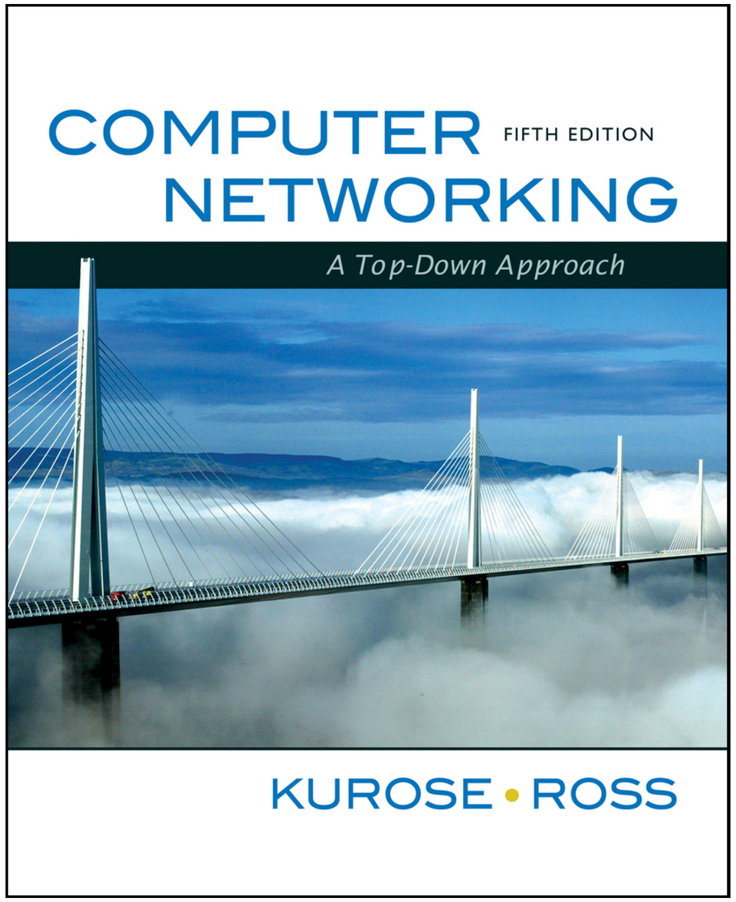
Module 2
Transport Layer Protocols

Please note: Most of these slides come from this book. Note their copyright notice below…



A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

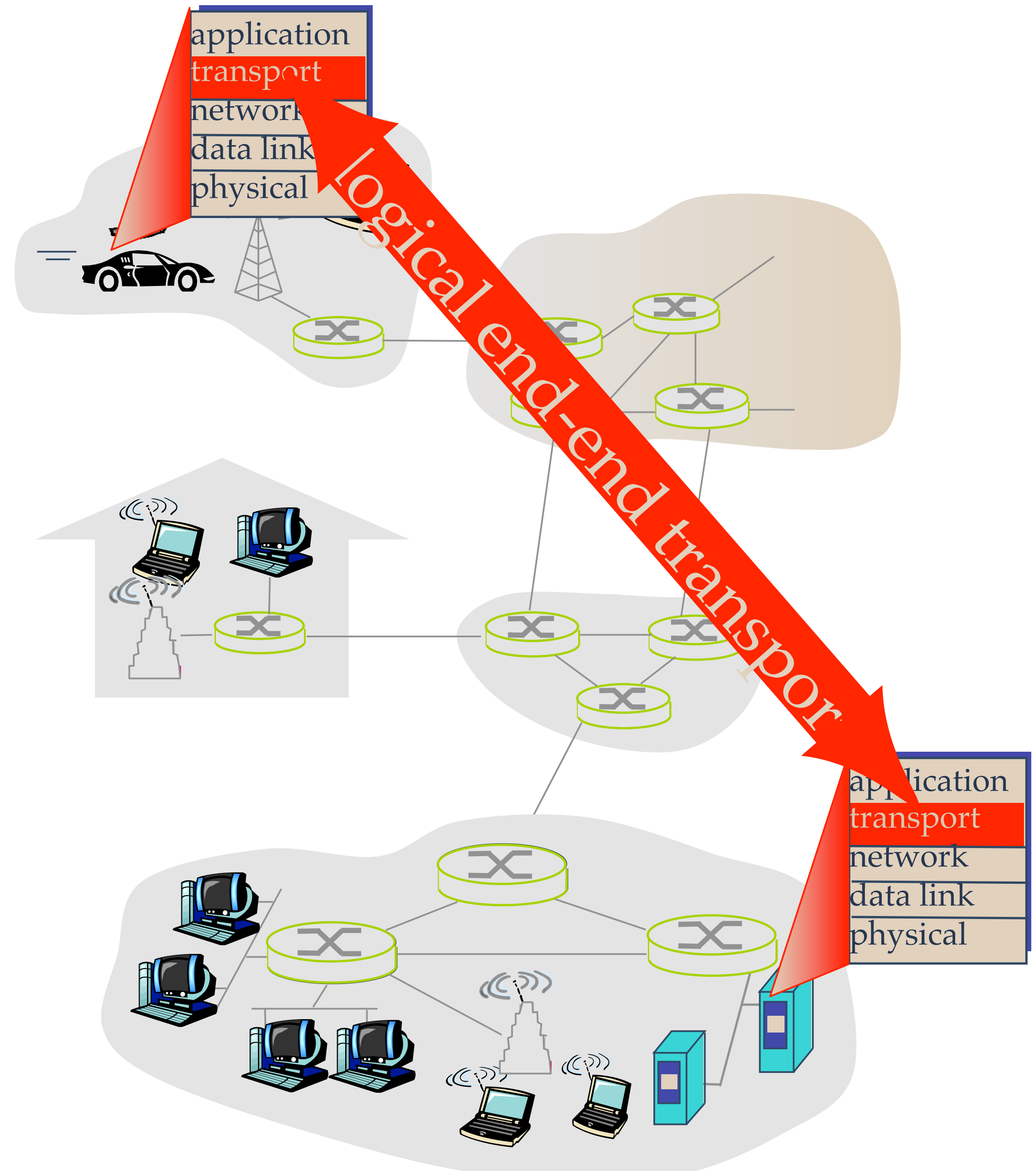
- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

Transport services & protocols

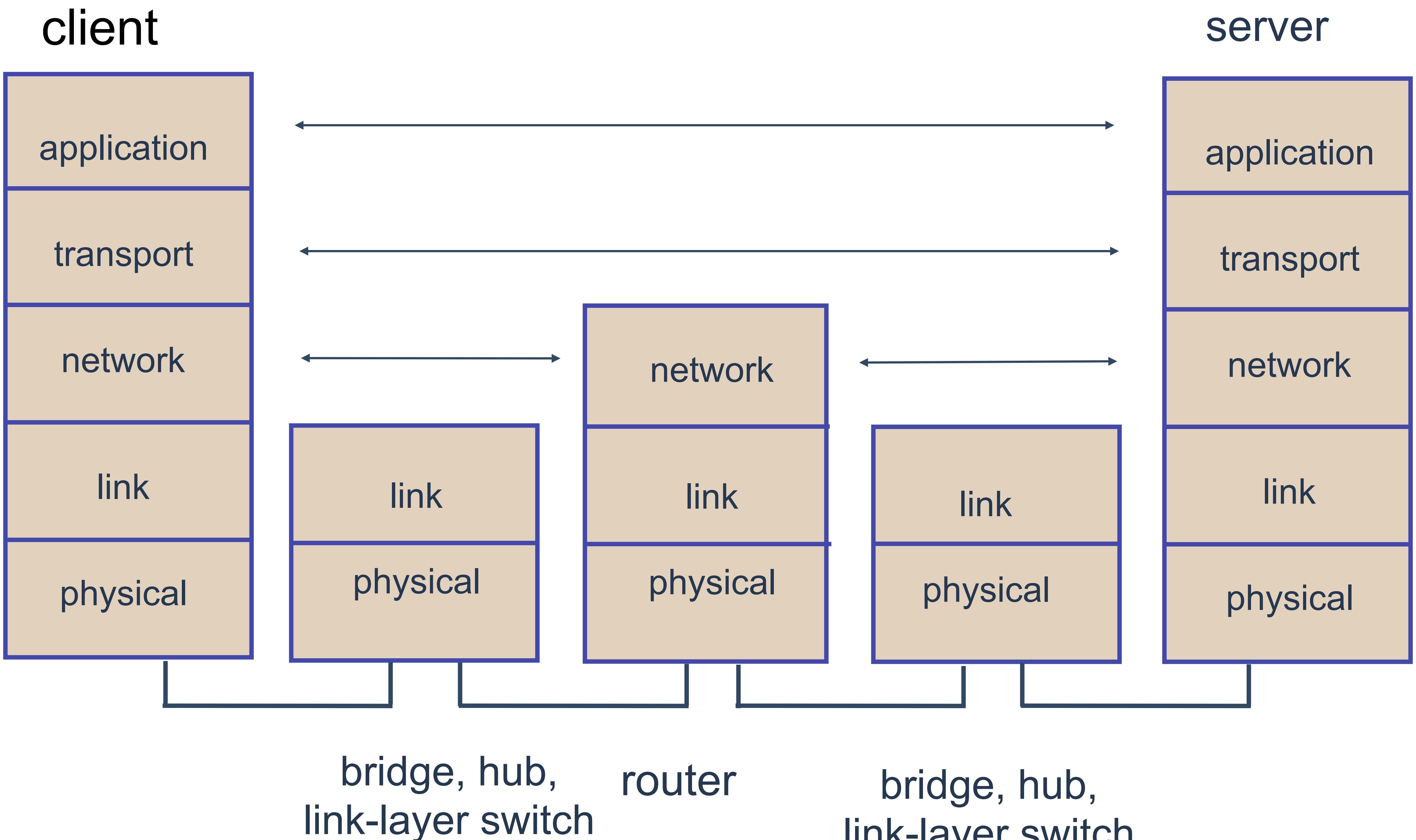
- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - ➡ send side: breaks app messages into segments, passes to network layer
  - ➡ rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol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



