCP is *connection-oriented* transport layer protocol wherein
  - Sender & Receiver
    - first establish connection
      - set initial sequence numbers, sliding windows, timers
    - then, transfer data; use error recovery (by re-txing, if reqd.)
    - finally, tear down connection when done
  - You get *full-duplex*, *reliable byte stream service* between two processes in two computers *anywhere*
  - Sequence numbers track transmitted & received bytes
  - Error detection and retransmission allows recovery from transmission/reception errors
  - Sliding windows with feedback, used for flow & congestion control

TCP: Reliable Byte-Stream Service

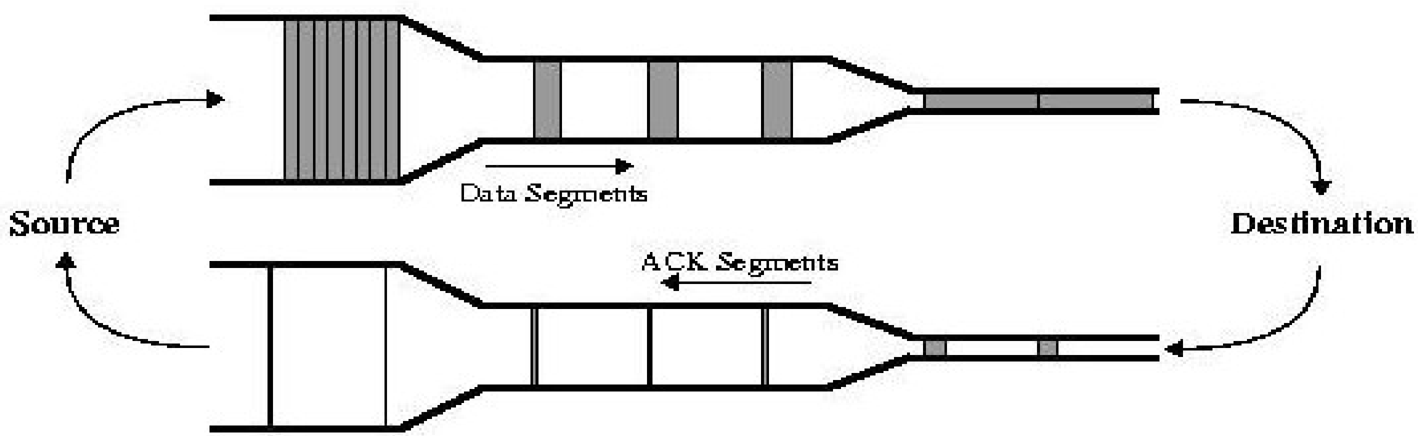
- Stream Data Transfer
  - transfers contiguous stream of bytes across network, with no indication of boundaries
  - groups bytes into segments (segments = packets)
  - transmits segments as convenient (Push, if needed)
- Reliability
  - error control mechanism



TCP: Congestion & Flow Control



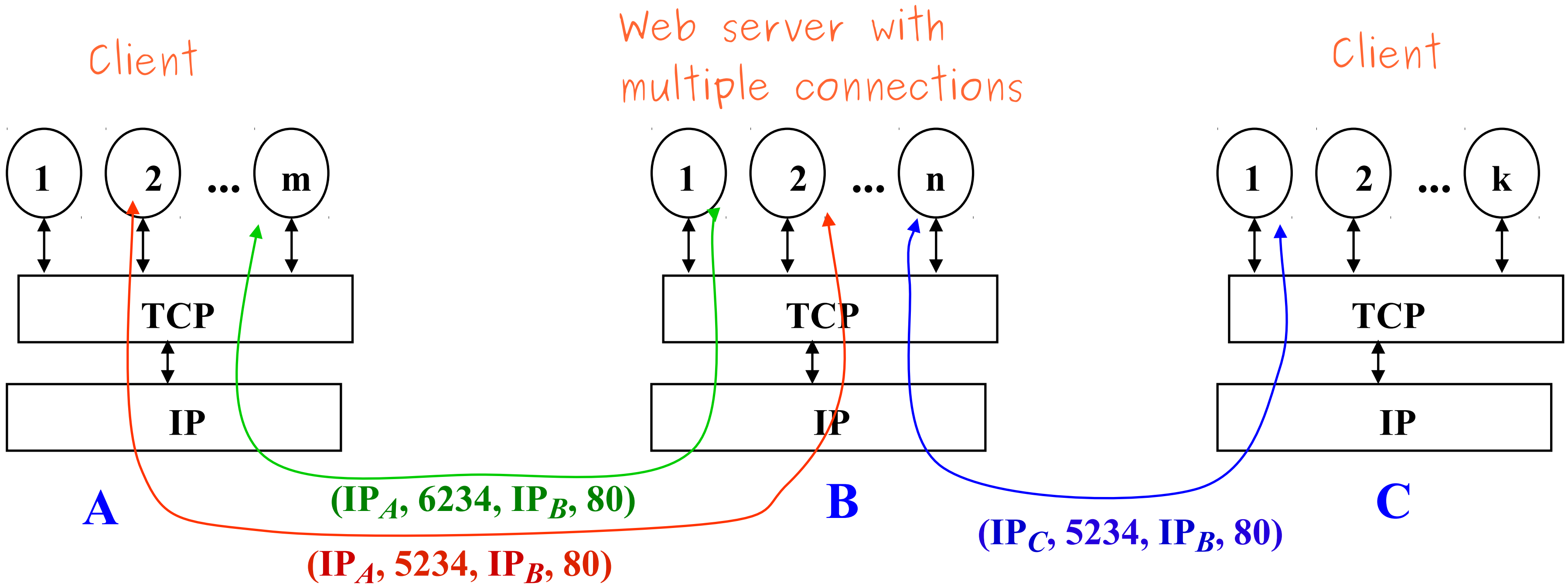
(a) Flow determined by Network Congestion



(b) Flow determined by Destination System

TCP: Multiplexing / Demultiplexing Connections

- TCP connection specified uniquely by
  - **(src IP addr, src port #, dest IP addr, dest port #)**
  - *above 4-tuple is called TCP's demultiplexing key*
  - *Caution! connection may have more than 1 incarnation*
- Multiplexing allows multiple end-to-end apps. simultaneously
- Arriving segment handed to appropriate app. based on connection 4-tuple



TCP: Ports & Port Numbers

- TCP Ports are **virtual** (not physical)
- Router port = router's physical interface

- Server processes (apps) *listen* on ports,
- Client processes (apps) *send* messages to ports

Question: How does client know server's port number?
Answer: Server processes (daemons) use *well-known ports* (*i.e. port numbers less than 1024*)

Examples:

App Process		Port #
TCP	telnet	23
TCP	ssh	22
TCP	ftp	21
TCP	http	80
UDP	DNS	53
UDP	talk	517

For a more complete list, see:
`http://www.networksorcery.com/enp/protocol/ip/ports00000.htm`

etc.

TCP: Ports & Port Numbers (contd.)

Question: What if the ports are not so *well known*?

Answer: Sometimes, server process uses non-std port, or uses different port number on each start up. Then, it registers its port # with a port-mapper service. The port-mapper service listens on a *well-known port* (UDP or TCP port number **111**)

- client contacts port-mapper to get server's port #
- port-mapper tells client server's port #

Question: How does server know client's port number?

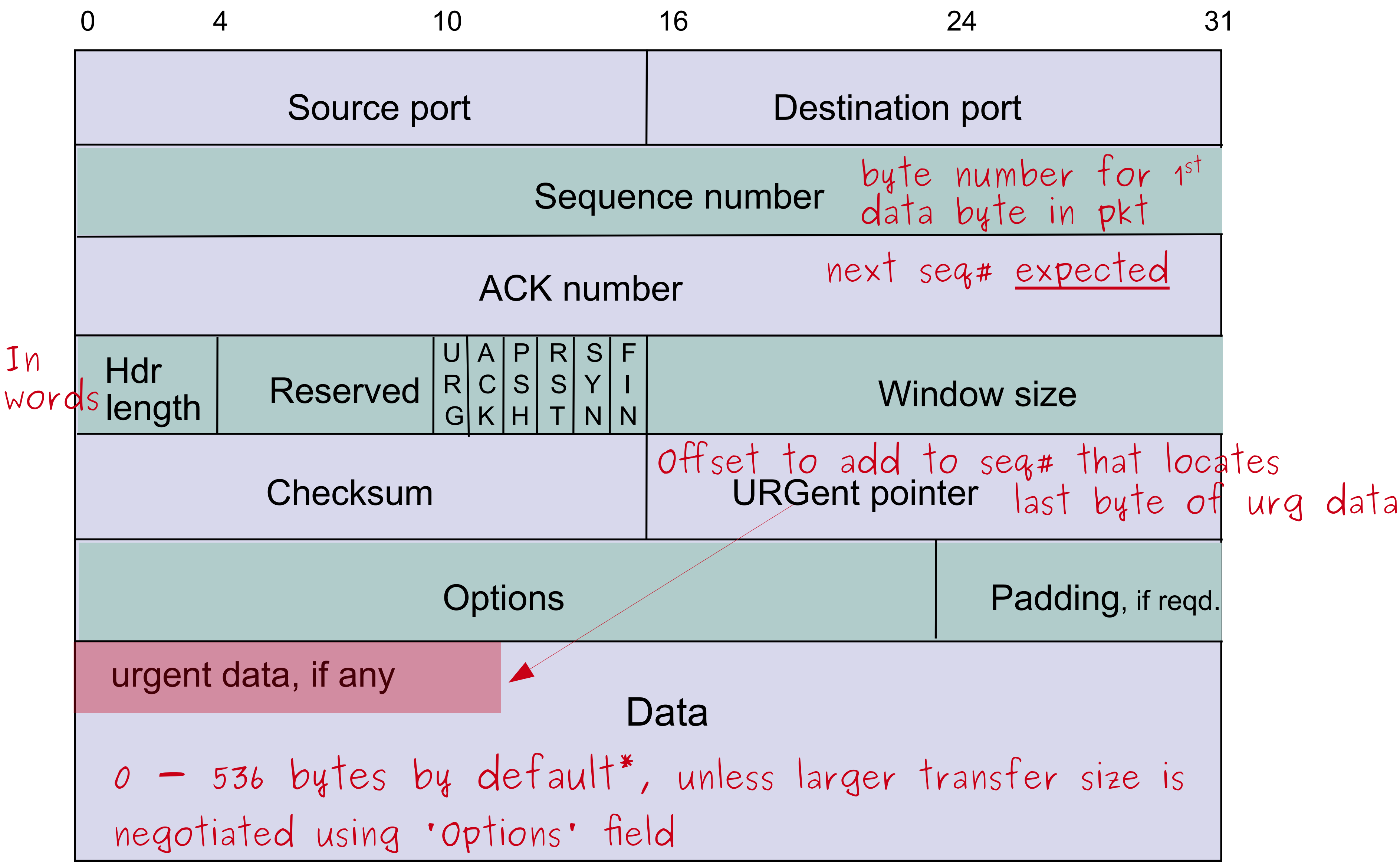
Answer: Client's port number is in client's service request (UDP or TCP packet header) sent to server

- **Port Implementation:**

  Incoming packets are put in finite-buffer FIFO queues

  - pkt drops are possible
  - App process blocks if it tries to read empty queue

TCP: Segment Format (Header & Data)



*IP layers are reqd to carry at least 576 bytes of IP packet ***without*** fragmentation

TCP Header Fields Explained

(refer to figure on previous page)

Port Numbers

- A socket identifies a connection endpoint
  - IP address + port #
- Connection is specified by a *socket pair*

Acknowledgement Number

- SN of next byte expected by receiver
- Acknowledges that all prior bytes in stream have been received correctly
- Valid if ACK flag is set

Sequence Number

- Byte count
- First byte in segment
- 32 bits long
- $0 \leq SN \leq 2^{32}-1$
- Initial sequence number is selected during connection setup

Hdr Length

- 4 bits long
- Length of header in multiples of 32-bit words
- Minimum hdr length = 20 bytes, max = 60 bytes

TCP Header Fields Explained (contd.)

Control

- 6 flags, 1 bit per flag
- URG: urgent pointer flag
  - Urgent message end = SN + **urgent pointer**
- ACK: ACK flag
- PSH: override TCP buffering
- RST: reset connection
  - Upon receipt of RST, connection is terminated and application layer notified
- SYN: establish connection
- FIN: close connection

Window Size

- 16 bits to advertise window size
- Used for **flow control**
- Sender will accept bytes with SN from ACK to ACK + window
- Maximum window size is 65535 bytes

Home reading

TCP Checksum – look this up

- Internet checksum method
- TCP pseudoheader + TCP segment

TCP Header Fields Explained (Contd.)

Checksum

<table>
<tr><td>0</td><td>8</td><td>16</td><td>31</td></tr>
<tr><td colspan="4">Source IP address</td></tr>
<tr><td colspan="4">Destination IP address</td></tr>
<tr><td>0 0 0 0 0 0 0 0</td><td>Protocol = 6</td><td colspan="2">TCP segment length</td></tr>
</table>

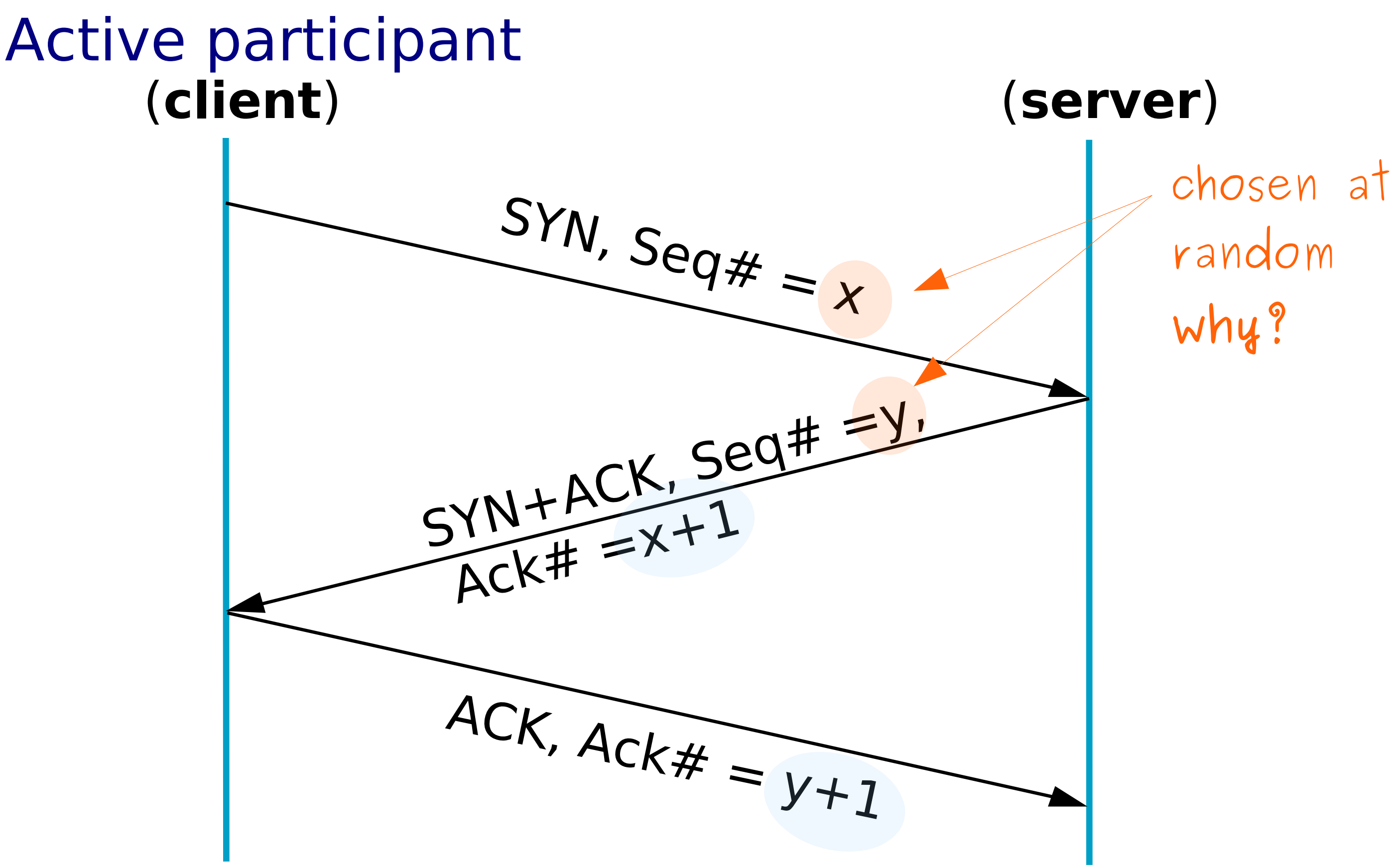
TCP pseudo-header

NOTE: *UDP and TCP use similar checksum computation. <u>Read on your own</u>.*

Options

- Variable length
- **NOP** (No Operation) option to pad TCP header to multiple of 32 bits; spacer betwn options
- **Time stamp** option for RTT measurements
- Maximum Segment Size **(MSS)** option specifies largest segment size acceptable to receiver
- **Window Scale** option increases TCP window size from 16 to 32 bits

TCP: Connection Establishment and Termination



A 3-way handshake for connection establishment

TCP: Connection Establishment and Termination (contd.)

Two issues to consider:

1.1 Why exchange seq #s at connection establishment?
(i.e. Why not simply start from seq# 0 on each new connection?)

1.2. Why not tie initial sequence number to local timer?

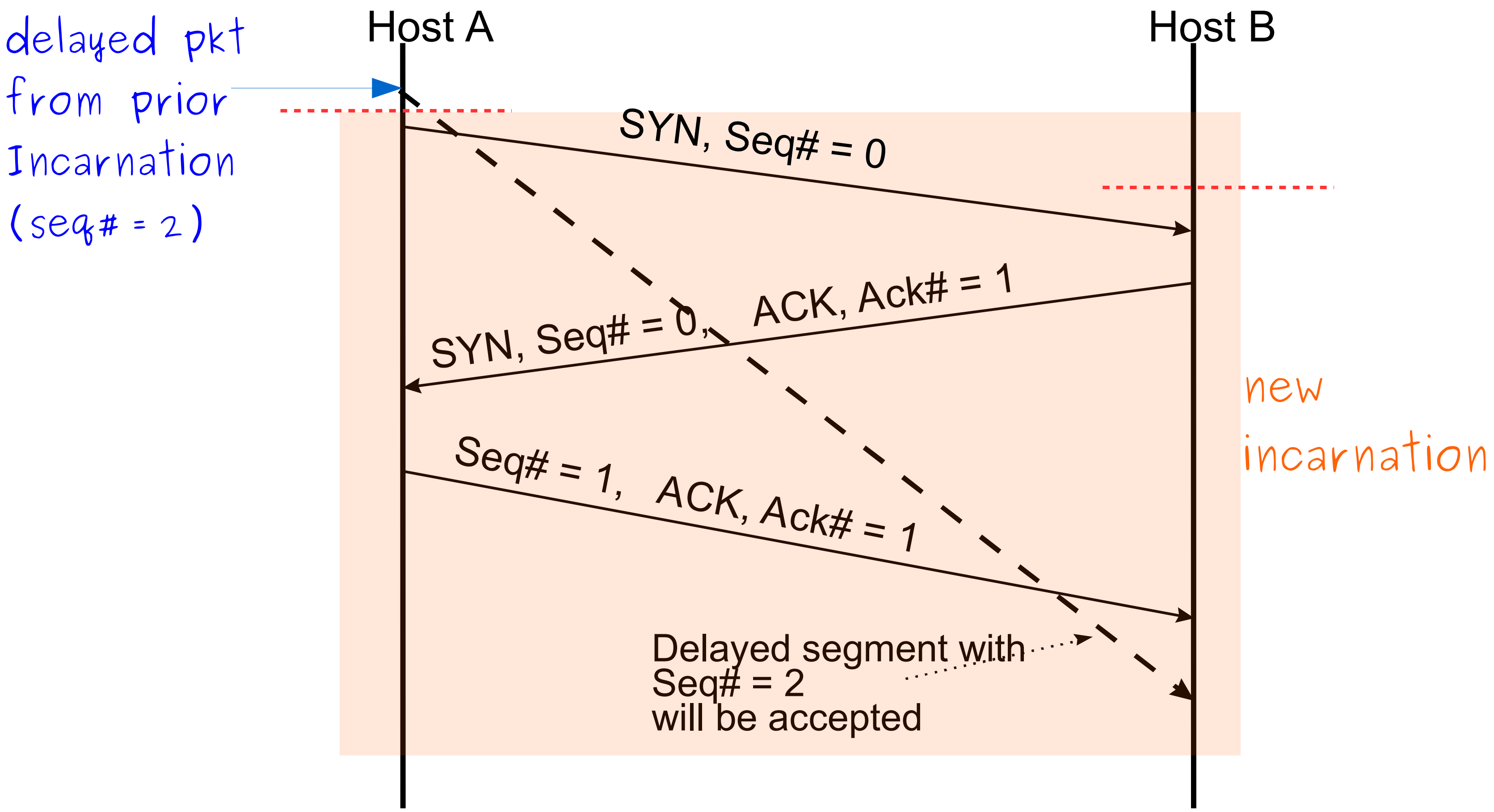
See: http://wiki.cas.mcmaster.ca/index.php/The_Mitnick_attack

2. What about the 'two armies' problem?
(during connection tear-down phase)

TCP: Connection Establishment and Termination (contd.)

1.1 Why not simply start from seq# 0 on each new connection?

Because, this may happen:



TCP: Connection Establishment and Termination (contd.)

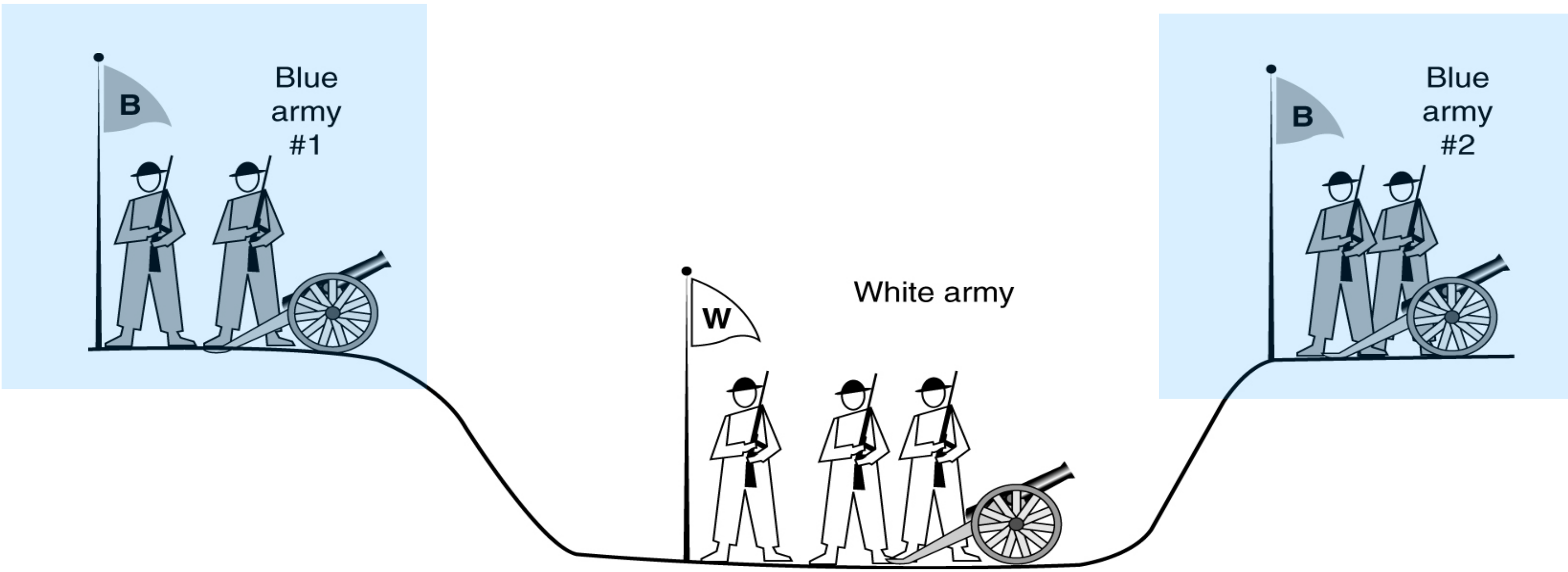
Sequence Numbers

- *Select initial sequence numbers (ISN)* to avoid overlap with sequence numbers of prior connections
- Use ~~local clock~~ to select ISN sequence number
- Time for sequence nos. to wrap around should be greater than the maximum lifetime of a segment (**MSL**);
  Typically MSL=120 seconds
  - High bandwidth connections pose a problem
  - $2^n > \underbrace{2 * \text{MSL sec} * R \text{ bytes/sec}}$

total sequence nos.

Bytes transmitted in 2MSL period

TCP: Connection Termination



2. The 2 armies problem ("Computer Networks, 5ed", A. Tanenbaum):
   a connection termination conundrum

Review: Sliding Window Protocol with Finite Seq Numbers

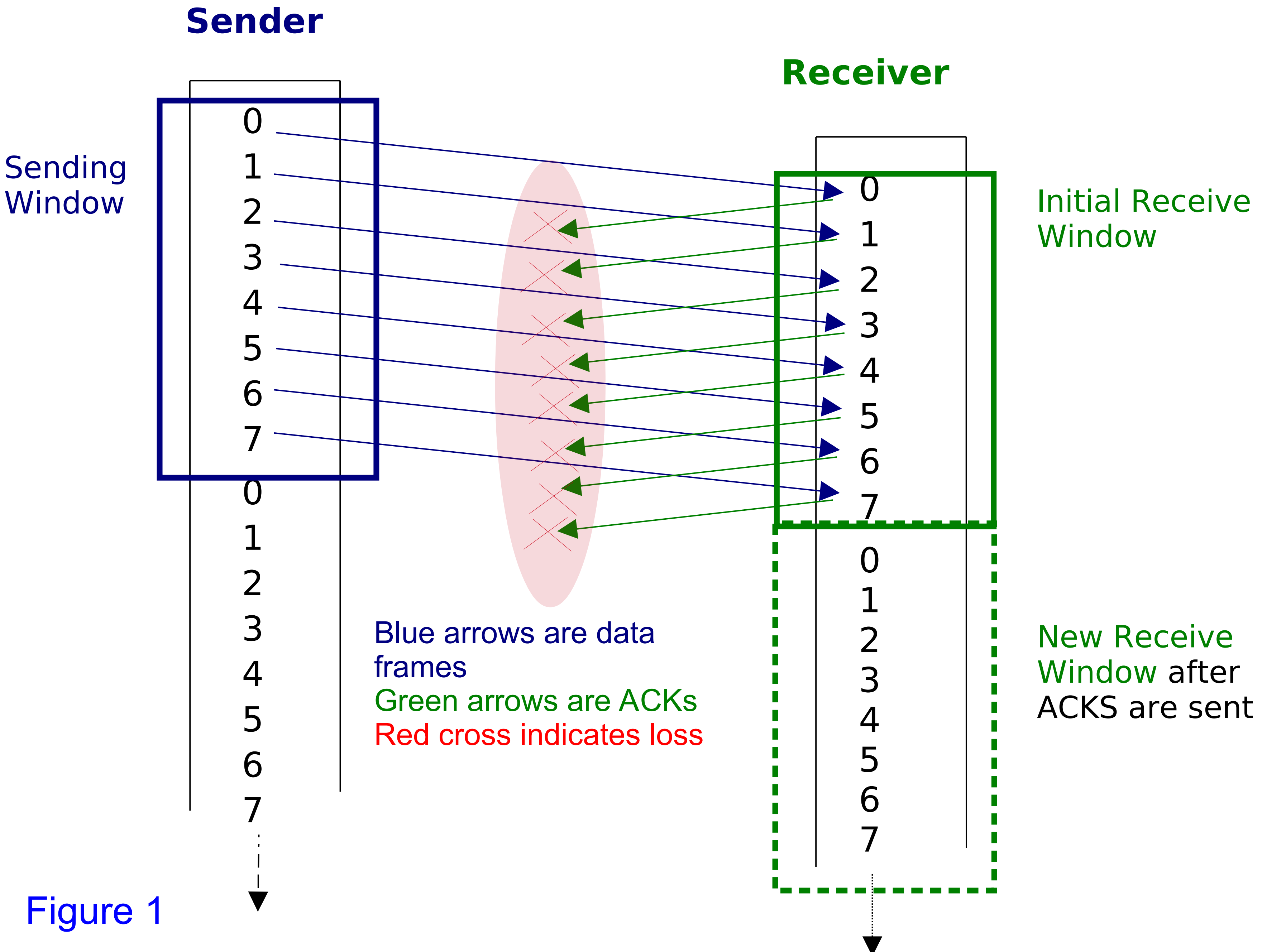


Figure 1

Review: Sliding Window Protocol with Finite Seq Numbers

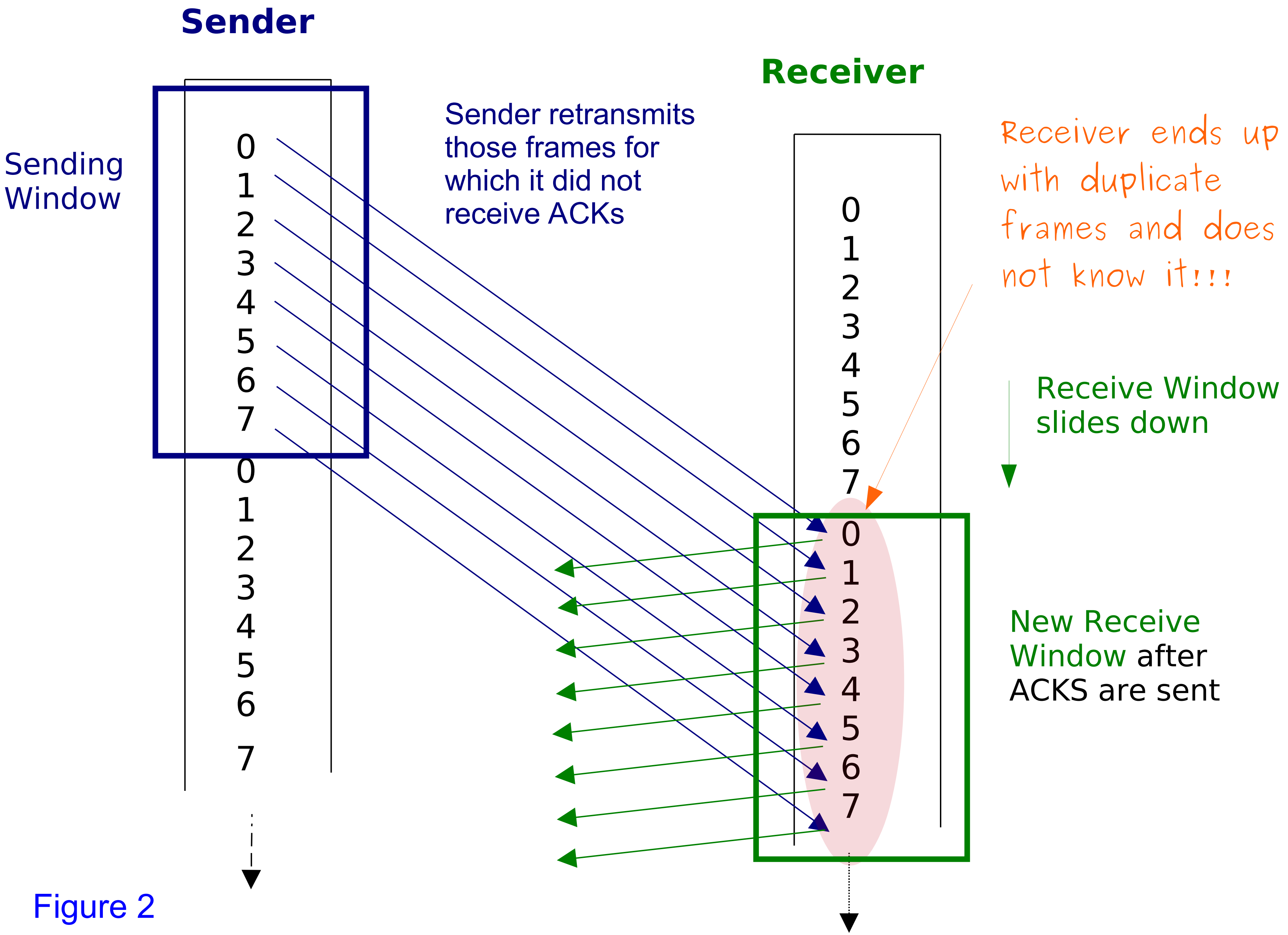


Figure 2

TCP: More On Sequence Numbers

The sliding window rule:

Receiver's successive windows must never overlap!

Let us see how TCP fares here ..

TCP's **`Sequence Number`** field ⇒ 32 bits
TCP's **`Advertised Window`** field ⇒ 16 bits

Therefore, no overlap possible between 2 successive advertised windows

But is that enough to prevent **`Sequence Number`** *wraparound problems that may arise?*

TCP: Sequence Number Wraparound

Consider from S's perspective..

1. S sends segment with **`sequence number`** ***x*** to R
2. S times out on ***x*** (no ACK recvd) .. ***x*** *is delayed in transit, but neither S nor R know that!*
3. S retransmits ***x*** to R

From R's perspective:

1. R recvs retransmitted ***x*** from S, sends ACK, then rolls down its window
2. R doesn't know that original copy of x is floating around on the net
3. Now, *some* time later seq nums at S roll over, S sends new (rolled over) ***x*** to R. Then, out of the blue ...
5. R receives very first copy of ***x*** sent by S, before it receives new ***x***

TCP: Sequence Number Wraparound (contd.)