Transport vs. network layer

- *network layer:* enables logical communication between hosts
  - ➡ your laptop and UWaterloo's computer
- *transport layer:* enables logical communication between processes
  - ➡ your laptop's Web browser and UWaterloo's Web server
  - ➡ your laptop's video player and CBC's video server
  - ➡ relies on, enhances, network layer services
    - ✦ Enhancements depend on the particular protocol

Transport Layer is End-to-End



Internet Protocols

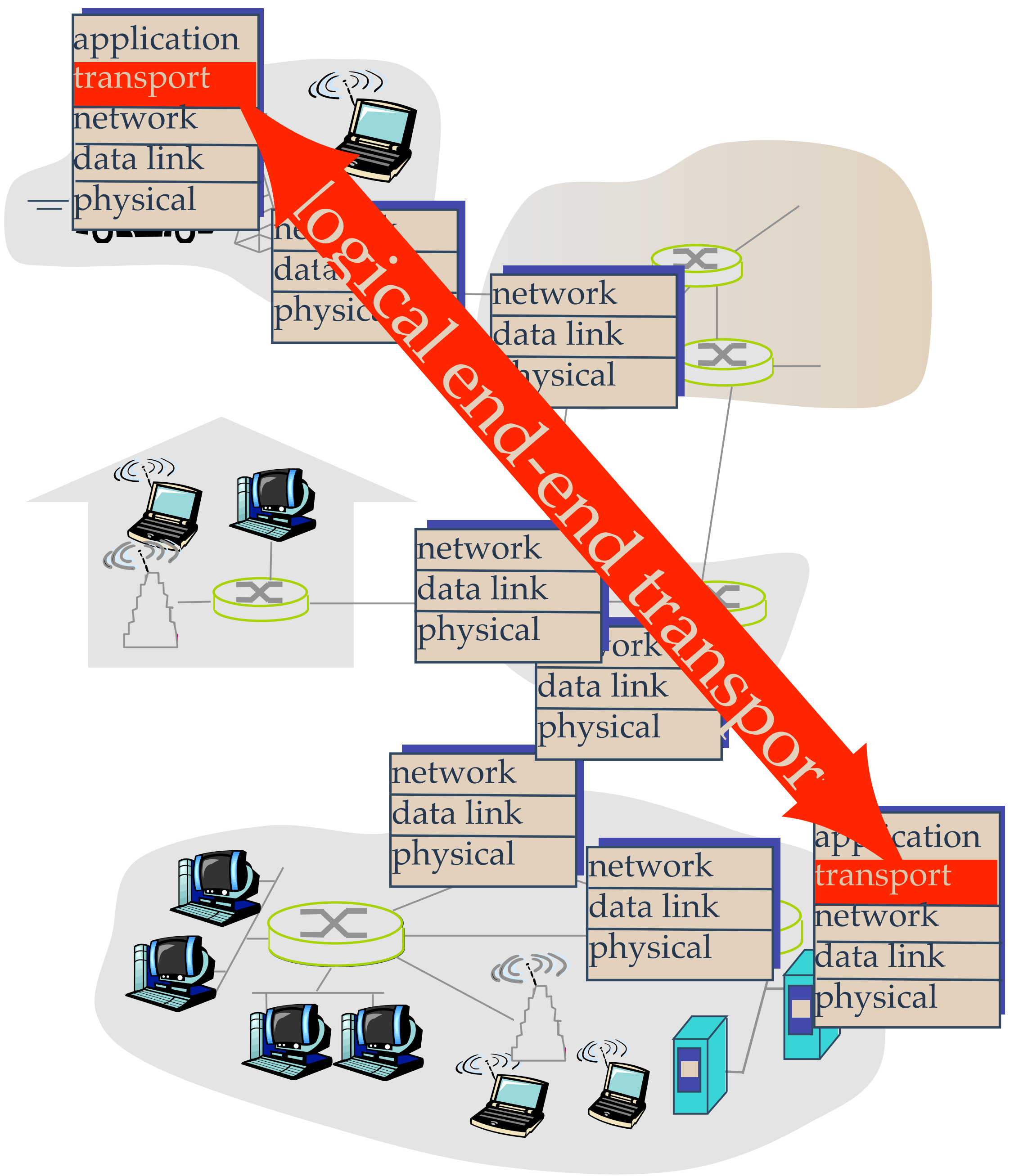
Application	FTP Telnet NFS SMTP HTTP …					
Transport	TCP			UDP		
Network	IP					
Data Link Physical	X.25	Ethernet	Packet Radio	ATM	FDDI	…

Internet Apps: Their Protocols & Transport Protocols

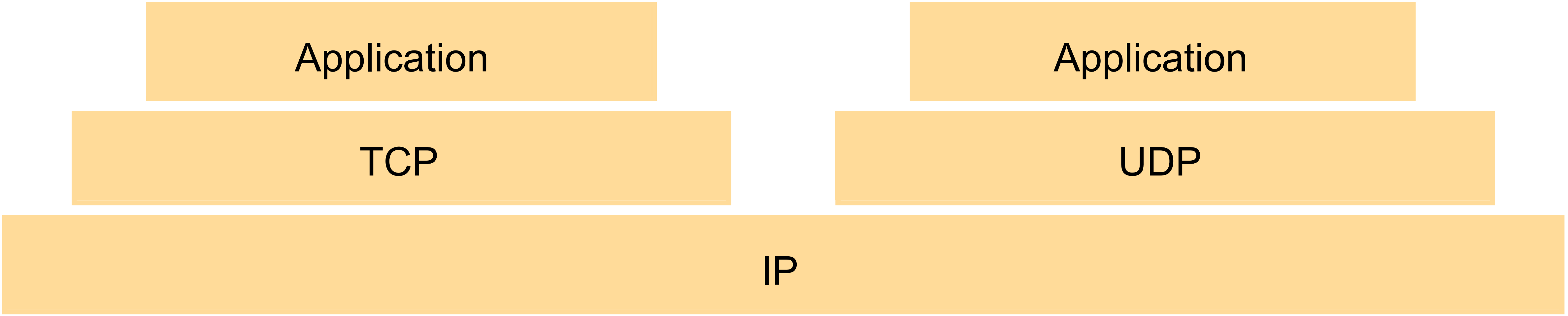
Applications	Data Loss	Throughput	Time Sensitive	Application Layer Protocol	Transport Protocol
Email	No loss	Elastic	No	smtp	TCP
remote terminal access	No loss	Elastic	Yes	telnet	TCP
Web	No loss	Elastic	No	http	TCP
File transfer	No loss	Elastic	No	ftp	TCP
streaming multimedia	Loss tolerant	audio: 5kbps-1Mbps video: 10kbps-5Mbps	Yes, 100's msec	Proprietary	TCP or UDP
Remote file server	No loss	Elastic	No	NFS	TCP or UDP (typically UDP)
Internet telephony	Loss tolerant	Depends on encoding, 32kbps typically	Yes, few secs	SIP, RIP, Prorietary	TCP or UDP (typically UDP)