Bandwidth	**Time until Wraparound** (of 32-bit sequence number)
Cable (10 Mbps)	57 minutes
Ethernet (100 Mbps)	6 minutes
STS-12 (622 Mbps)	55 seconds (!) .. *less than MSL*
STS-48 (2.4 Gbps)	14 seconds (!) .. *less than MSL*

TCP: Sequence Number Wraparound (contd.)

Solution?

Sending TCP process

- inserts 32-bit timestamp in **`options`** field of segment

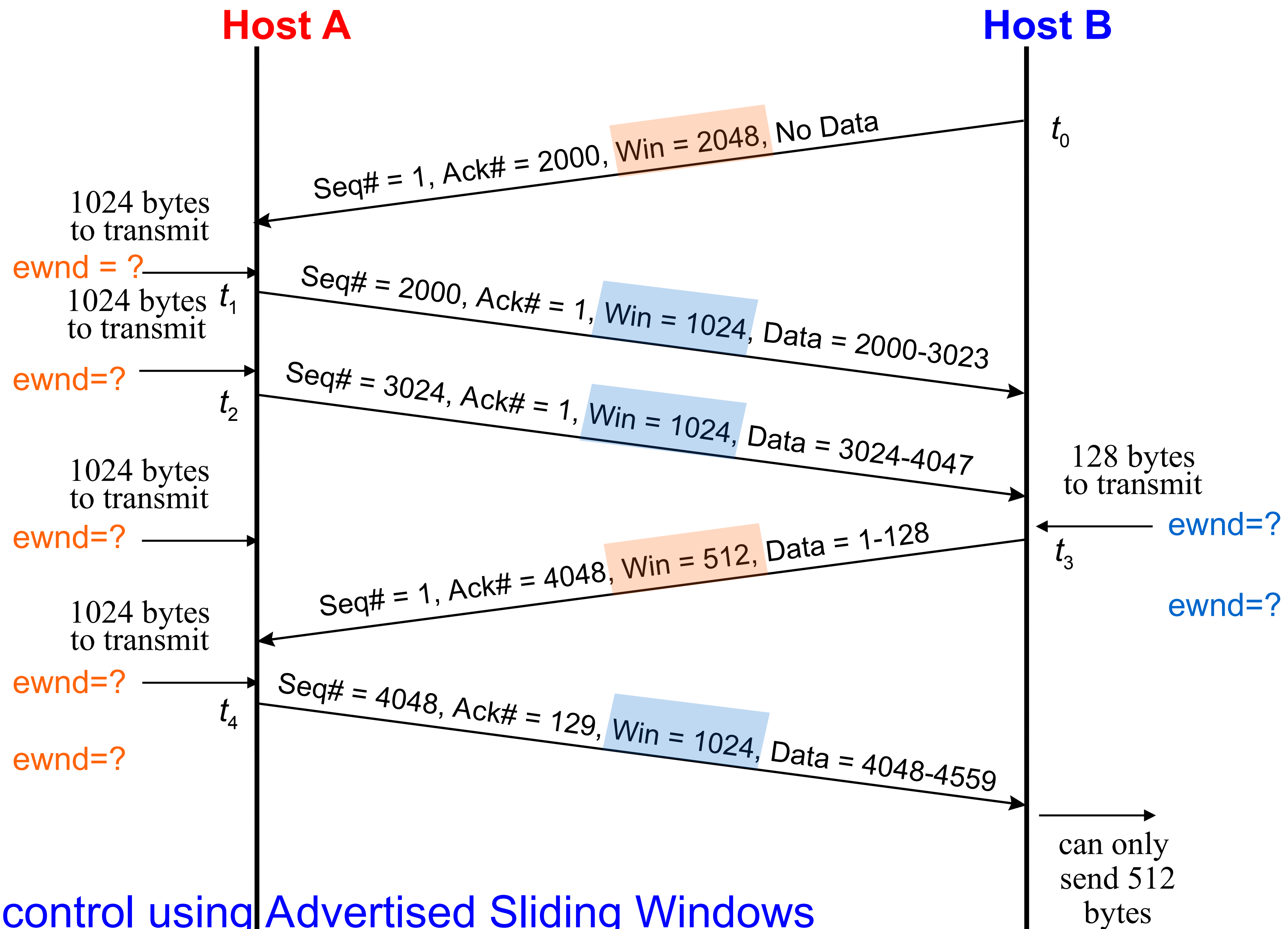
Receiving TCP process

- checks timestamp as well as sequence number
- if segment ***n*** with timestamp smaller than segment ***n-i*** arrives, segment ***n*** is discarded

  Why?

  Because segment with lower valued timestamp is a delayed segment that just appeared at recvr

TCP: Advertised & Effective** Windows



Flow control using Advertised Sliding Windows

*** definition of effective window will be updated later when we study congestion*

TCP: Size of Advertised Window

Also, recall ...

- *To increase utilization and throughput, window size must be at least = ?*

Now, let us run the numbers ...

Advertised Window (**awnd**) field in TCP header is 16 bits
awnd is expressed as # of bytes
Therefore, **awnd** size is 2^{16} bytes (max)
= 64 KB (max)

Also, typically, RTT for cross-country segment = 100 ms

Now, refer to table on next slide

TCP: Size of Advertised Window (contd.)

Bandwidth	**RTTDelay-Bandwidth Product** (when RTT = 100ms)
Slow DSL (1.5 Mbps)	18 KB
Cable (10 Mbps)	122 KB (!) .. >> Max. AdvertisedWindow
Ethernet (100 Mbps)	1.2 MB (!) .. >> Max. AdvertisedWindow
STS-12 (622 Mbps)	7.4 MB (!) .. >> Max. AdvertisedWindow
STS-24 (2.4 Gbps)	29.6 MB (!) .. >> Max. AdvertisedWindow

So, for better line utilization, receiver must advertise a window larger than 64KB

How?

Use **`options`** field in TCP header to provide scaling (multiplication) factor for **`awnd`** size field

TCP: A Tiny Problem

One problem ...

- Suppose recvr's **AdvertisedWindow** (**`awnd`**) reaches **`0`**
  - ➔ So, sender stops sending
    Now, neither sender, nor receiver are sending anything to each other *deadlock?*

- Suppose now:

  Recvr's application reads from receive buffer
  (and therefore frees up 'receive buffer' space)

  How does sender know new size of `awnd` ?

TCP: Solution to the Tiny Problem

Smart sender/dumb recvr rule...

When recvr's **`awnd`** reaches **`0`**

- Sender sends segment with 1 byte of data once every ***x*** secs
- Receiver sends ACK segment in response

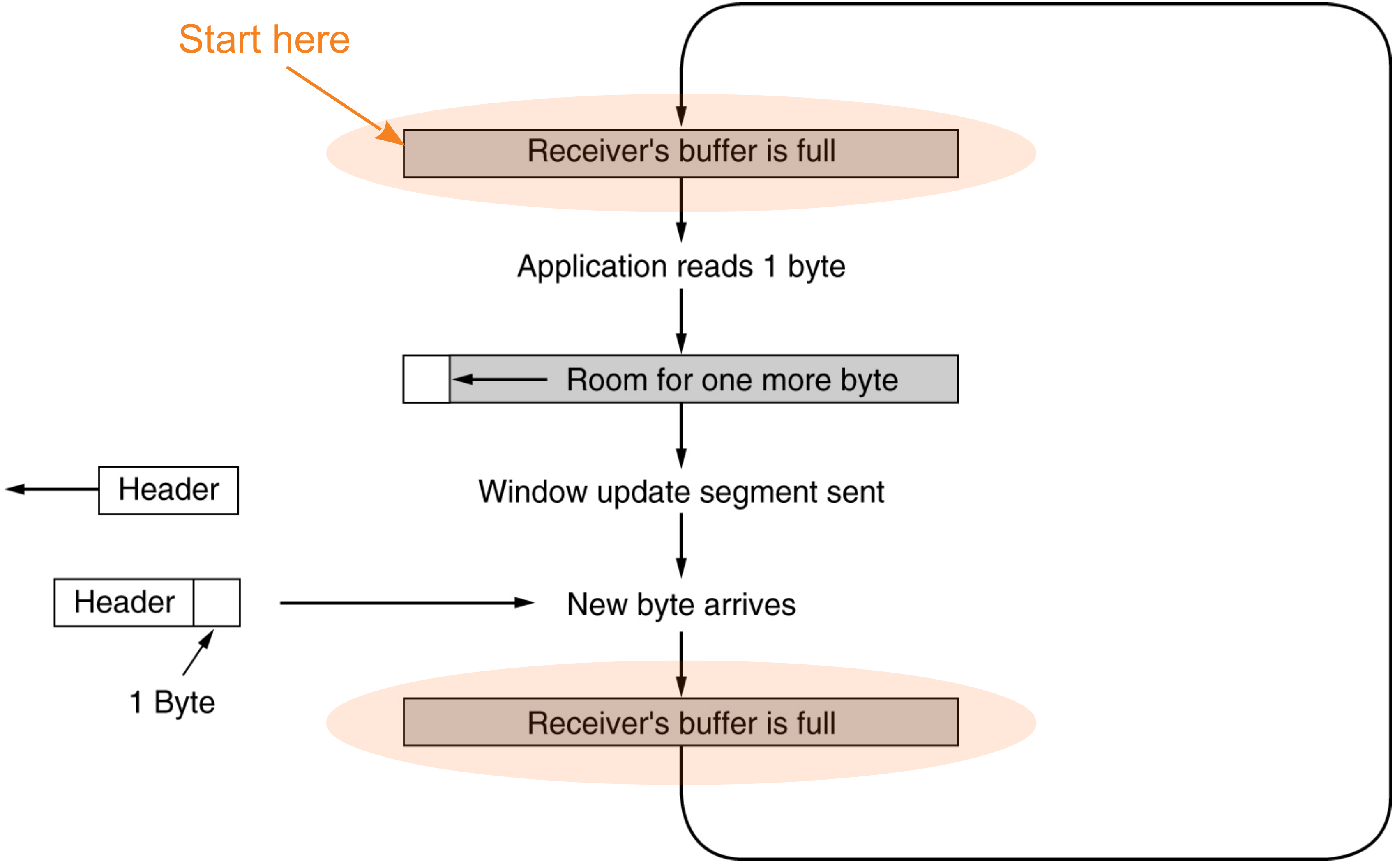
From **`ACK#`** field in TCP hdr of response segment,

- sender knows if pkt has been accepted/rejected

AND

- **`awnd`** field in TCP header of response segment tells sender the receiver's new advertised window

TCP: The *Silly Window Syndrome* (*SWS*)



Receiver's Window size stuck at 1 byte

TCP: The *Silly Window Syndrome* (contd.)

Two complementary approaches to SWS:

1. **Clark's solution** (on recvr's side)-

   Receiver may send new **`awnd`** <u>only if</u>

   a. receive buffer has available space equal to MSS** (negotiated at connection establishment time), ***or***

   b. receive buffer is half empty

2. **Nagle's algorithm**

   (= <u>self-clocking solution</u> on sender's side) .. *next slide*

**The Max Segment Size (MSS) is the maximum size of the TCP Payload (TCP, IP, MAC headers/trailers are excluded)

TCP: The *Silly Window Syndrome* (contd.)

2. Nagle's algorithm (= self-clocking solution):

when application produces data to send

  if (**data-to-send** ≥ **MSS** and **EffectiveWindow** ≥ **MSS**)

    then send a full segment

  else

    if there is unACKed data *in flight*

      then buffer newly produced data until ACK arrives

    else

      send all new data now

Note:

1. *Algorithm adjusts to RTT*

- *Short RTT - send frequently at low efficiency*
- *Long RTT - send less frequently at greater efficiency*

2. *If application cannot afford delay introduced by above algorithm, set **TCP_NODELAY** option when establishing socket .. this disables Nagle's algorithm altogether*

TCP: The *Silly Window Syndrome* (contd.)

- Question to ask yourself:

  How does an application like telnet or ssh that generates data at a low rate, work with Nagle's algorithm?

- Answer:
  - *Trace through Nagle's algorithm to see how the 1st and the 2nd segments are transmitted*
  - *Two successive 'small' writes to the TCP socket cause the 2nd TCP segment to wait while the ACK to the 1st segment is pending*
  - *Causes delay of upto 500 ms!*
  - *Better to use TCP_NODELAY option to disable Nagle's algorithm*

TCP: State Transition Diagram

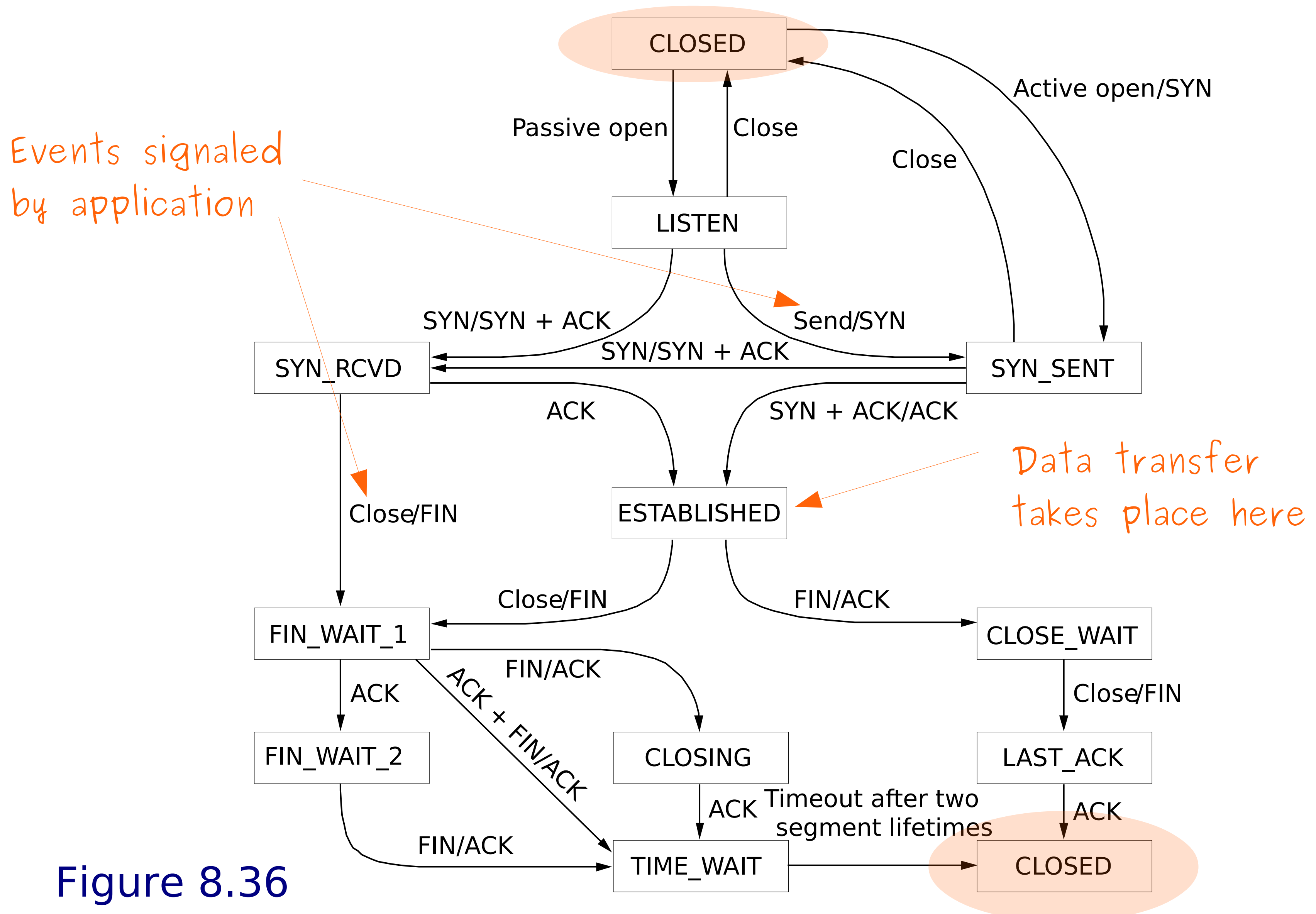
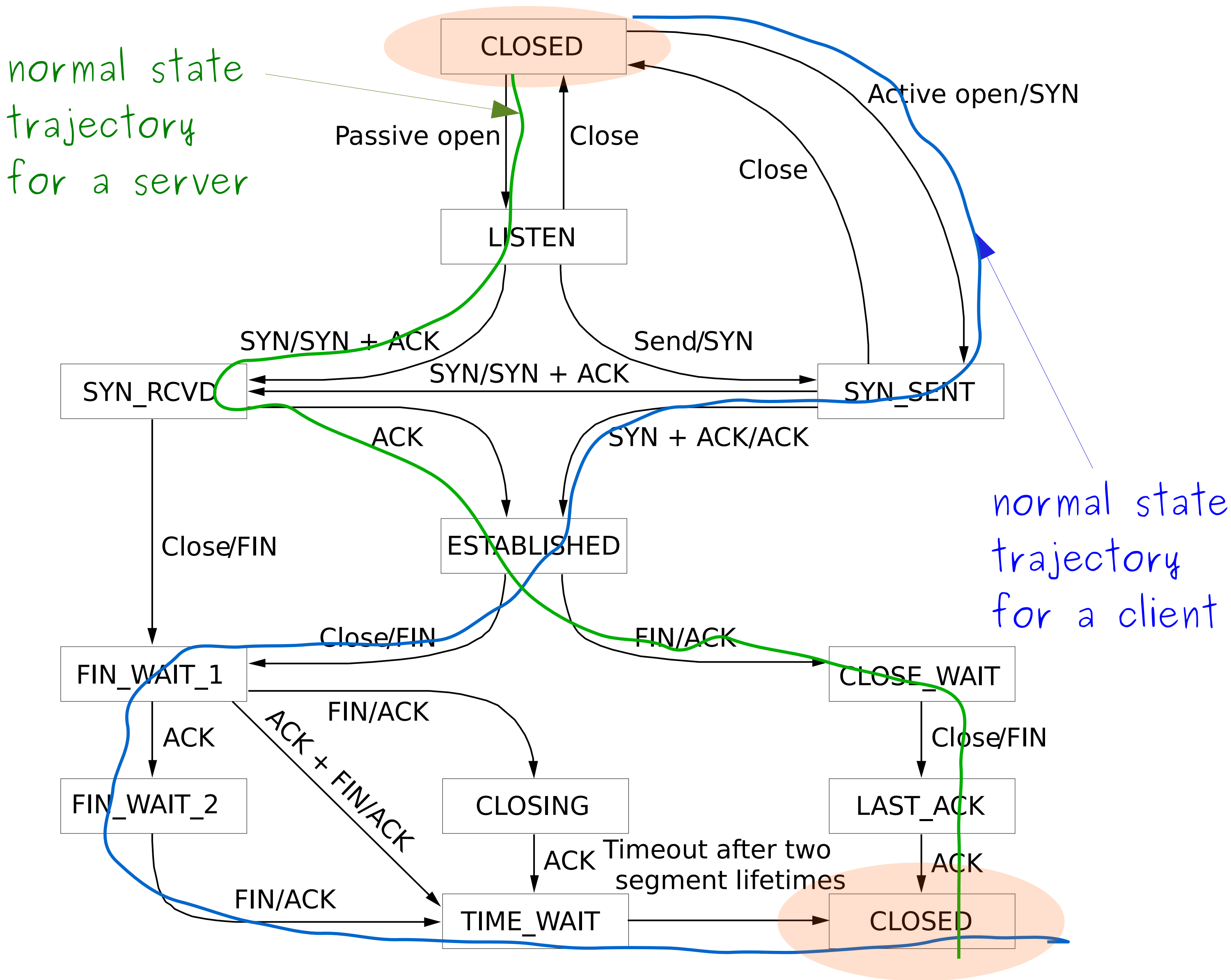