Internet transport-layer protocols

- UDP: unreliable, unordered delivery
  - ➡ Process-to-process data delivery
    - ✦ Multiplexing/demultiplexing
  - ➡ End-to-end error checking
- TCP: reliable, in-order delivery
  - ➡ congestion control
  - ➡ flow control
  - ➡ connection setup
- services not available:
  - ➡ delay guarantees
  - ➡ bandwidth guarantees

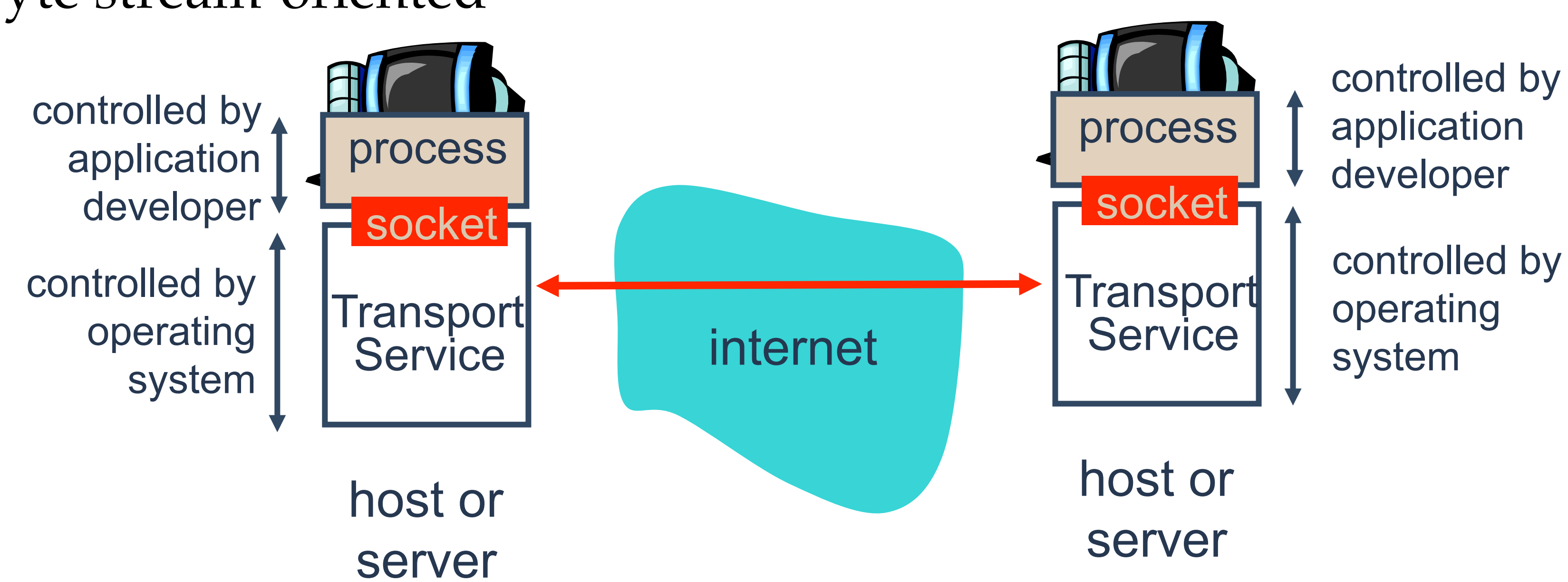


The programmer's conceptual view of a TCP/IP Internet



How Apps Access Transport Services

- Through sockets
- Socket: a *host-local*, *application-created*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another application process
- Socket API
  - introduced in BSD4.1 UNIX, 1981
  - explicitly created, used, released by apps
  - client/server paradigm
  - two types of transport service via socket API:
    - unreliable datagram
    - reliable, byte stream-oriented



UDP: User Datagram Protocol

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

Client/server socket interaction: UDP

Server (running on **hostid**)

create socket,
port= x.
serverSocket =
DatagramSocket()

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

Client

create socket,
clientSocket =
DatagramSocket()

Create datagram with server IP and
port=x; send datagram via
clientSocket

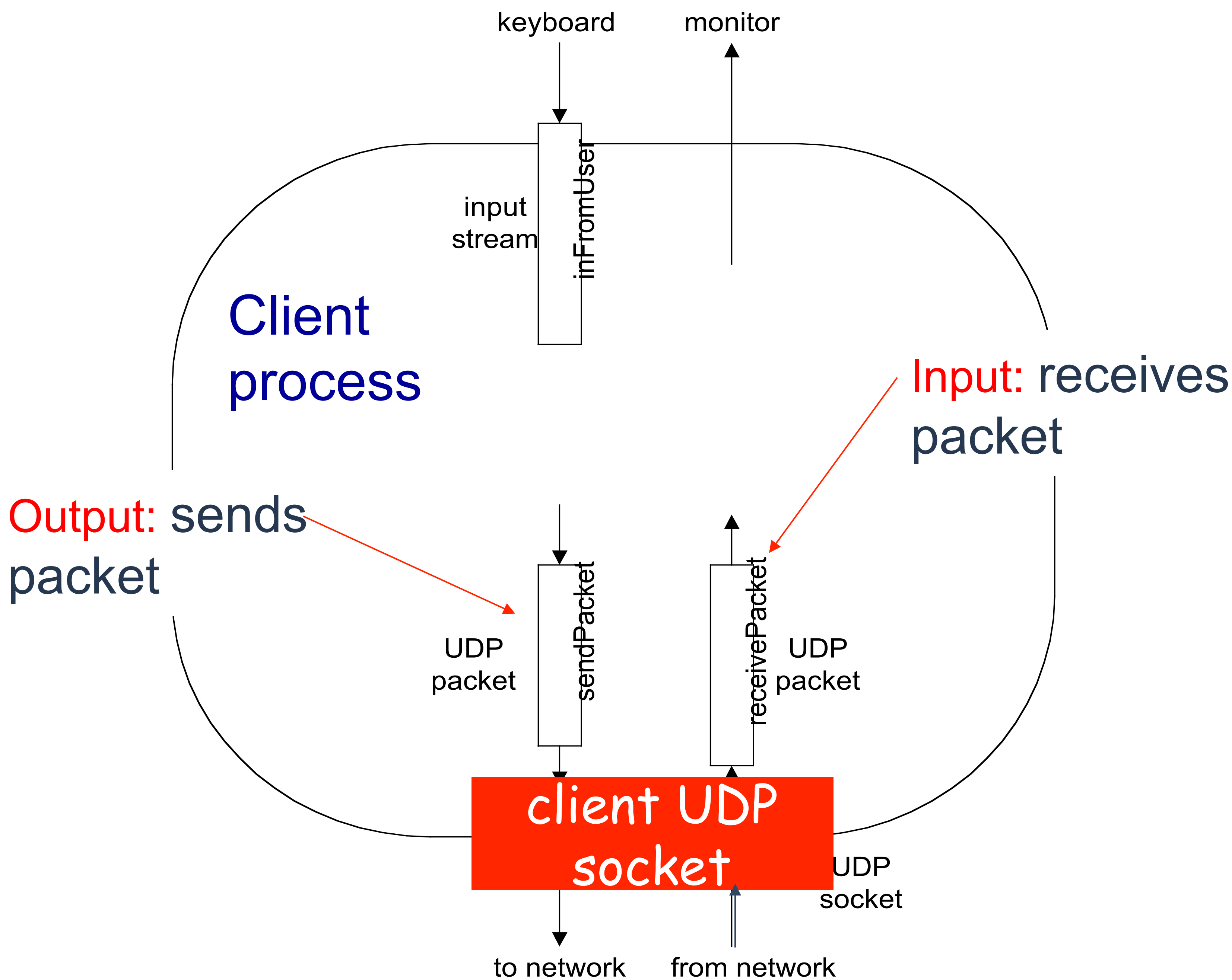
read datagram from
clientSocket

close
clientSocket

Example: Java client (UDP)

Example client-server app:

1) client reads line from standard input (**inFromUser** stream) , sends to server via socket (**outToServer** stream)
2) server reads line from socket
3) server converts line to uppercase, sends back to client
4) client reads, prints modified line from socket (**inFromServer** stream)



Example: UDP Java client

```
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception
    {

      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));

      DatagramSocket clientSocket = new DatagramSocket();

      InetAddress IPAddress = InetAddress.getByName("hostname");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();
      sendData = sentence.getBytes();
```

create input stream

create client socket

translate hostname to IP address using DNS

Example: UDP Java client (cont'd)

create datagram with data-to-send, length, IP addr, port

send datagram to server

read datagram from server

```
DatagramPacket sendPacket =
   new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

clientSocket.send(sendPacket);

DatagramPacket receivePacket =
   new DatagramPacket(receiveData, receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence =
    new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}
}
```

Example: UDP Java server

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
    {

      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
        {

          DatagramPacket receivePacket =
             new DatagramPacket(receiveData, receiveData.length);
          serverSocket.receive(receivePacket);
```

create datagram socket at port 9876

create space for received datagram

receive datagram

Example: UDP Java server (cont'd)

```
String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket =
   new DatagramPacket(sendData, sendData.length, IPAddress,
                  port);

serverSocket.send(sendPacket);
    }
  }
}
```

get IP addr port #, of sender

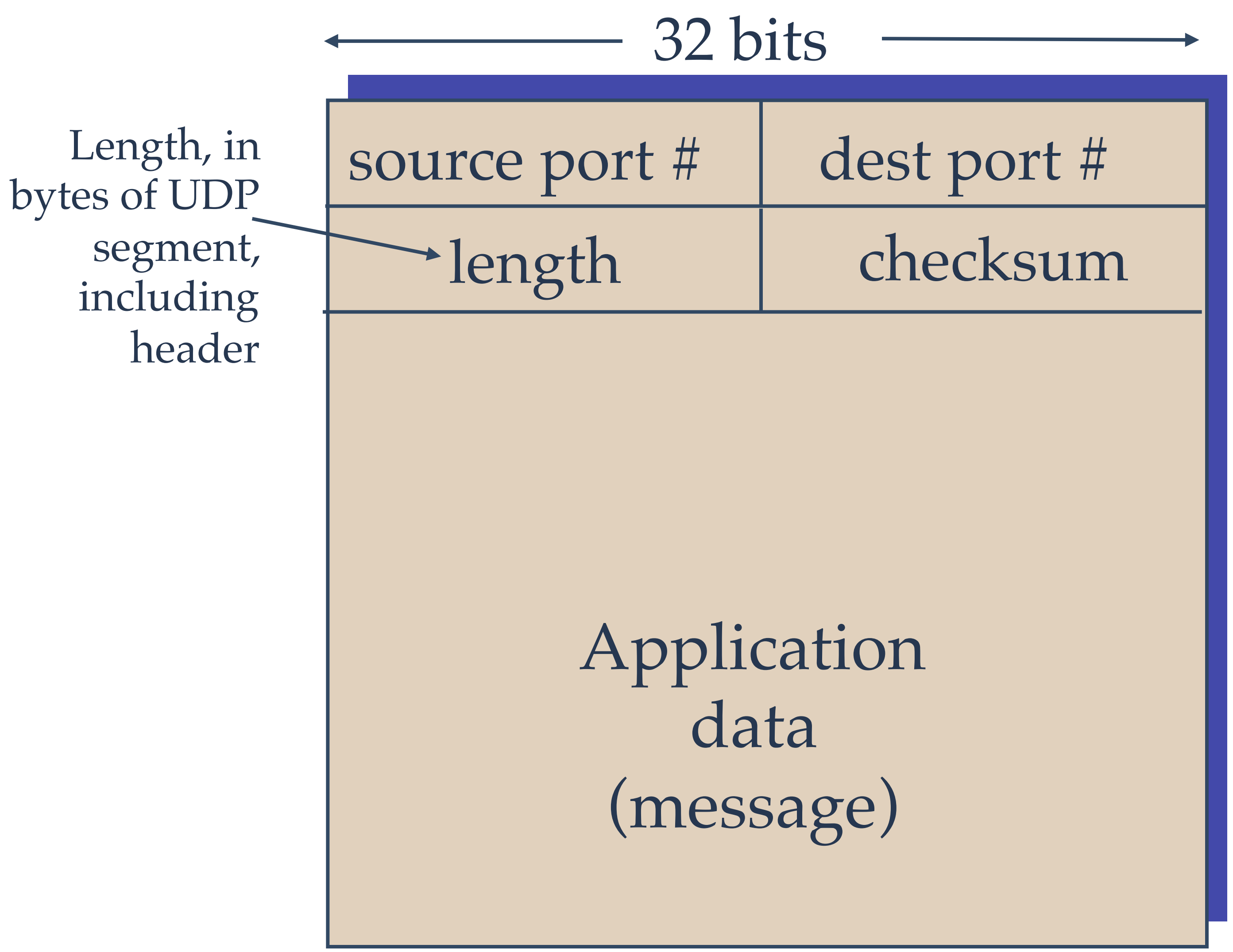
create datagram to send to client

write out datagram to socket

end of while loop, loop back and wait for another datagram

UDP Use and Format

- often used for streaming multimedia apps
  - ➡ loss tolerant
  - ➡ rate sensitive
- other UDP uses
  - ➡ DNS
  - ➡ SNMP
- reliable transfer over UDP: add reliability at application layer
  - ➡ application-specific error recovery!



UDP segment format

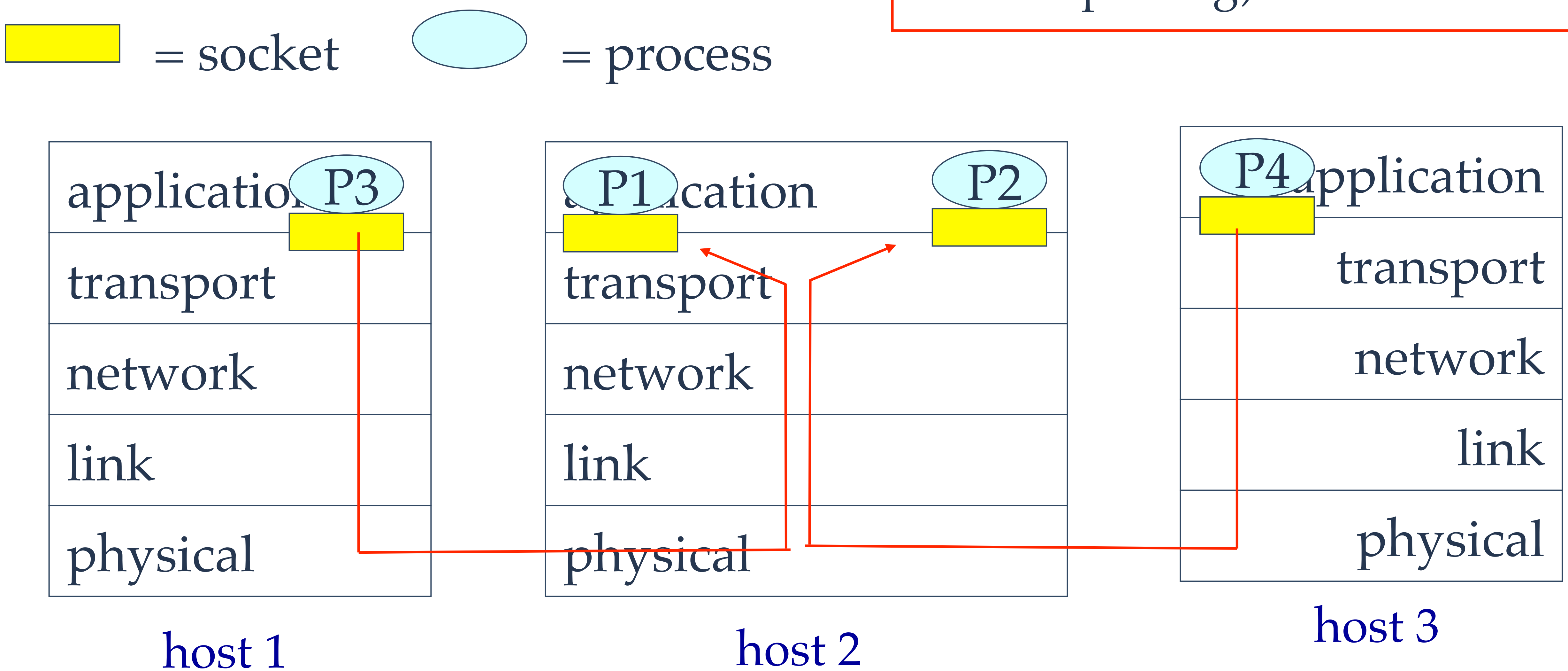
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

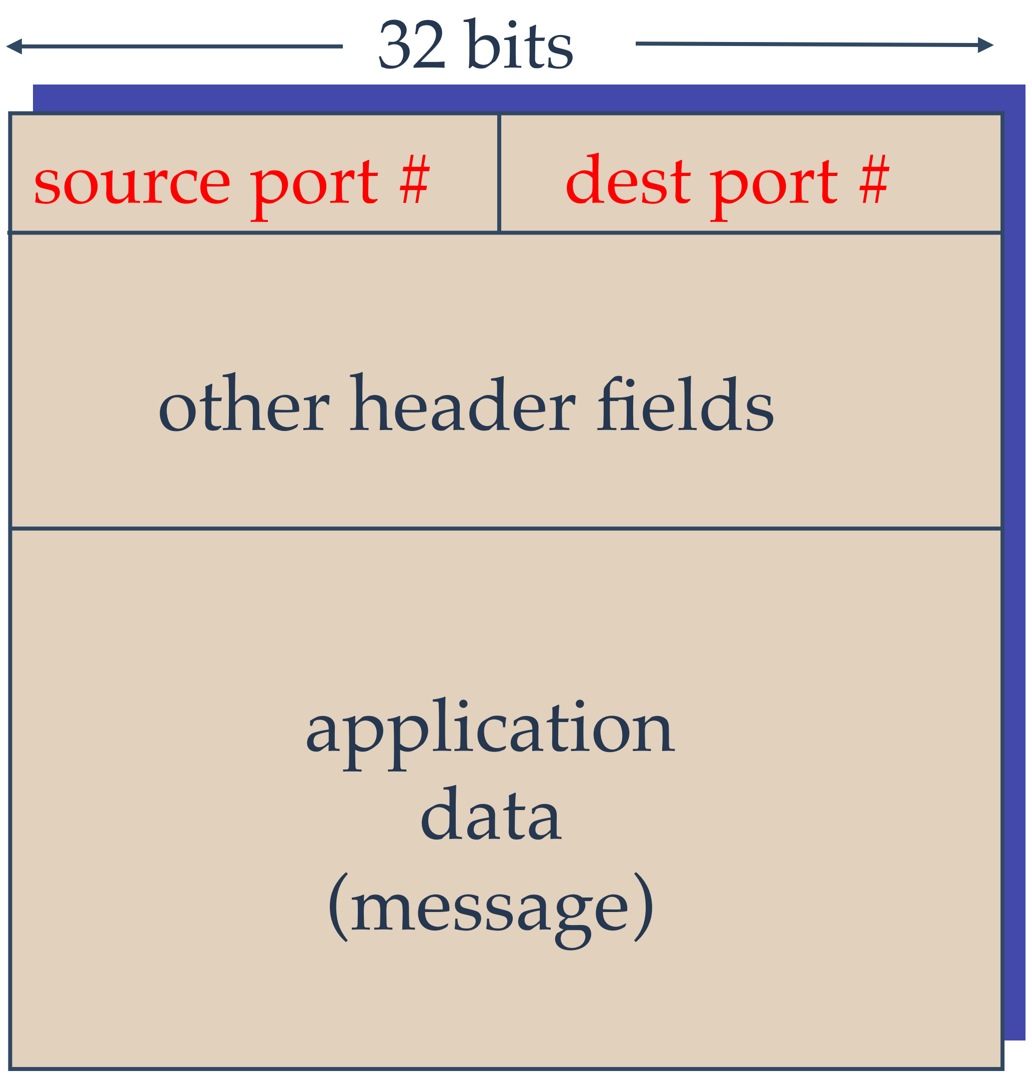
Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- *recall:* create sockets with host-local port numbers:

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
DatagramSocket mySocket2 = new DatagramSocket(12535);
```