Figure 8.36

TCP: State Transition Diagram



TCP: **State Transition Diagram** (contd.)

Some issues to consider:

1. What if client's (active initiator's) ACK to server (from SYN_SENT state) is lost?

2. Why the transition from LISTEN to SYN_SENT?

3. What if we get stuck in a state?
   (*i.e.* what if the event that we are expecting never occurs?)

4. What if one side never sends a FIN?

5. Why is this TIME_WAIT state required?

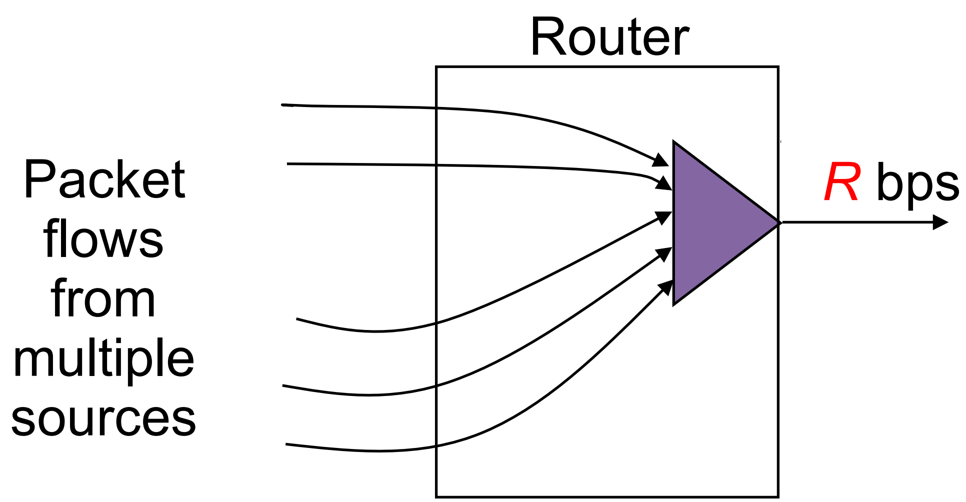
TCP: The Time Wait State

When TCP receives ACK to last FIN, TCP enters TIME_WAIT state

- Protects future incarnations of connection from delayed segments
- TIME_WAIT = 2 x MSL
- Only valid segment that can arrive while in TIME_WAIT state is FIN retransmission
  - If such segment arrives, resend ACK & restart TIME_WAIT timer
- When timer expires, close TCP connection & delete connection record

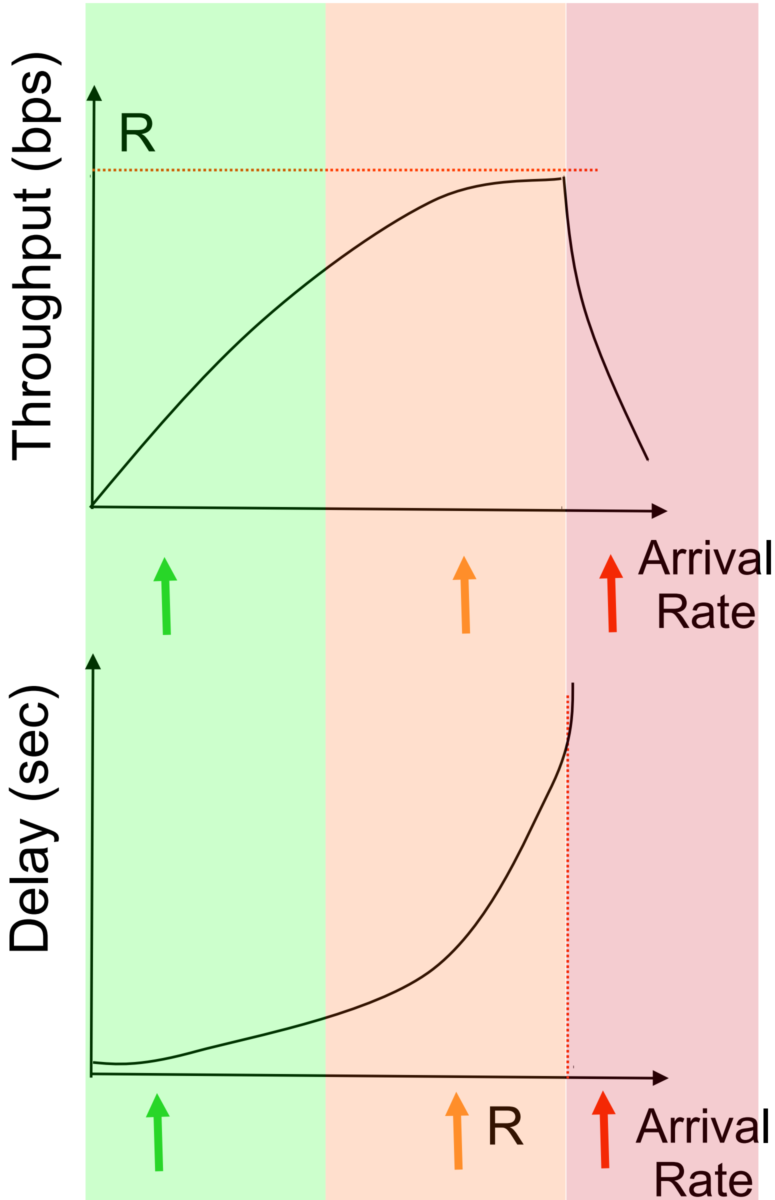
TCP: Congestion Control

- *Advertised window* size is used to prevent receiver's buffer overflow
- However, buffers at intermediate routers between source and destination may overflow ..***network congestion***!



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

Phases of Congestion



1. Light traffic

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

2. Knee (congestion onset)

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

3. Congestive collapse

- Arrival rate > *R*
- Large delays, packet loss
- Throughput drops

Window Based Congestion Control

- Desired operating point: just before knee
  - Sources must control sending rates such that aggregate traffic avoids knee region
- Every TCP sender maintains *congestion window* `cwnd` to limit congestion at intermediate routers
- **Effective window*** is `min(cwnd, awnd)` - *data_in_flight*
- Problem: source unaware of its "fair" share of available bandwidth
- Solution: <u>adapt dynamically</u> to available Bw
  - Sources probe the network by increasing `cwnd`
  - When congestion detected, sources reduce rate
  - Ideally, sources' sending rate stabilizes near desired range

** Definition of effective window updated here*

TCP: Congestion Window Dynamics

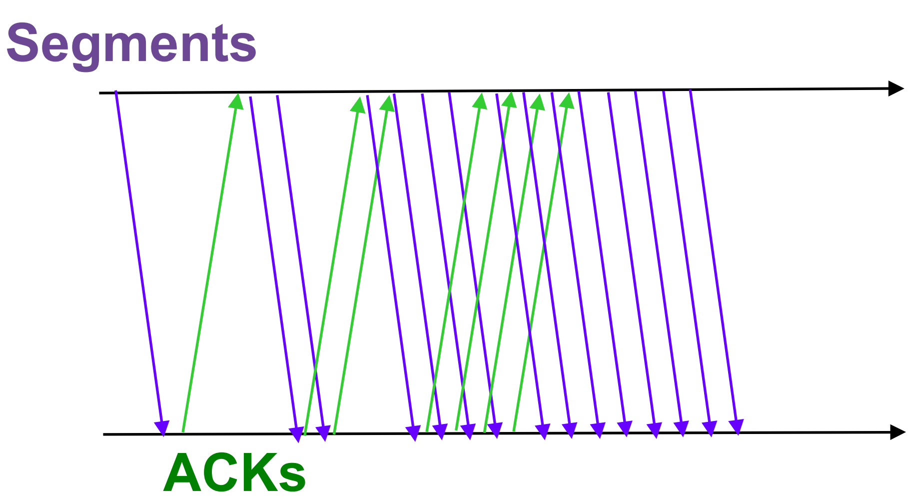
TCP changes congestion window dynamically ..

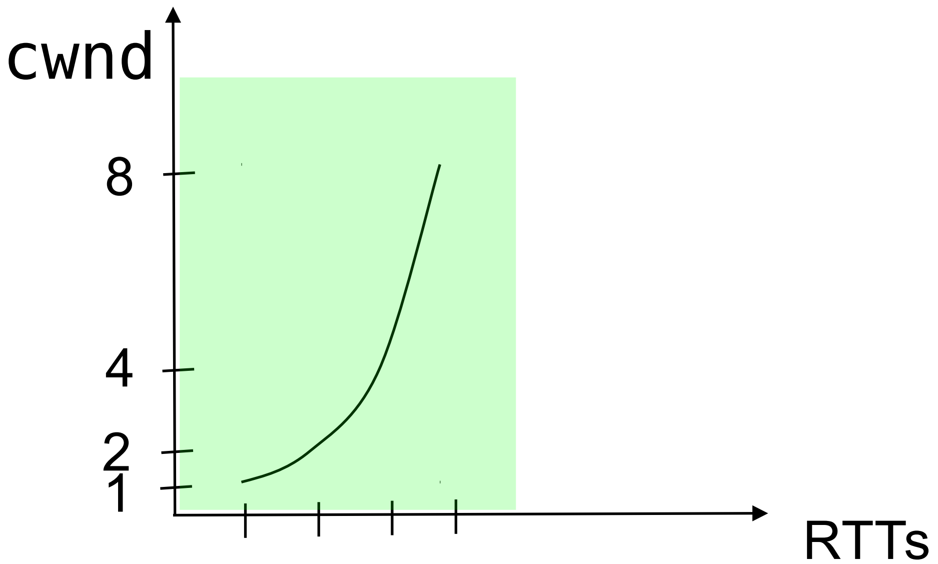
- At light traffic: each segment is ACKed quickly
  - increase `cwnd` aggresively
- At knee: segment ACKs arrive, but more slowly
  - slow down increase in `cwnd`
- At congestion: (re)transmission timeouts occur due to delays and drops ..*sender gets duplicate ACKs*
  - reduce transmission rate, then probe again

TCP: Congestion Control - Slow Start

Slow start:

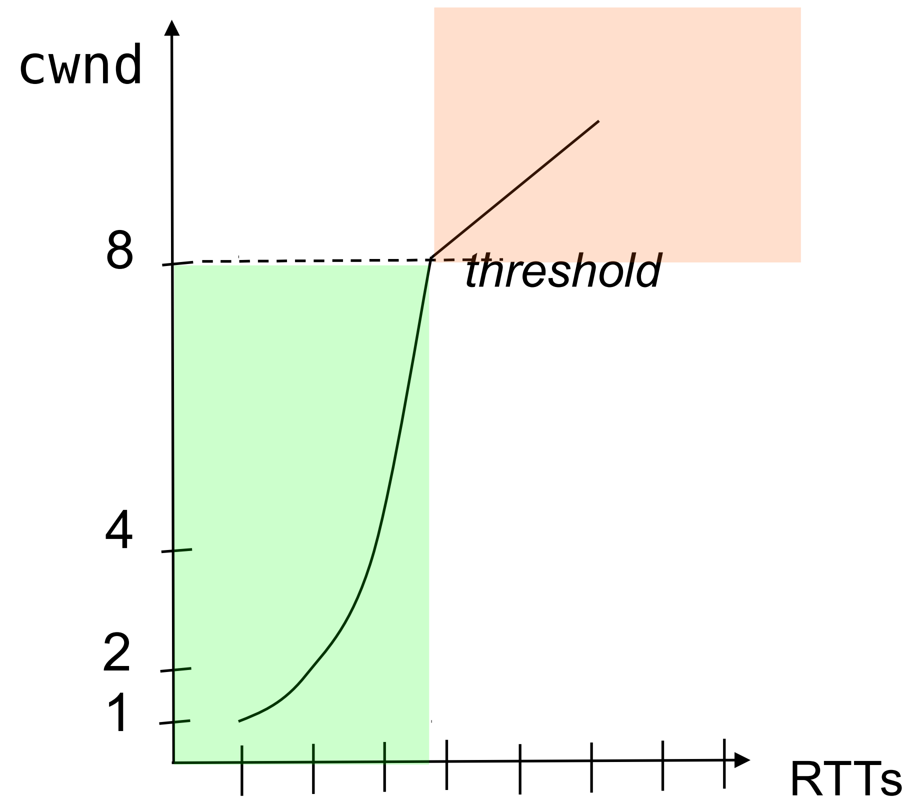
- (re)start tx with window size = 1 segment (MSS bytes)
- increase `cwnd` size by one segment each time ACK recvd
- congestion window <u>increases exponentially</u> !
- thus, not really slow at all!



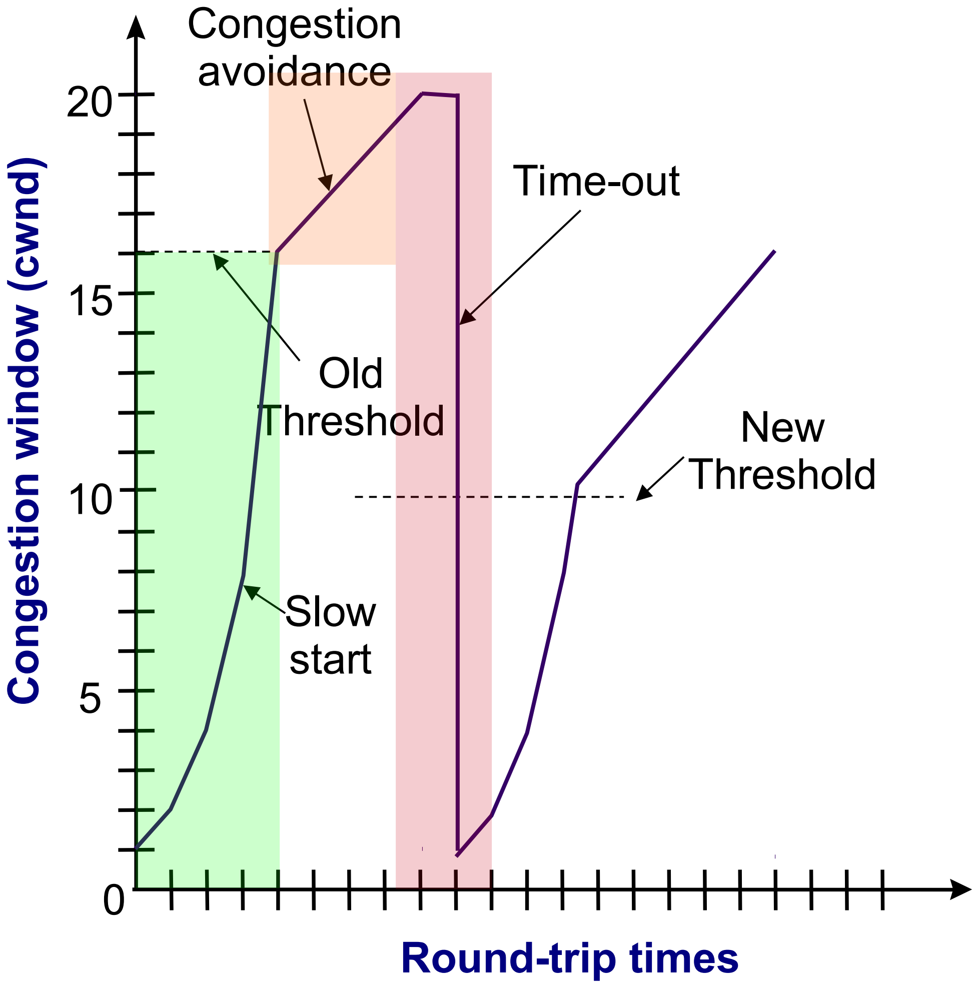


TCP: Congestion Control – Congestion Avoidance

- Algorithm progressively sets a *congestion threshold*
  - When cwnd > threshold, slow down rate of increase of cwnd
- Increase congestion window size by one segment per round-trip-time (RTT)
  - Each time an ACK arrives, cwnd 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 Recovery



Timeout or duplicate ACKs means congestion in <u>wired</u> world

Eventually `cwnd` reaches bandwidth capacity. Then ..