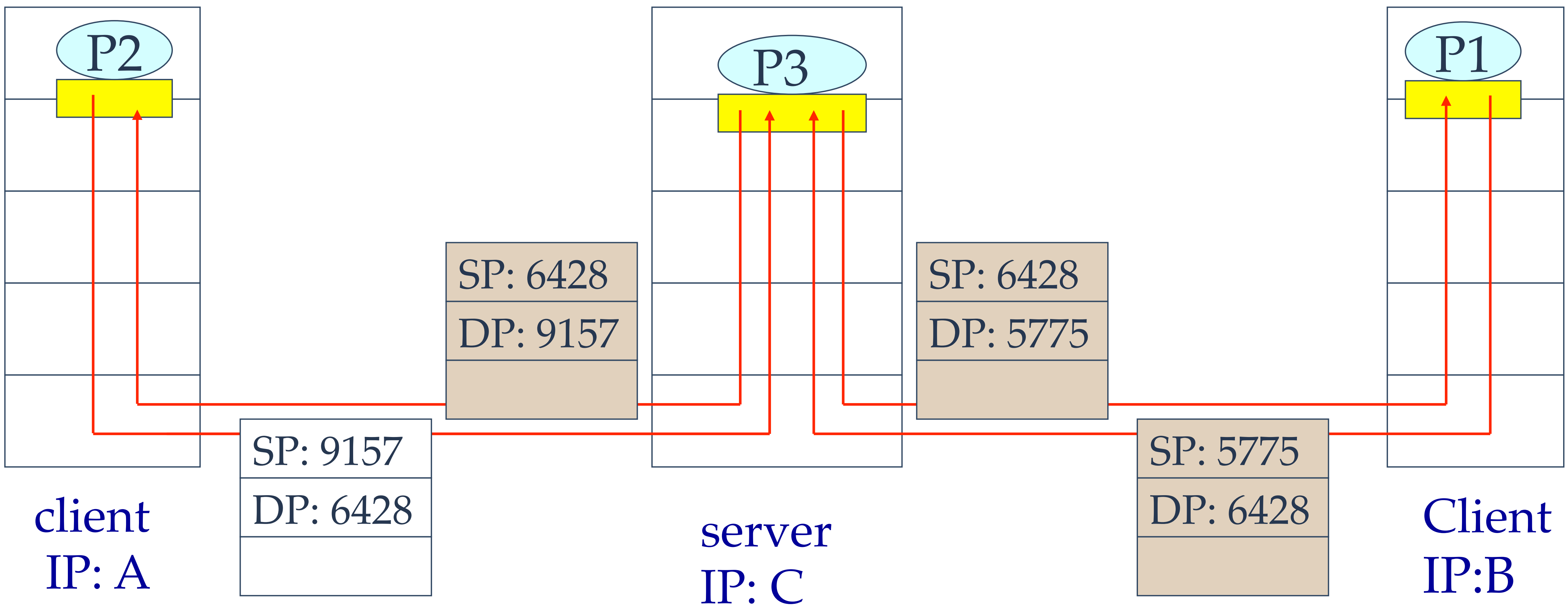
- *recall:* when creating datagram to send into UDP socket, must specify

  (dest IP address, dest port number)

- when host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



SP provides "return address"

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

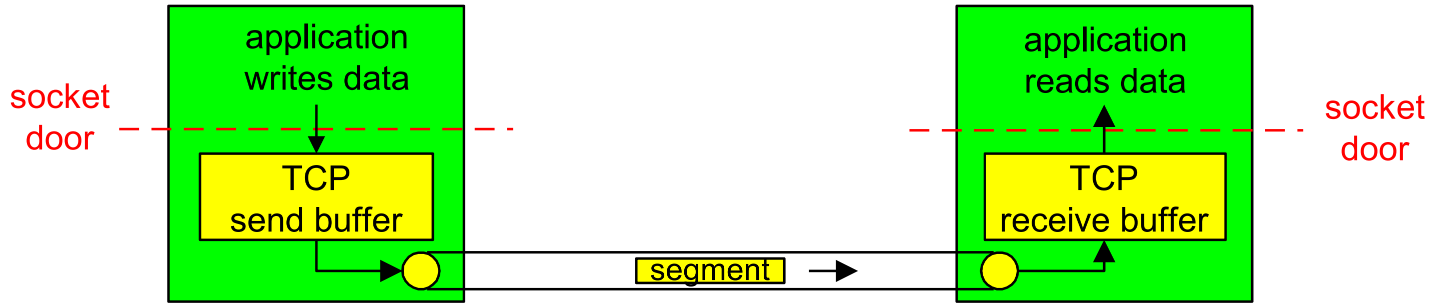
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.

TCP: Transport Control Protocol

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*



- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

Socket programming with TCP

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

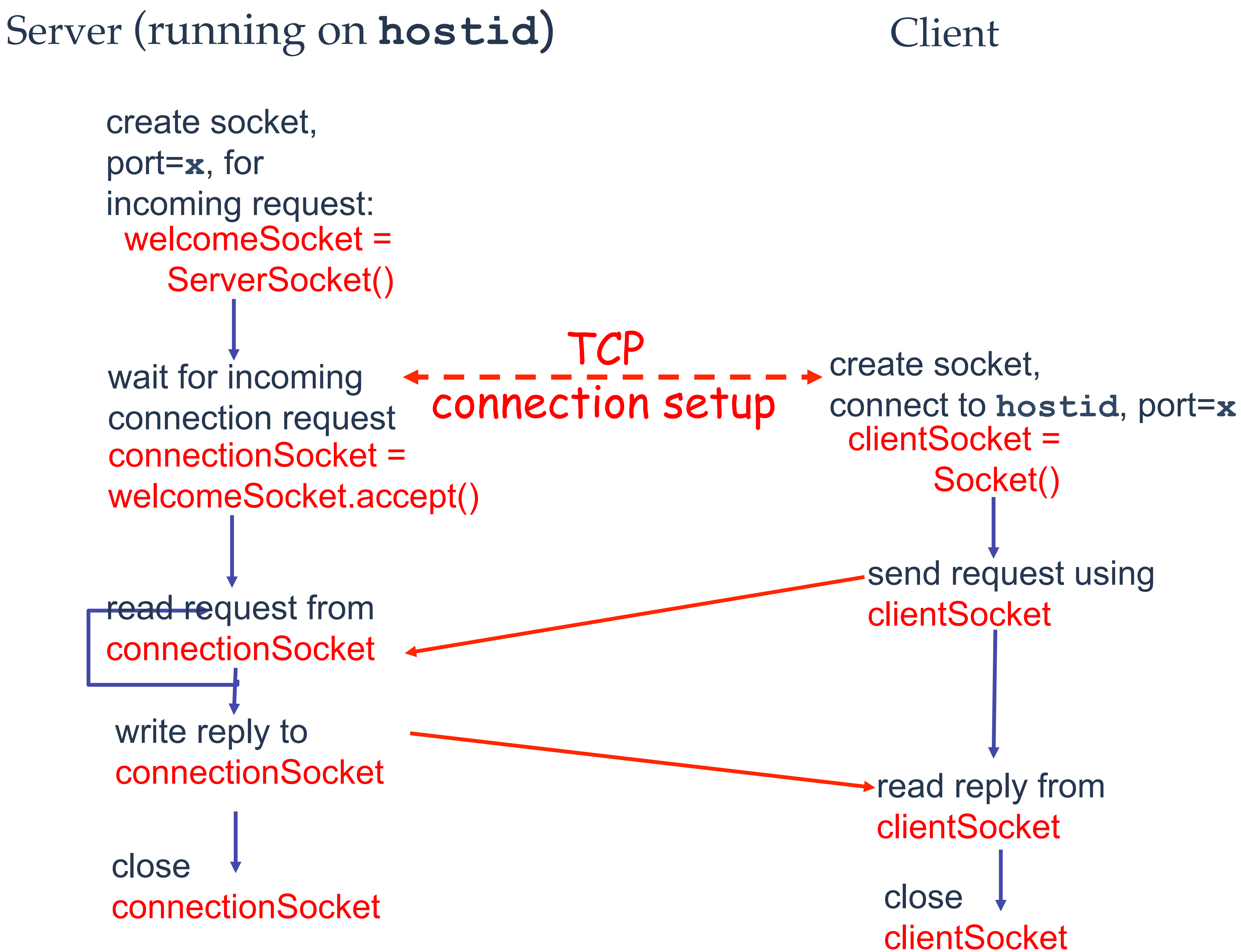
- creating client-local TCP socket
- specifying IP address, port number of server process
- when client creates socket: client TCP establishes connection to server TCP

- when contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

application viewpoint

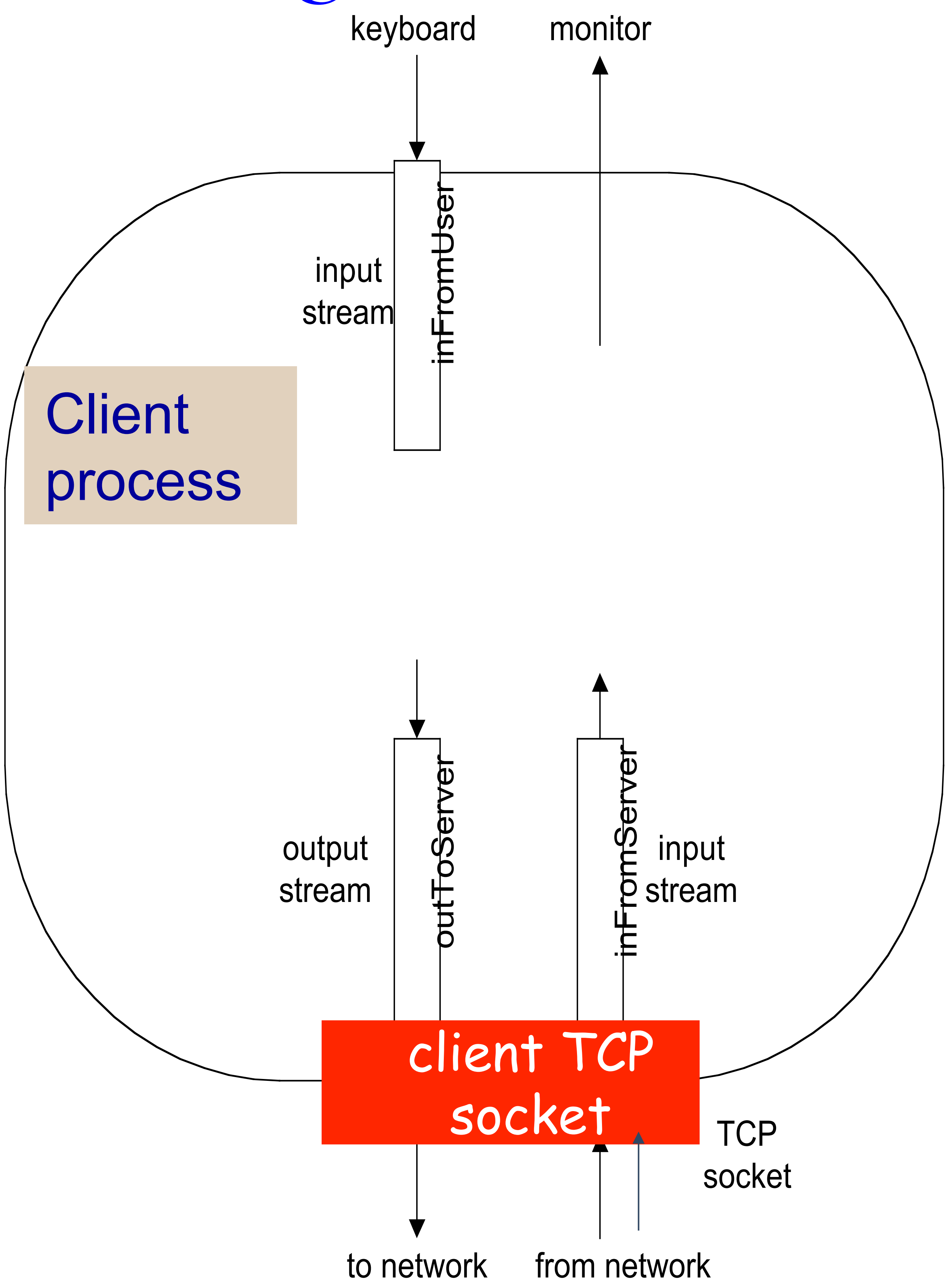
TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

TCP Client/server socket interaction



Stream Jargon

- stream is a sequence of characters that flow into or out of a process.
- input stream is attached to some input source for the process, e.g., keyboard or socket.
- output stream is attached to an output source, e.g., monitor or socket.



Socket programming with TCP

Example client-server app:

1) client reads line from standard input (**inFromUser** stream) , sends to server via socket (**outToServer** stream)

2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (**inFromServer** stream)

Example: TCP Java client

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());
```

This package defines Socket() and ServerSocket() classes

server name, e.g., www.umass.edu

server port #

create input stream

create clientSocket object of type Socket, connect to server

create output stream attached to socket

Example: TCP Java client (cont'd)

create
input stream
attached to socket →

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();
```

send line
to server →

```
outToServer.writeBytes(sentence + '\n');
```

read line
from server →

```
modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);
```

close socket
(clean up behind yourself!) →

```
clientSocket.close();

  }
}
```

Example: TCP Java server

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

        Socket connectionSocket = welcomeSocket.accept();