On multiple timeouts or duplicate ACKs (many pkt drops in seq):

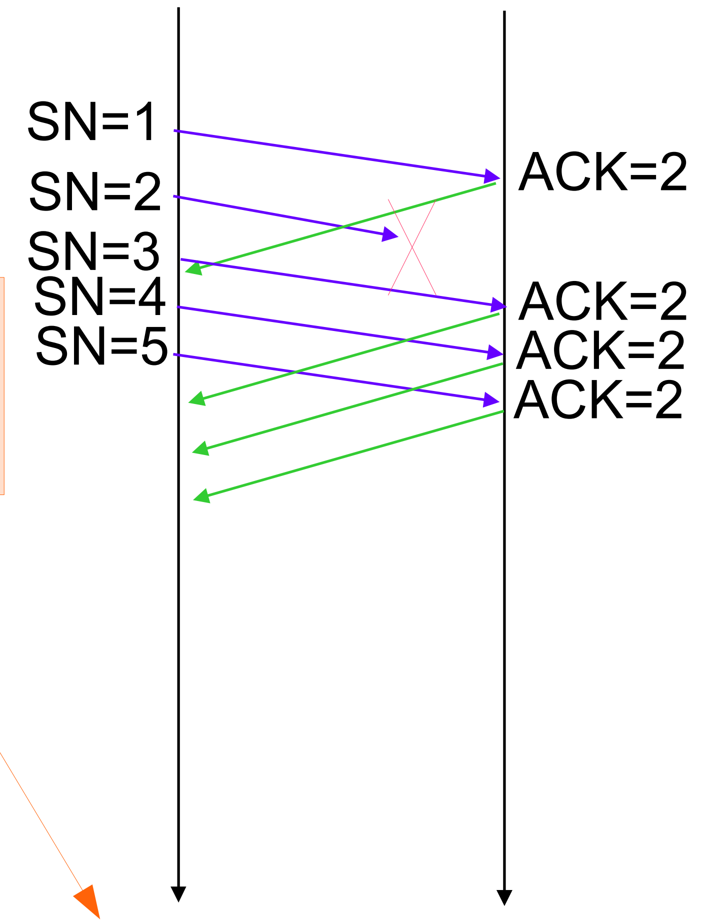
- Adjust congestion threshold* = ½ x current `cwnd`
- Reset `cwnd` to 1
- Go back to slow-start

Over several cycles, threshold converges to about ½ the max bandwidth

**During connection initialization, the threshold is set to an arbitrarily high value*

TCP: Fast Retransmit & Fast Recovery

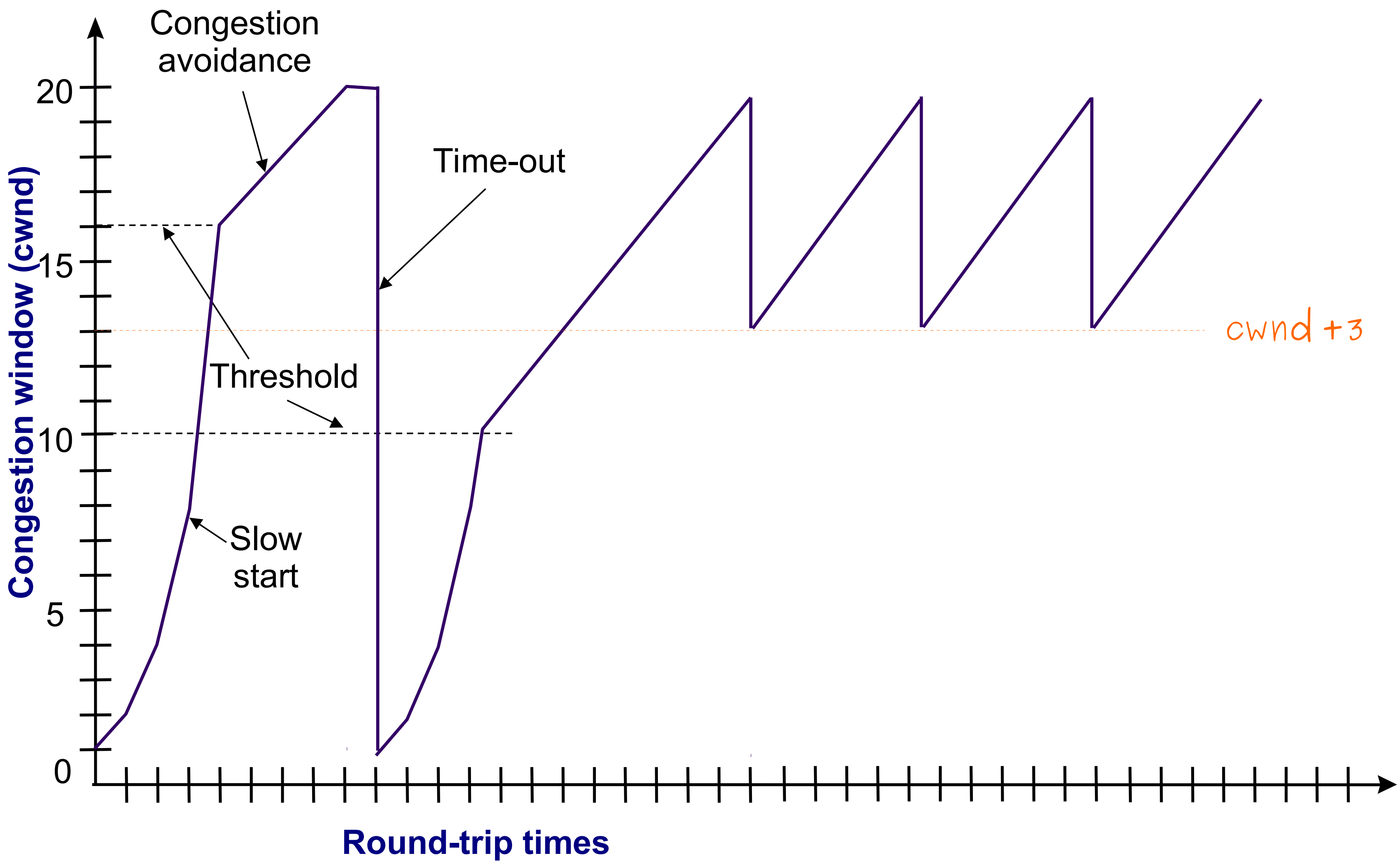
- Congestion causes multiple segment drops
- If only single segment dropped, then subsequent segments trigger duplicate ACKs before timeout
- Can avoid large decrease in `cwnd` as follows:
  - When three duplicate ACKs arrive, retransmit lost segment immediately
  - Reset congestion threshold to <u>½ `cwnd`</u>
  - Reset `cwnd` to **congestion threshold + 3** corresp. to the 3 duplicate ACKs
  - <u>Remain in congestion avoidance phase</u>
  - However if timeout occurs, reset `cwnd` to 1
  - In absence of timeouts, `cwnd` will oscillate around optimal value



Fast retransmit

Fast Recovery

TCP: Congestion Control - Fast Retransmit & Fast Recovery



TCP: Adaptive Retransmission

Issue: How to determine value of *retransmission timer*

Q: Why is this value critical?

A: Because,

- If set too high, lost pkts wait too long before being re-txed
  ... *this lowers line utilization & throughput*
- If set too low, pkts may time out and trigger re-tx unnecessarily
  ... *bandwidth wastage, low throughput*

Q: How best to determine RTT value(s)?

A: let us see..

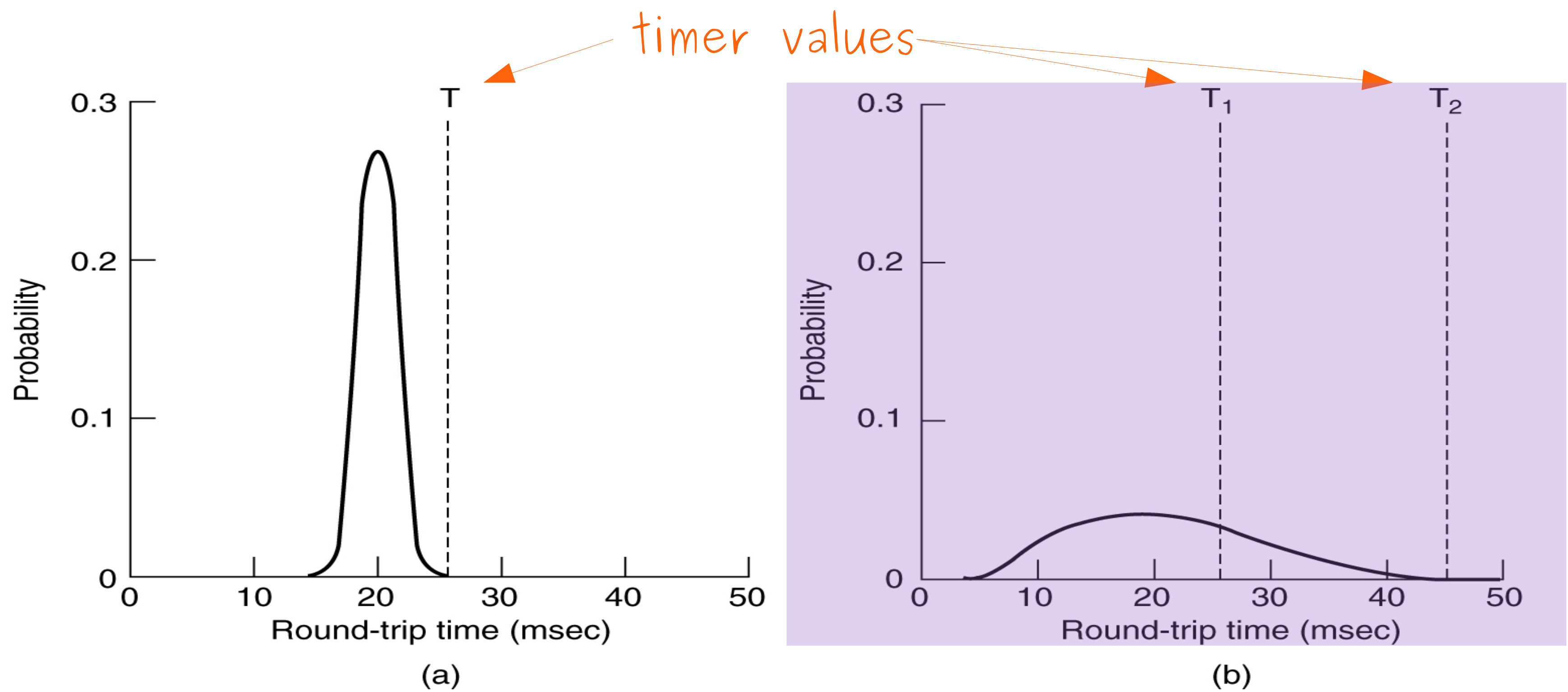
TCP: Adaptive Retransmission (contd.)

Q: How about simply calculating the mean of observed RTT values?

A: ?

TCP: Adaptive Retransmission (contd.)

What other pitfalls could we have in computing the retransmission timeout value?



(a) Probability Density of RTT (1-hop) as seen by the **data link layer**

(b) Probability Density of RTT (end-to-end) experienced by **TCP**

[*The mean RTT value is 20 ms in both the above cases; and $T_1 = T$*]

Notice the difficulty in setting the pkt timeout value for TCP

TCP: Adaptive Retransmission (contd.)
(RFC 793 - Exponential Averaging)

Estimated Round-Trip Time (ERTT):

$$\text{ERTT}(K+1) = \boldsymbol{\alpha} \times \text{ERTT}(K) + \mathbf{(1 - \boldsymbol{\alpha})} \times \text{SampleRTT}(K)$$

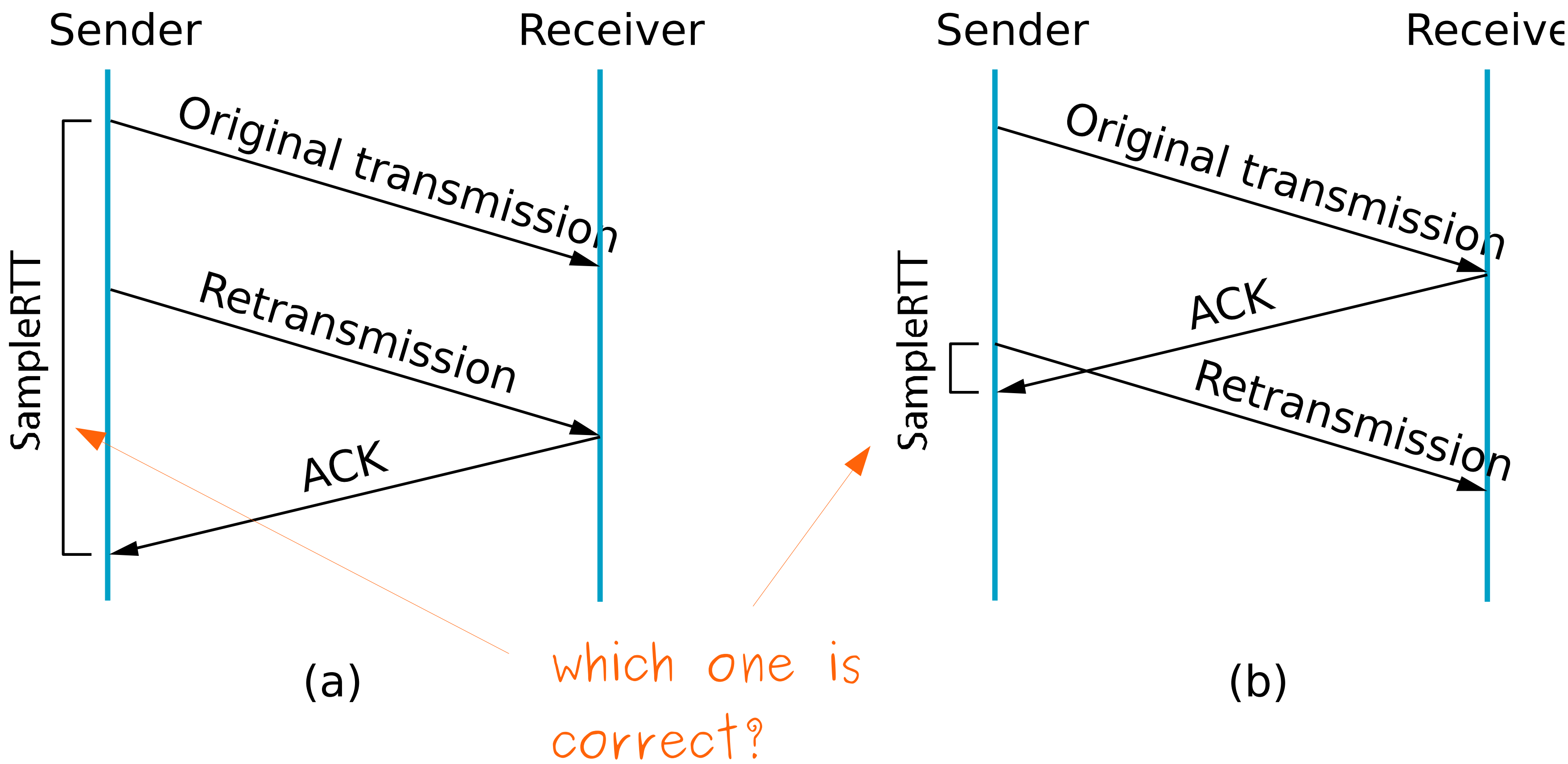
where $0 < \alpha < 1$ *(suggested value is $0.8 < \alpha < 0.9$)*

Idea

The older the observation, the lower its weight in the average...
i.e. "*exponential forgetting*"

And, RTO = 2 x ERTT
(RTO = Retransmission TimeOut)

TCP: Adaptive Retransmission (contd.)