       BufferedReader inFromClient =
         new BufferedReader(new
         InputStreamReader(connectionSocket.getInputStream()));
```

create welcoming socket at port 6789

wait, on welcoming socket accept() method for client contact create, *new* socket on return

create input stream, attached to socket

Example: TCP Java server (cont'd)

create output stream, attached to socket

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());
```

read in line from socket

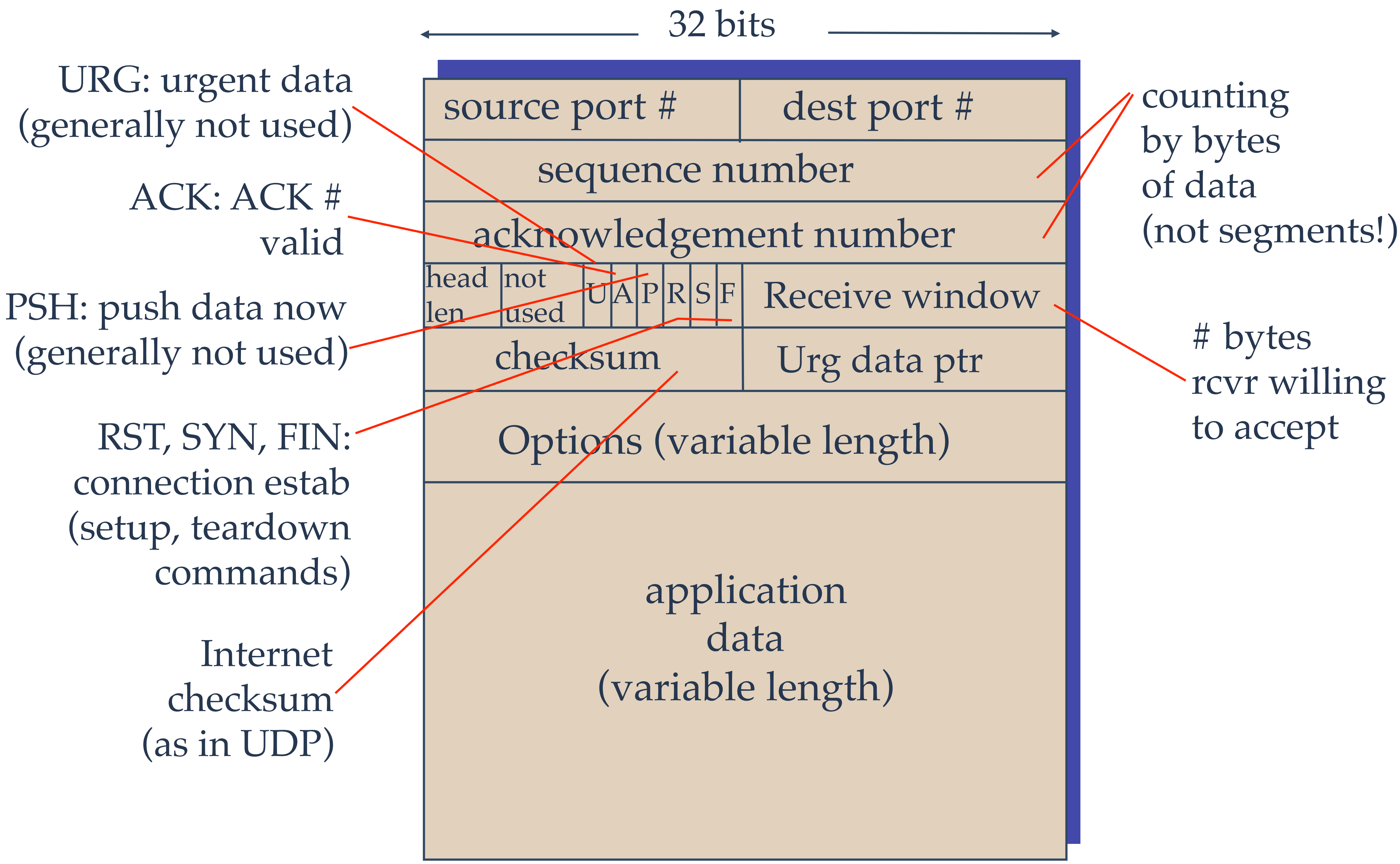
```
clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

write out line to socket