A problem of association!

– why SampleRTT computation is not trivial

TCP: Adaptive Retransmission (contd.)

Stated simply ...

- If ACK received for retransmitted segment, then 2 possibilities:
  - ACK is for first tx, *or*
  - ACK is for second tx

TCP source *cannot distinguish* betwn above 2 cases

To sidestep this problem, we use the **Karn-Partridge** algorithm *(see next slide)*

TCP: Adaptive Retransmission (contd.)
(Karn-Partridge Algorithm)

- On segment re-tx, stop measuring SampleRTT
- Increase RTO on each segment retransmission
  (*i.e.* **backoff** process)

  $\mathrm{RTO}_{i+1} = q \times \mathrm{RTO}_i$, *where* q = *constant*

  if $q = 2$, it is called ***binary exponential backoff***

TCP: Adaptive Retransmission (contd.)
(Jacobson/Karels Algorithm)

$$\text{Difference}(K) = \text{SampleRTT}(K) - \text{ERTT}(K)$$

$$\text{ERTT}(K+1) = \text{ERTT}(K) + [\,\delta \times \text{Difference}(K)]$$

$$\text{Deviation}(K+1) = \text{Deviation}(K) + \delta \times [\,|\text{Difference}(K)| - \text{Deviation}(K)\,]$$

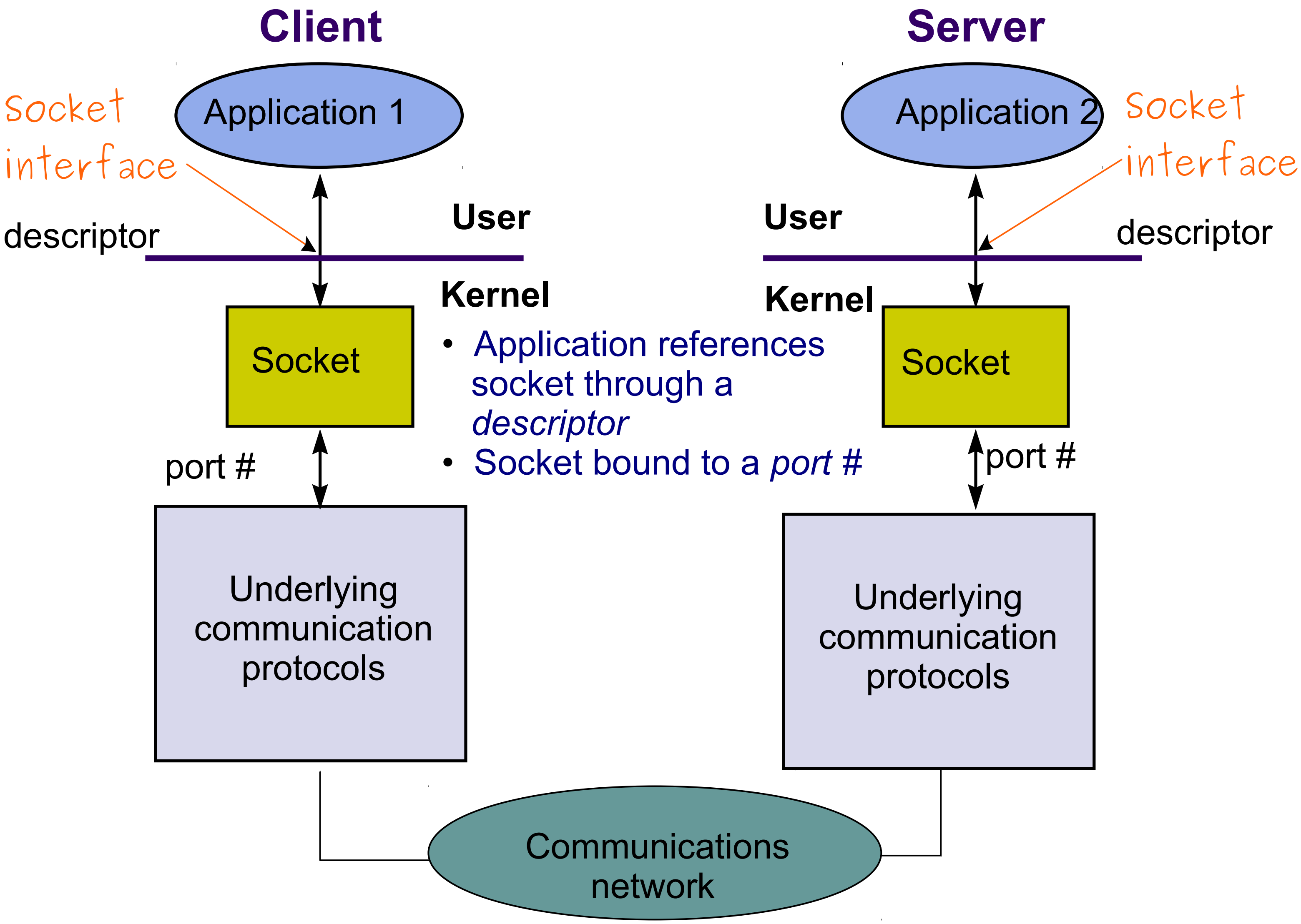
$$\text{RTO}(K+1) = \mu \times \text{ERTT}(K+1) + \Phi \times \text{Deviation}(K+1)$$

where: $0 < \delta < 1$, $\mu = 1$ *(usually)*, *and* $\Phi = 2$ *or* 4
(*most current implementations use* $\Phi = 4$)

Berkeley Sockets: The Socket API

- API (Application Programming Interface)
  - Provides std functions that can be called by applications
- Berkeley UNIX Sockets API
  - Abstraction for applications to send or receive data; hides details of underlying protocols & mechanisms
  - Applications create sockets that “plug into” network
  - Applications write/read to/from sockets
  - Implemented in OS kernel
- Also in Windows (WinSock) , Linux, and other OSes

Communications through Socket Interface



- Application references socket through a *descriptor*
- Socket bound to a *port #*

Stream Mode of Service

Connection-oriented (w/ **TCP**)

- First, set up connection between two peer application processes
- Then, reliable bidirectional in-sequence transfer of *byte stream* (boundaries not preserved in transfer)
- Multiple write/read between peer processes
- Finally, connection release

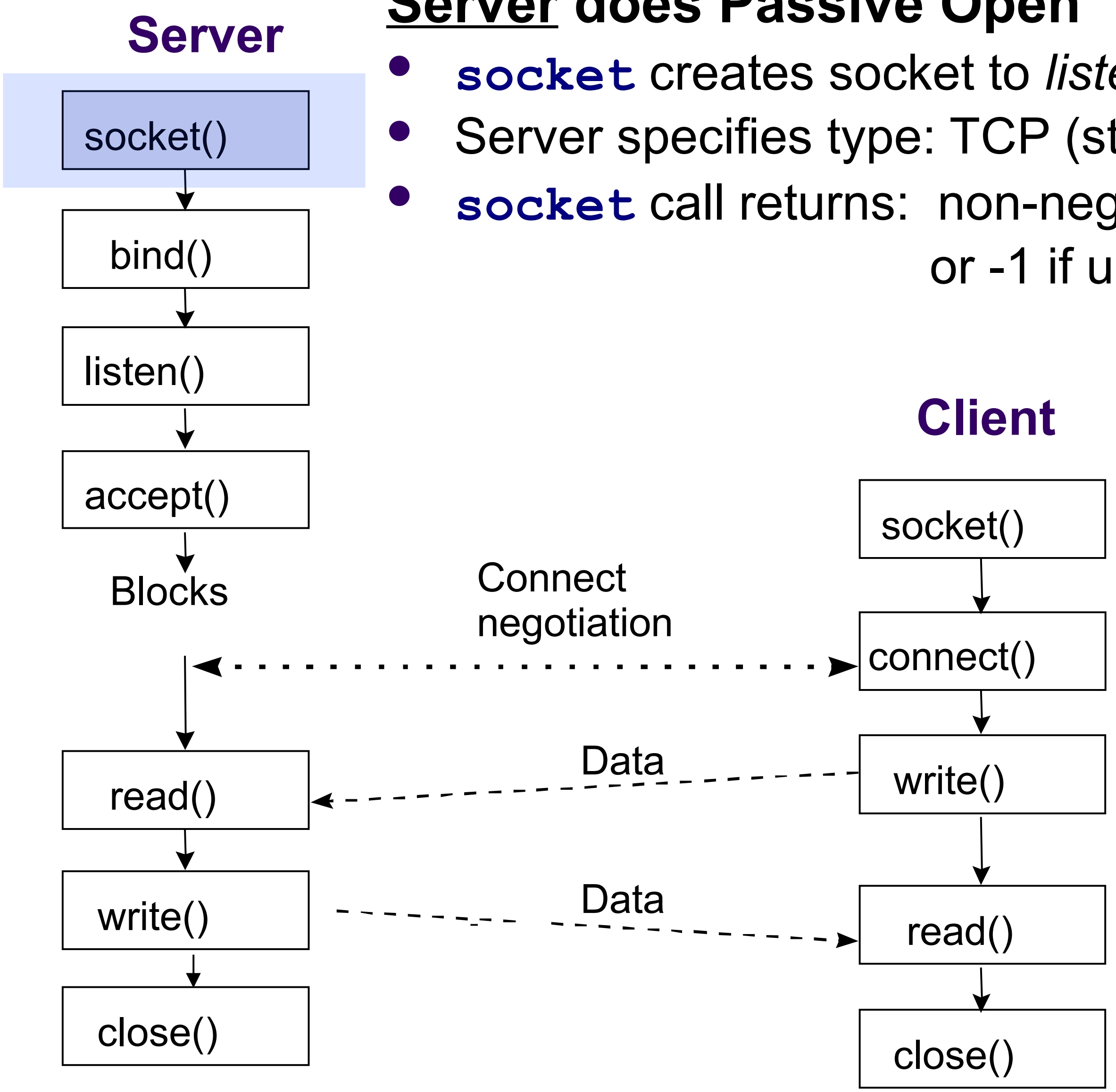
Connectionless (w/ **UDP**)

- Immediate transfer of one block of information (boundaries preserved)
- No setup overhead & delay
- Destination address with each block
- Send/receive to/from multiple peer processes
- Possibly out-of-order, possible pkt loss

Client & Server Differences

- **Server**
  - Specifies well-known port # when creating socket
  - May have multiple IP addresses (net interfaces)
  - Waits passively for client requests
- **Client**
  - Assigned ephemeral port #
  - Initiates communications with server
  - Needs to know server’s IP address & port #
    - → or, (logical name + DNS support) and (protocol)
  - Server learns client’s address & port # from request pkt

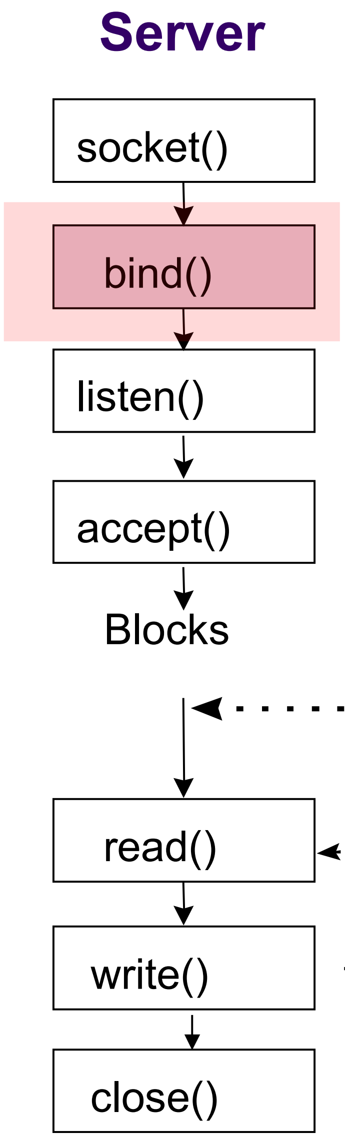
Socket Calls for Connection-Oriented Mode: socket() *page 1*



<u>Server</u> does Passive Open

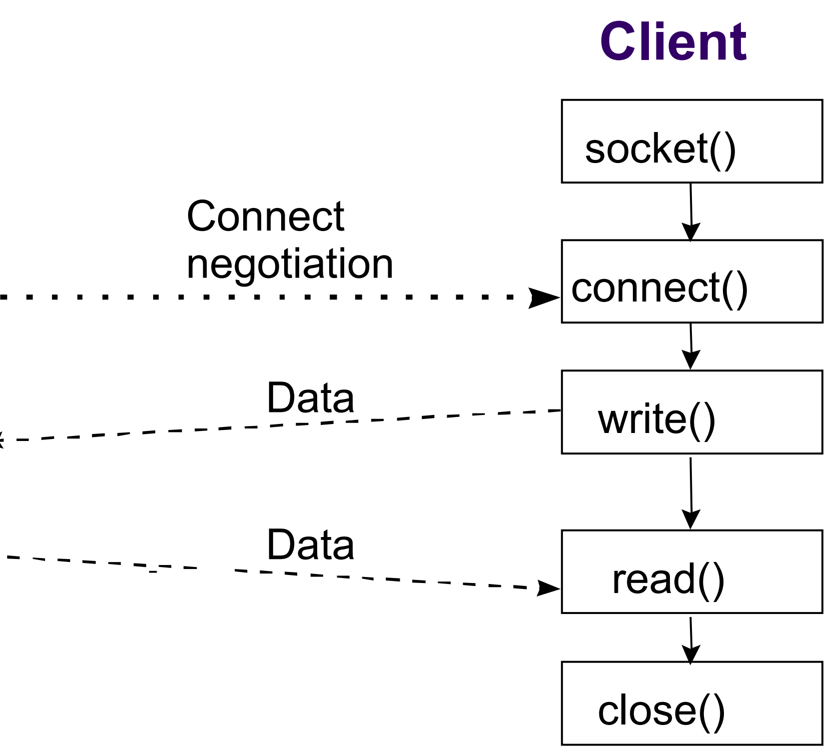
- `socket` creates socket to *listen* for connection requests
- Server specifies type: TCP (stream)
- `socket` call returns: non-negative integer *descriptor*; or -1 if unsuccessful

Socket Call: bind() *page 2*

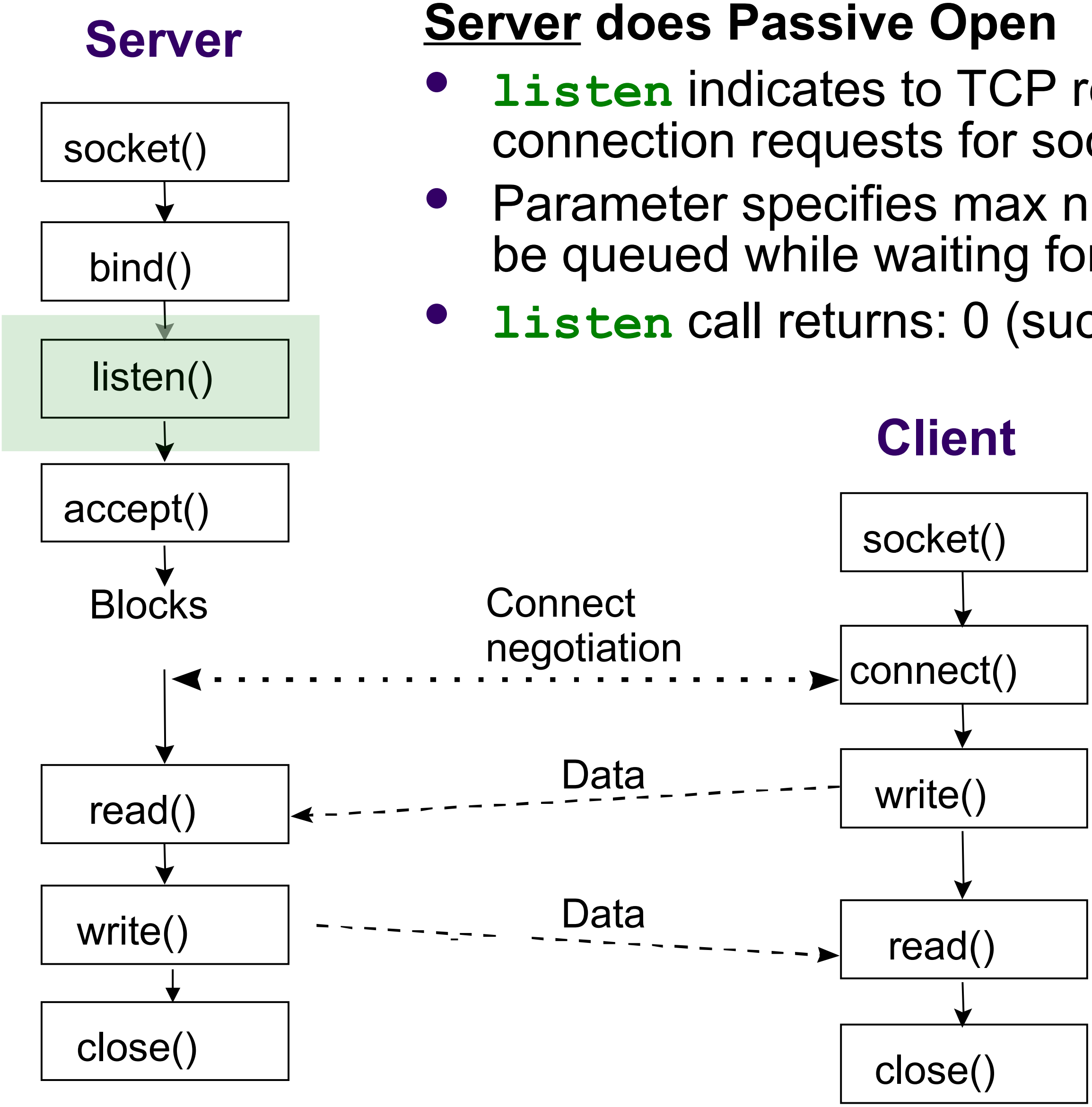


<u>Server</u> does Passive Open

- **`bind`** assigns local address & port # to socket with specified descriptor
- Wildcard IP address for multiple net interfaces
- **`bind`** call returns: 0 (success); or -1 (failure)
- Failure if port # already in use and <u>reuse</u> option not set



Socket Call: listen() *page 3*



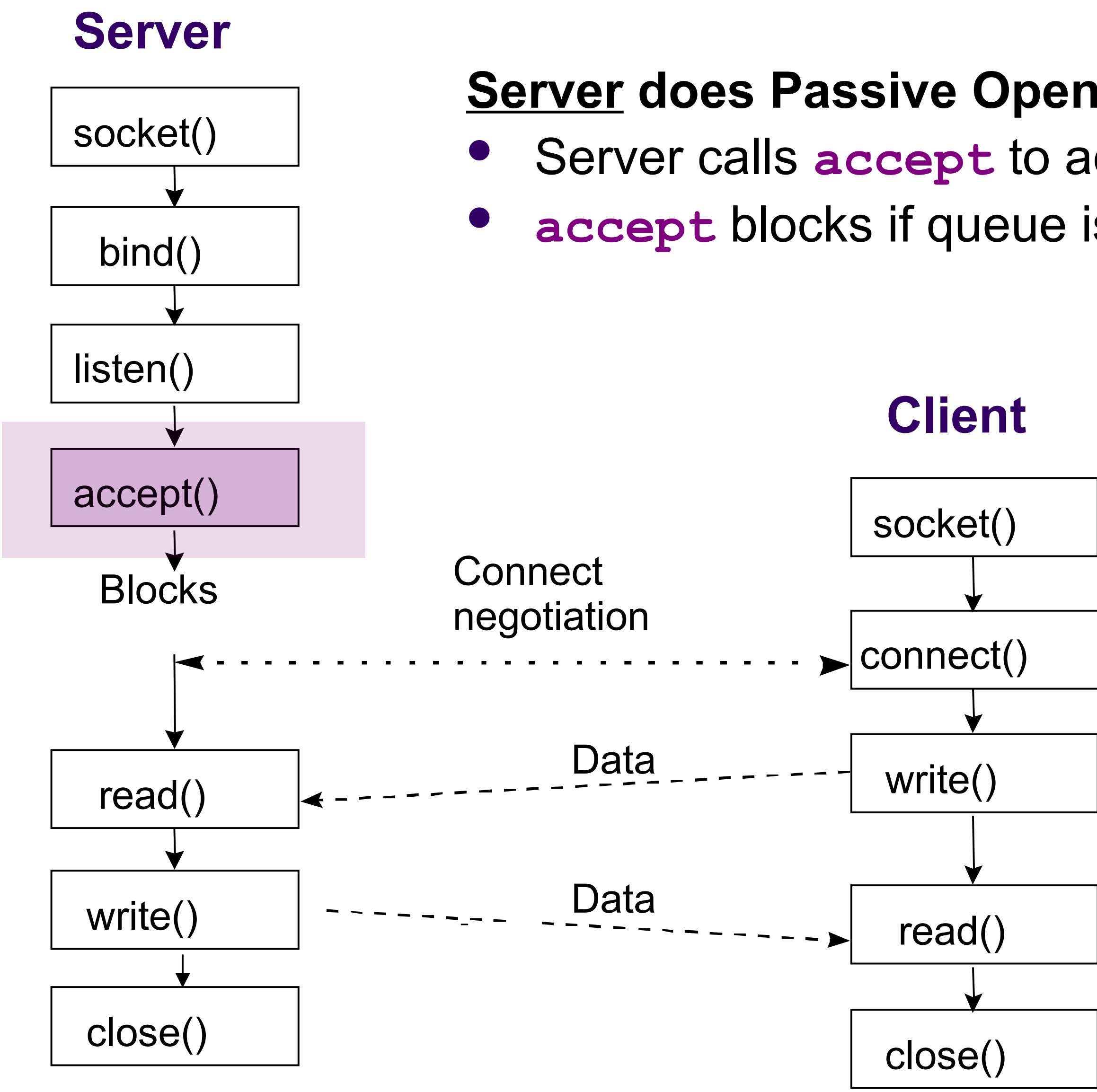
<u>Server</u> does Passive Open

- `listen` indicates to TCP readiness to receive connection requests for socket with given descriptor
- Parameter specifies max number of requests that may be queued while waiting for server to accept them
- `listen` call returns: 0 (success); or -1 (failure)

Socket Call: accept()*page 4*

<u>Server</u> does Passive Open

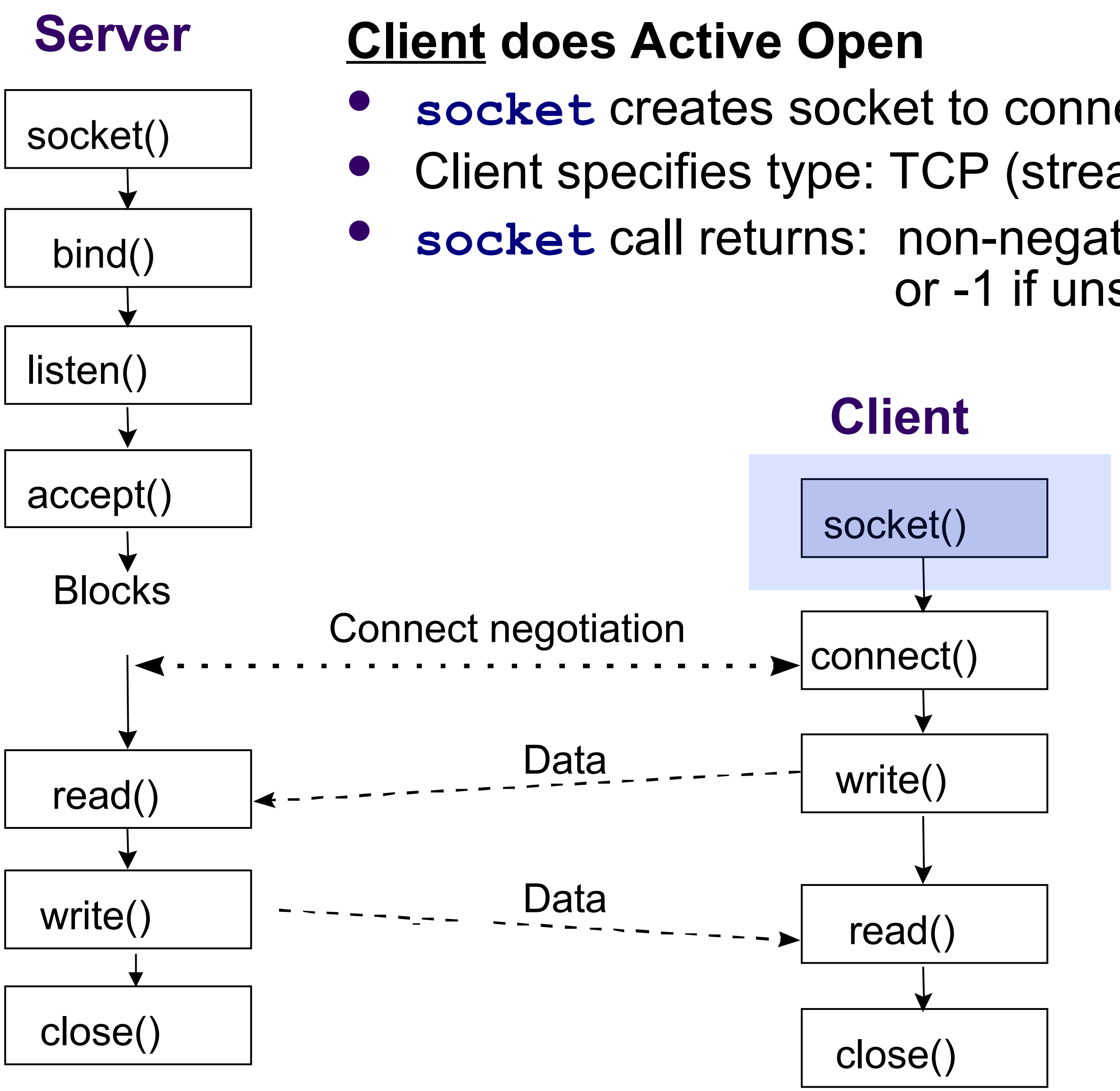
- Server calls `accept` to accept incoming requests
- `accept` blocks if queue is empty



Socket Call: Client's Side socket() *page 5*

<u>Client</u> does Active Open

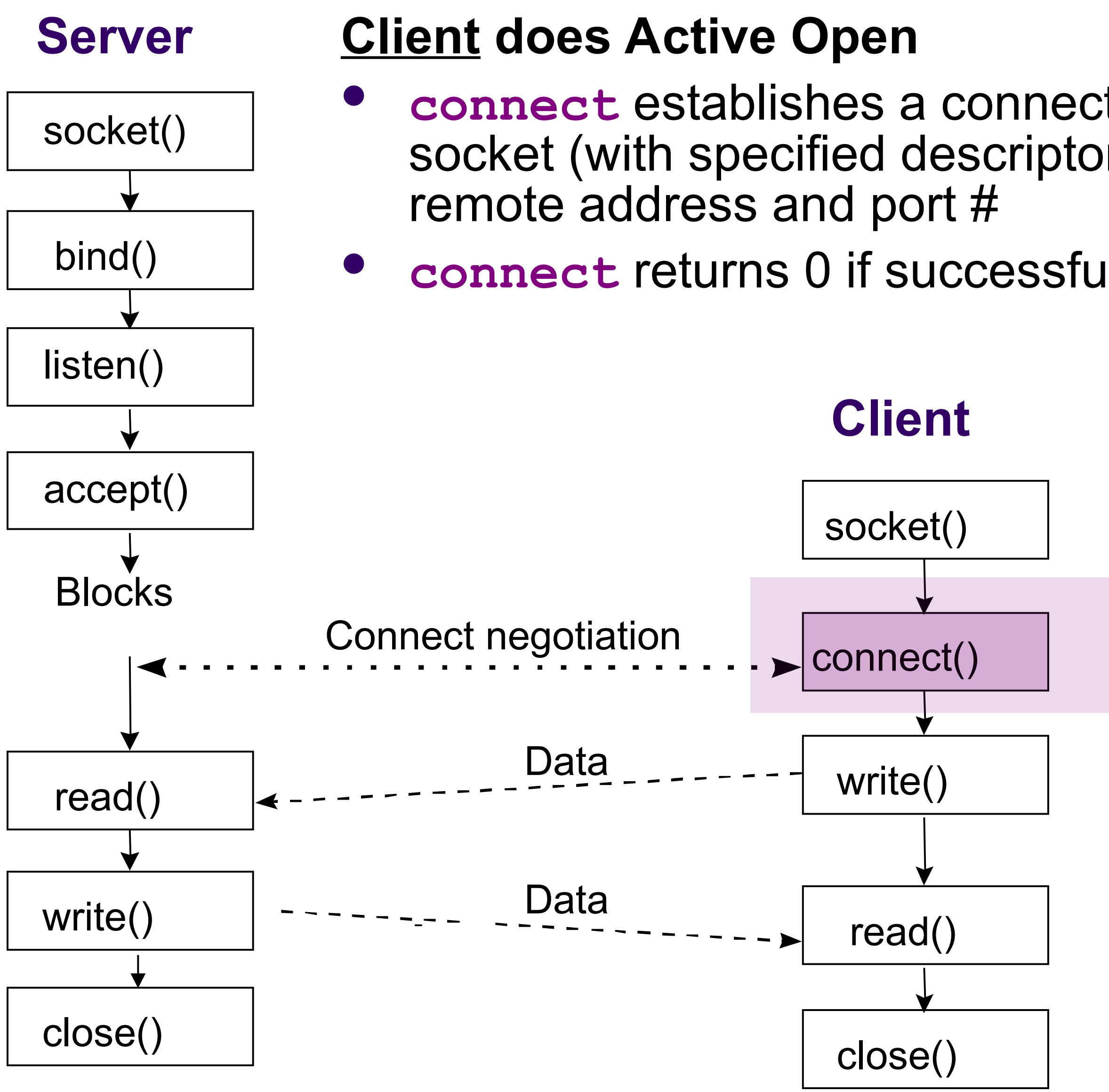
- `socket` creates socket to connect to server
- Client specifies type: TCP (stream)
- `socket` call returns: non-negative integer *descriptor*;
  or -1 if unsuccessful



Socket Call: connect() *page 6*

Client does Active Open

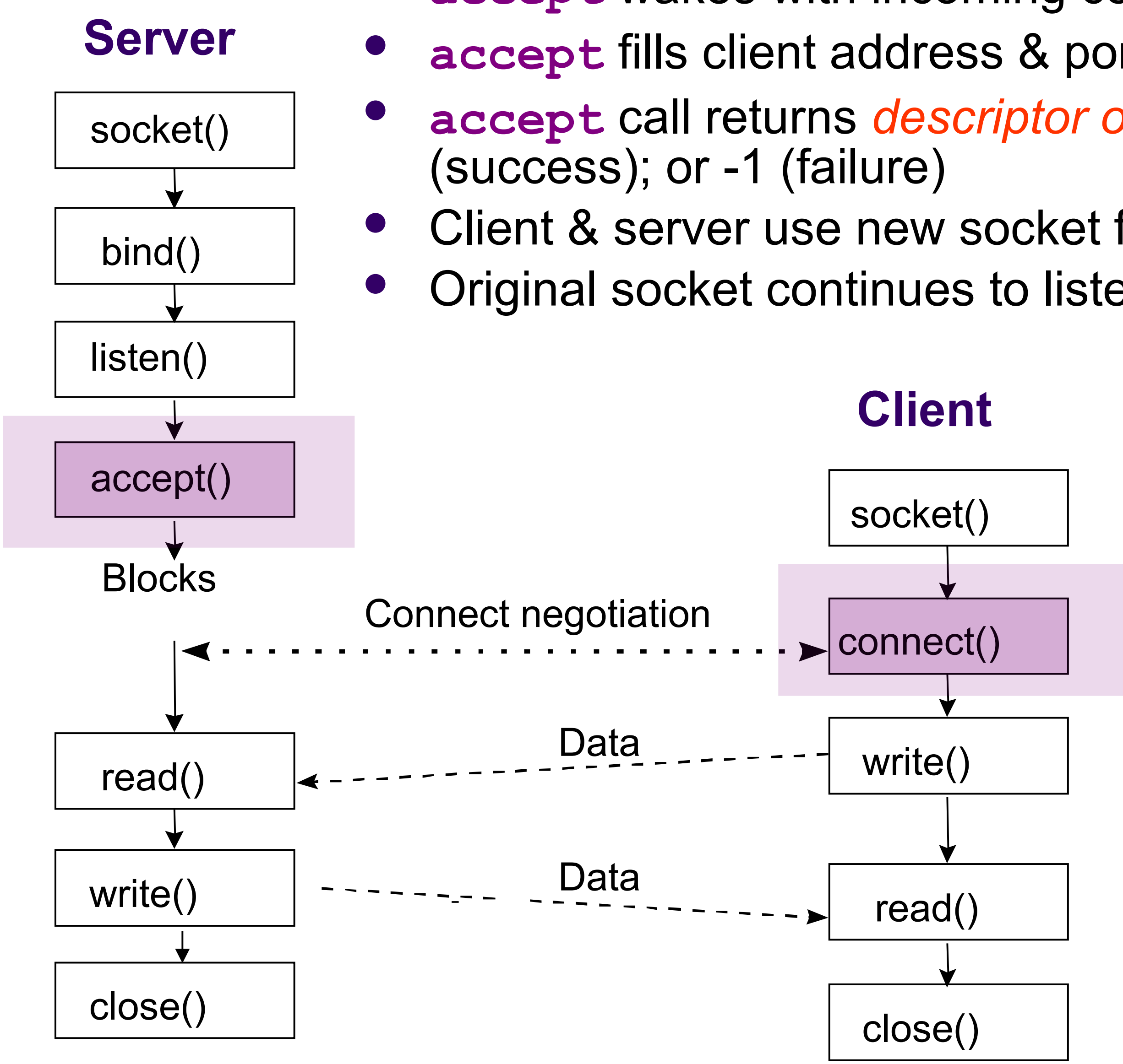
- **connect** establishes a connection on the local socket (with specified descriptor) to the specified remote address and port #
- **connect** returns 0 if successful; -1 if unsuccessful



Note:
connect initiates TCP's three-way handshake

Socket Call: accept() *page 7*

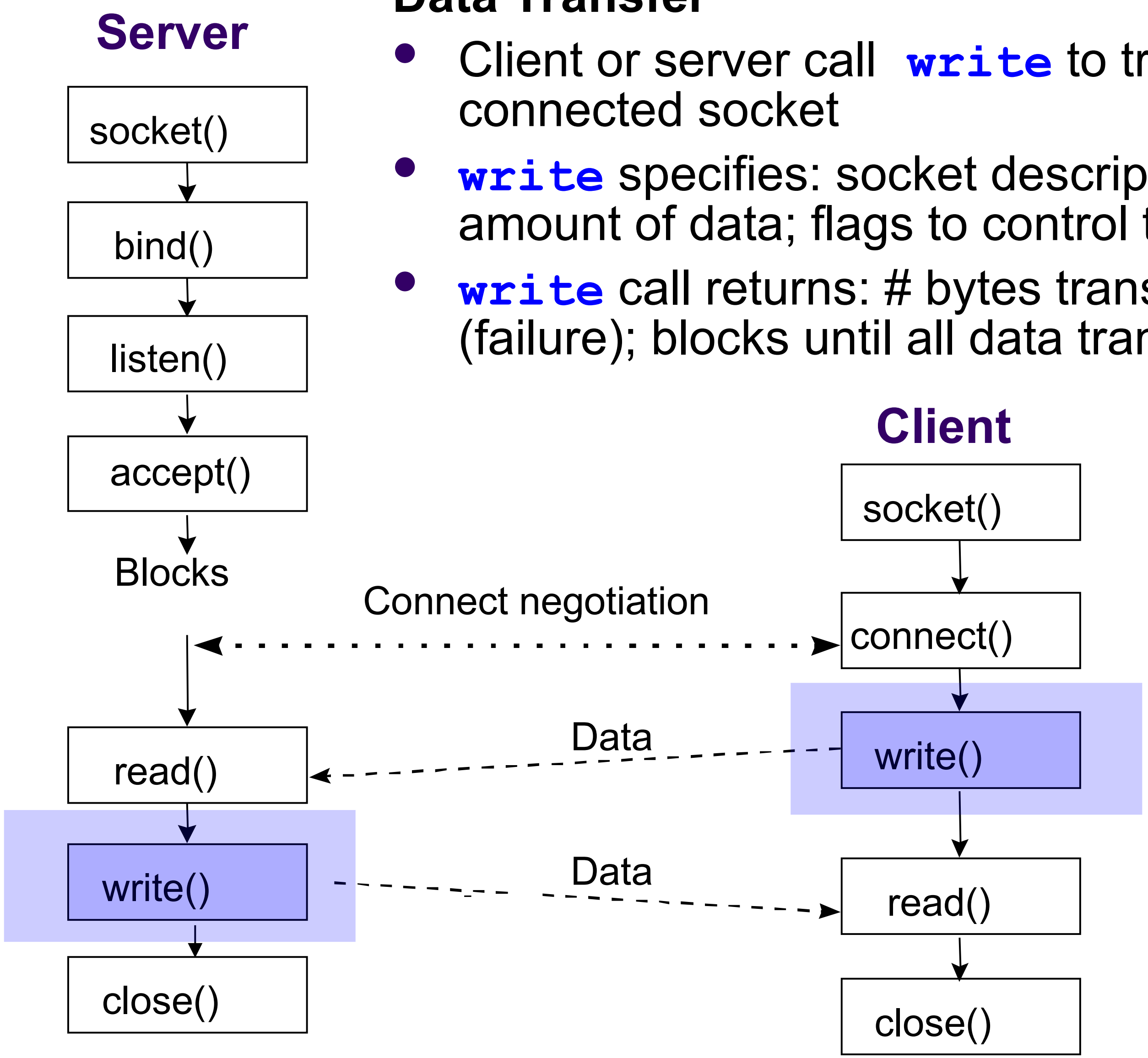
- `accept` wakes with incoming connection request
- `accept` fills client address & port # into address structure
- `accept` call returns *descriptor of **new** connection socket* (success); or -1 (failure)
- Client & server use new socket for data transfer
- Original socket continues to listen for new requests



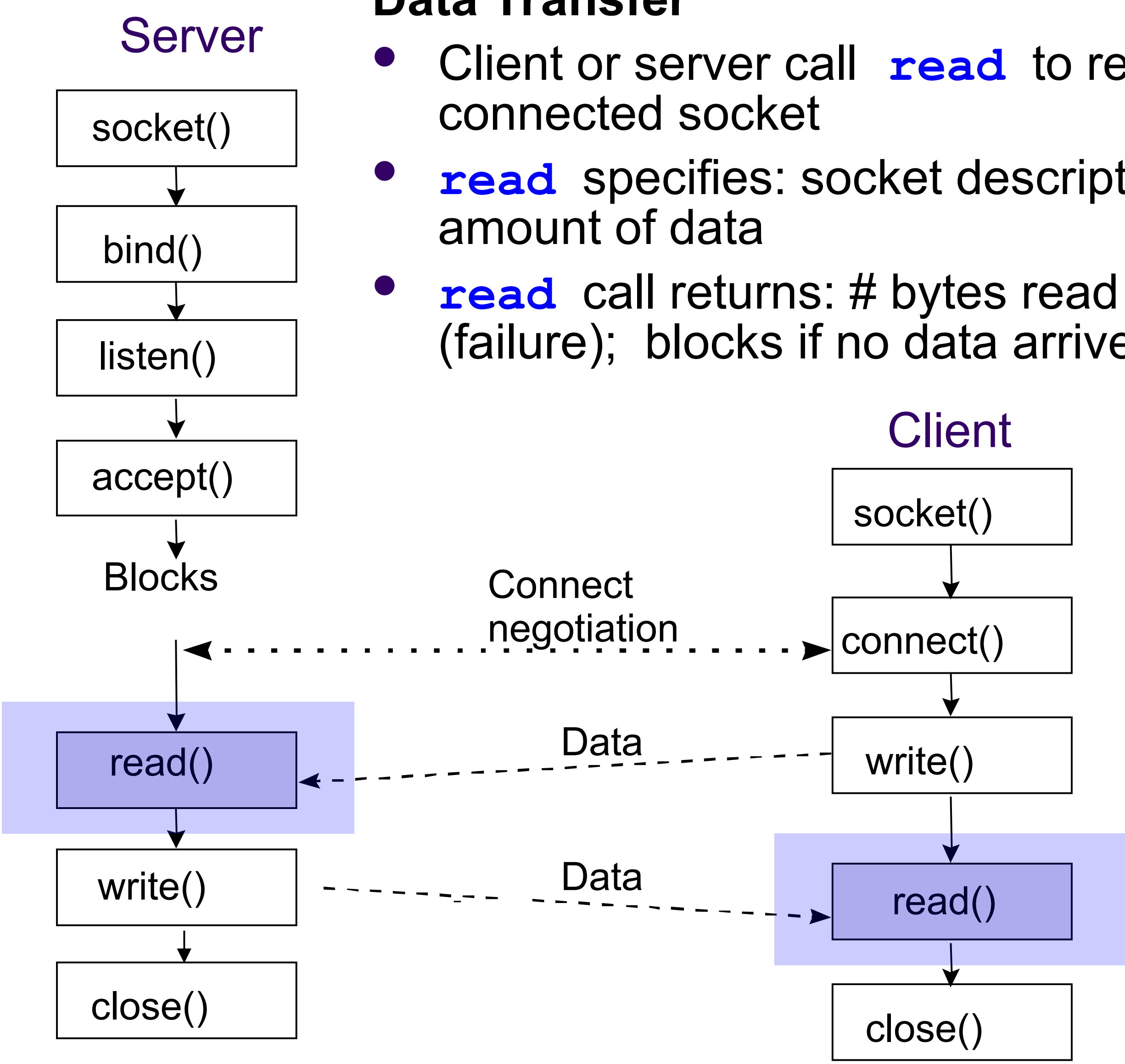
Socket Call: write() *page 8*

Data Transfer

- Client or server call **`write`** to transmit data into a connected socket
- **`write`** specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior
- **`write`** call returns: # bytes transferred (success); or -1 (failure); blocks until all data transferred



Socket Call: read() *page 9*



Data Transfer

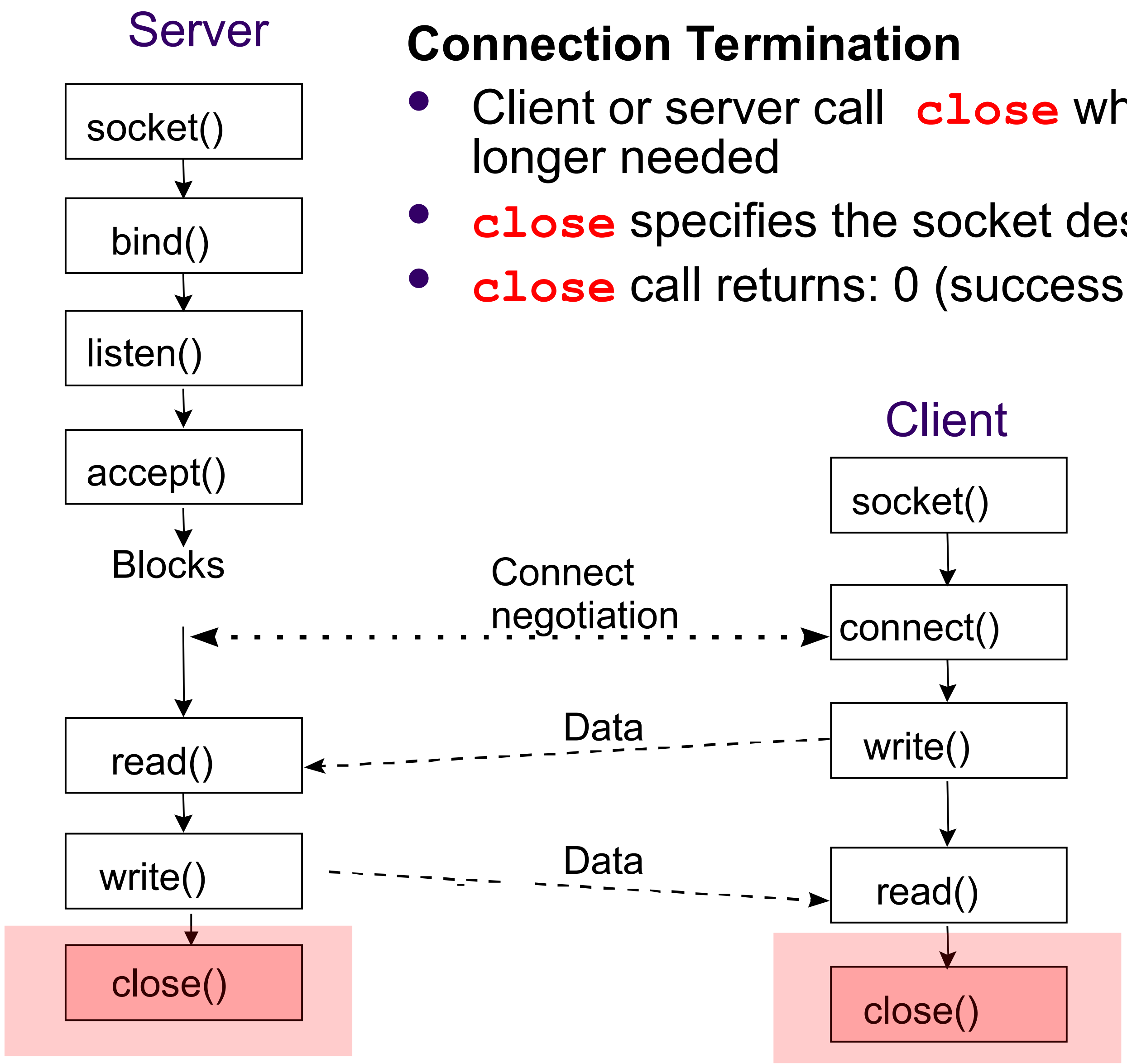
- Client or server call **`read`** to receive data from a connected socket
- **`read`** specifies: socket descriptor; pointer to a buffer; amount of data
- **`read`** call returns: # bytes read (success); or -1 (failure); blocks if no data arrives

Note:
`write` and **`read`** can be called multiple times to transfer byte streams in both directions

Socket Call: close() *page 9*

Connection Termination

- Client or server call **close** when socket is no longer needed
- **close** specifies the socket descriptor
- **close** call returns: 0 (success); or -1 (failure)



Note:
close initiates TCP's graceful close sequence

Example: TCP Echo Server *home reading from this point on*

```
/* A simple echo server using TCP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_TCP_PORT 3000
#define BUFLEN          256

int main(int argc, char **argv)
 {
  int n, bytes_to_read;
  int sd, new_sd, client_len, port;
  struct sockaddr_in   server, client;
  char *bp, buf[BUFLEN];

  switch(argc) {
    case 1:
          port = SERVER_TCP_PORT;
          break;
    case 2:
          port = atoi(argv[1]);
          break;
    default:
          fprintf(stderr, "Usage: %s [port]\n",
    argv[0]);
          exit(1);
   }

   /* Create a stream socket */
   if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket\n");
        exit(1);
      }

   /* Bind an address to the socket */
   bzero((char *)&server, sizeof(struct sockaddr_in));
   server.sin_family = AF_INET;
   server.sin_port = htons(port);
   server.sin_addr.s_addr = htonl(INADDR_ANY);
   if (bind(sd, (struct sockaddr *)&server,
      sizeof(server)) == -1) {
         fprintf(stderr, "Can't bind name to socket\n");
         exit(1);
   }

   /* queue up to 5 connect requests */
   listen(sd, 5);

   while(1) {
       client_len = sizeof(client);
       if ((new_sd = accept(sd, (struct sockaddr *)&client,
            &client_len)) == -1) {
               fprintf(stderr, "Can't accept client\n");
               exit(1);
       }

       bp = buf;
       bytes_to_read = BUFLEN;
       while ((n = read(new_sd, bp, bytes_to_read)) > 0) {
               bp += n;
               bytes_to_read -= n;
       }
       printf("Rec'd: %s\n", buf);

       write(new_sd, buf, BUFLEN);
       printf("Sent: %s\n", buf);
       close(new_sd);
   }
   close(sd);
   return(0);
} /*end*/
```

Example: TCP Echo Client

```
/* A simple TCP client */
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_TCP_PORT 3000
#define BUFLEN          256

int main(int argc, char **argv)
{
      int   n, bytes_to_read;
      int   sd, port;
      struct hostent    *hp;
      struct sockaddr_in server;
      char  *host, *bp, rbuf[BUFLEN], sbuf[BUFLEN];

      switch(argc) {
      case 2:
            host = argv[1];
            port = SERVER_TCP_PORT;
            break;
      case 3:
            host = argv[1];
            port = atoi(argv[2]);
            break;
      default:
            fprintf(stderr, "Usage: %s host [port]\n",
      argv[0]);
            exit(1);
      }

      /* Create a stream socket */
      if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -
      1) {
            fprintf(stderr, "Can't create socket\n");
            exit(1);
      }
```

```
      bzero((char *)&server, sizeof(struct sockaddr_in));
      server.sin_family = AF_INET;
      server.sin_port = htons(port);
      if ((hp = gethostbyname(host)) == NULL) {
         fprintf(stderr, "Can't get server's address\n");
         exit(1);
      }
      bcopy(hp->h_addr, (char *)&server.sin_addr, hp-
      >h_length);

      /* Connecting to the server */
      if (connect(sd, (struct sockaddr *)&server,
      sizeof(server)) == -1) {
            fprintf(stderr, "Can't connect\n");
            exit(1);
      }
      printf("Connected: server's address is %s\n", hp-
      >h_name);

      printf("Transmit:\n");
      gets(sbuf);
      write(sd, sbuf, BUFLEN);

      printf("Receive:\n");
      bp = rbuf;
      bytes_to_read = BUFLEN;
      while ((n = read(sd, bp, bytes_to_read)) > 0) {
            bp += n;
            bytes_to_read -= n;
      }
      printf("%s\n", rbuf);

      close(sd);
      return(0);
}
```

Socket Calls for Connection-Less Mode (UDP)

Read this on your own from Ch 2 of the textbook