```
outToClient.writeBytes(capitalizedSentence);
      }
    }
  }
```

end of while loop,
loop back and wait for
another client connection

TCP segment structure



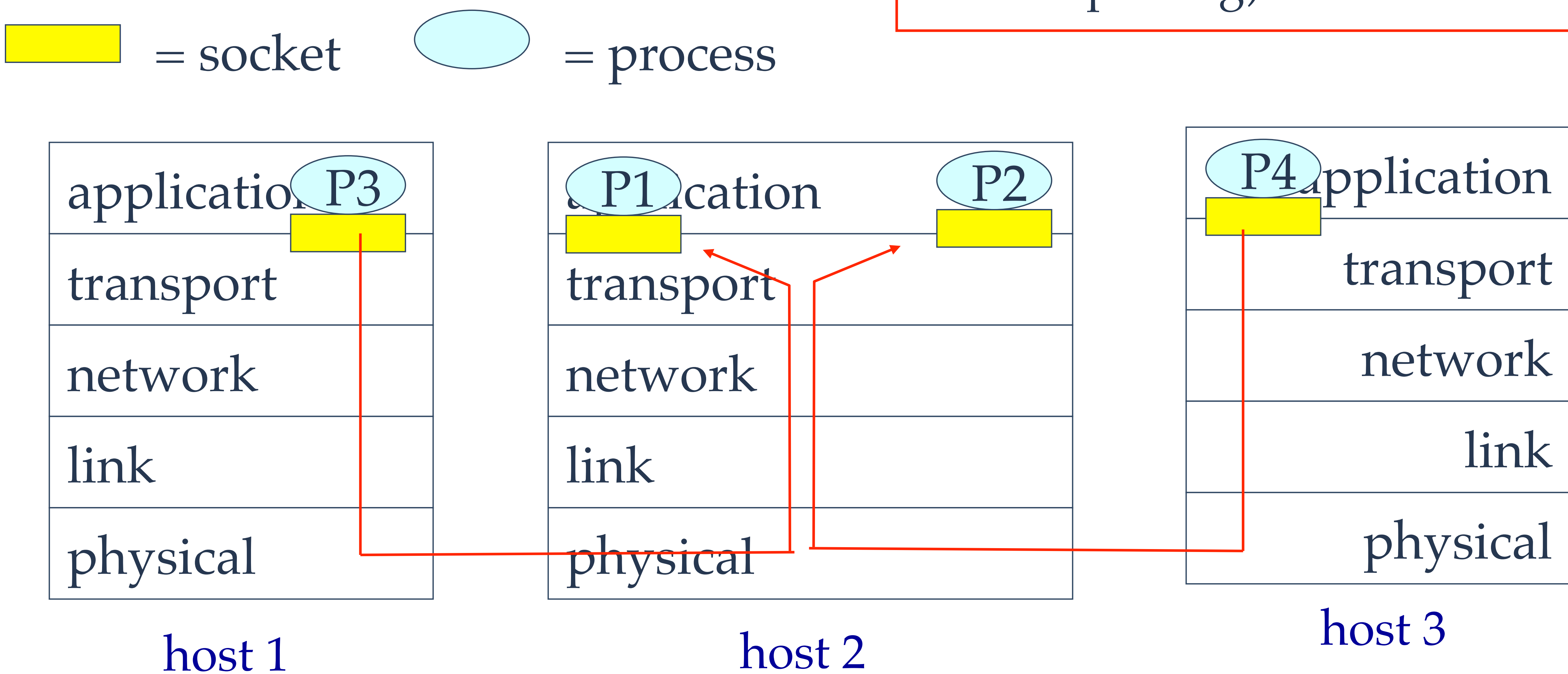
Multiplexing/demultiplexing

<u>Demultiplexing at rcv host:</u>

delivering received segments to correct socket

<u>Multiplexing at send host:</u>

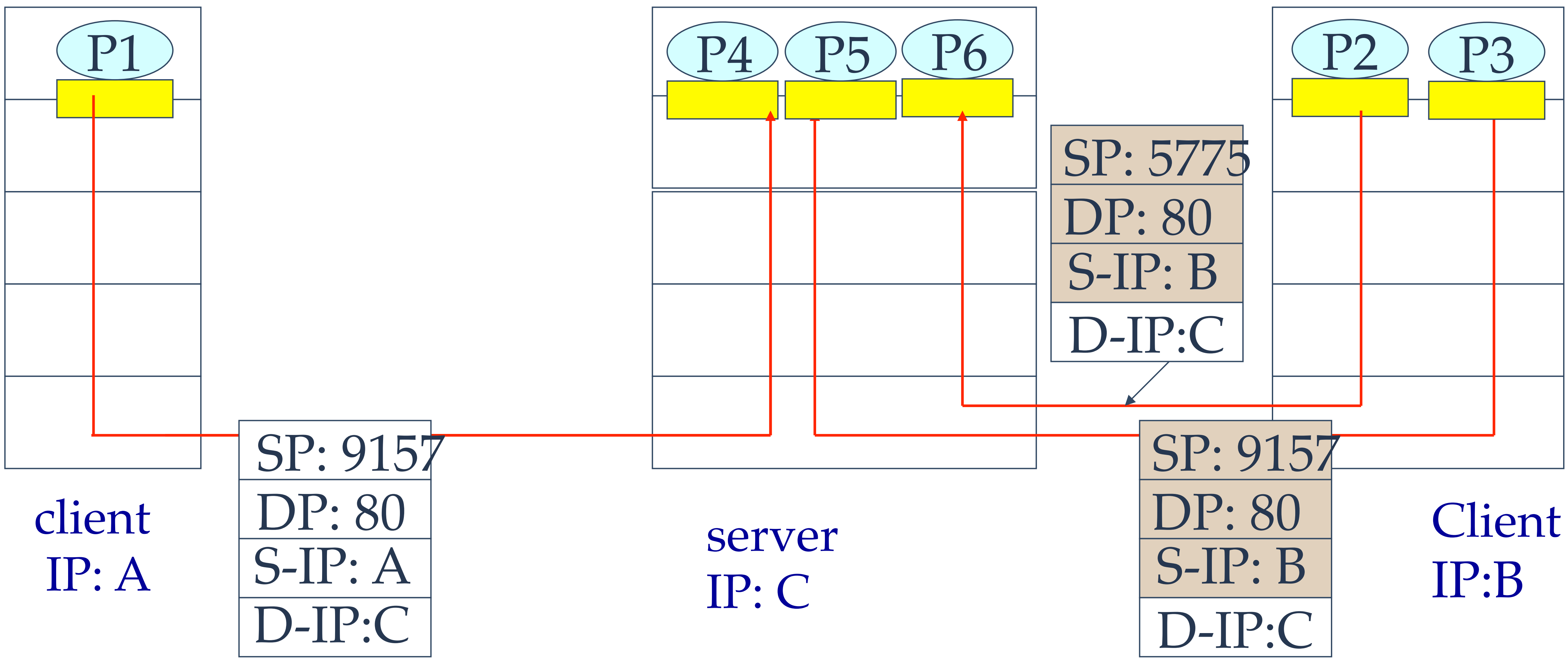
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



Connection-oriented demux

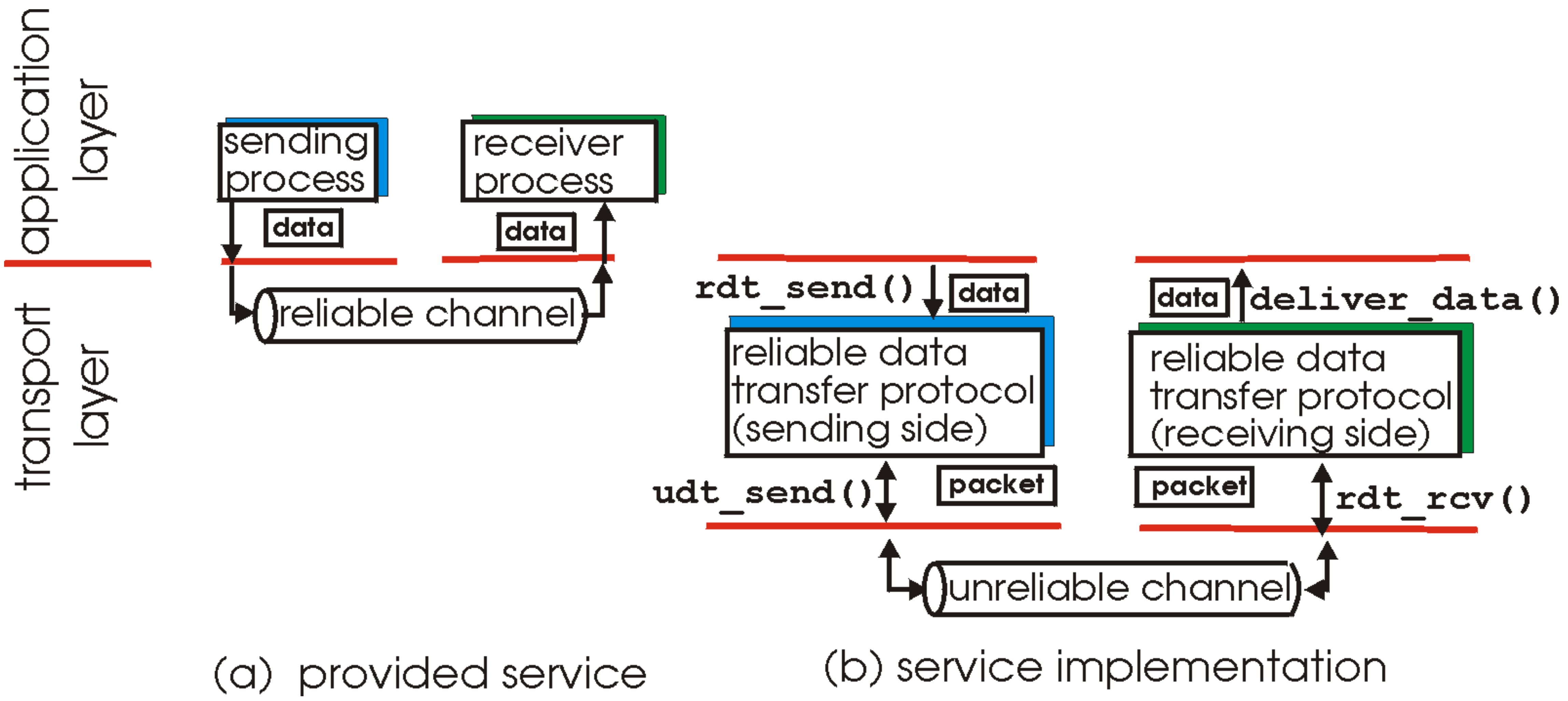
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux (cont)



Principles of Reliable Data Transfer

- important in app., transport, link layers

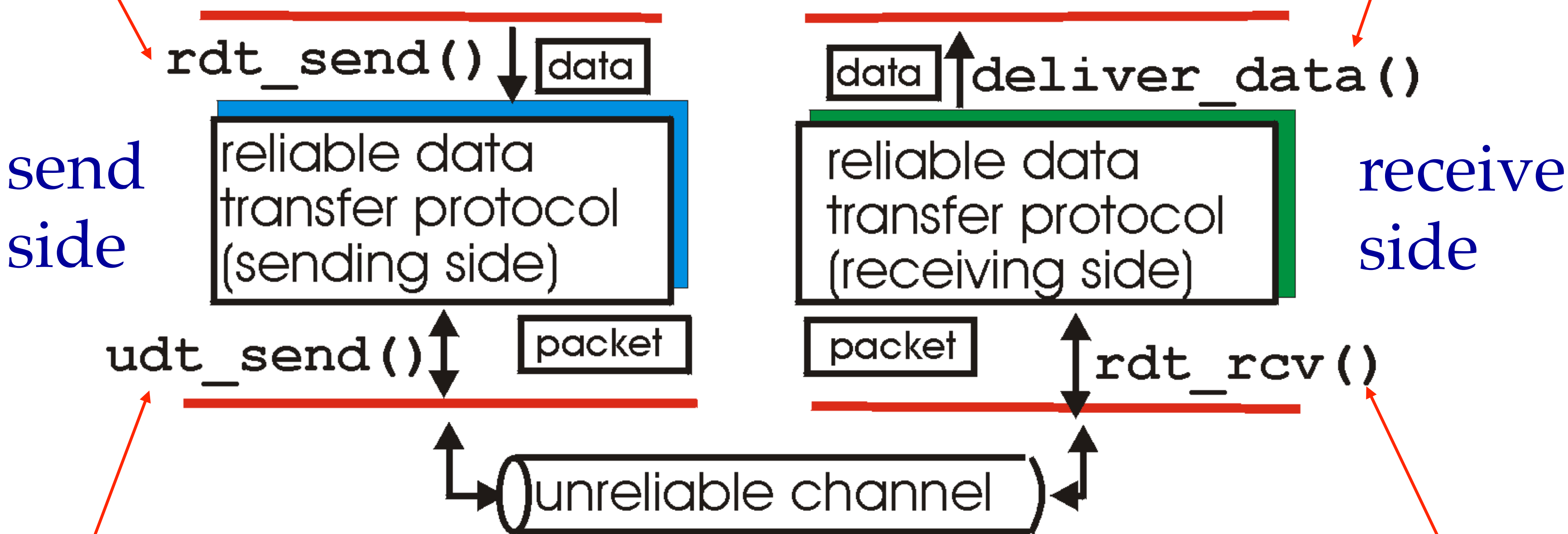


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol
- Note: slides use the term "packet" but at transport layer, these are segments

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper

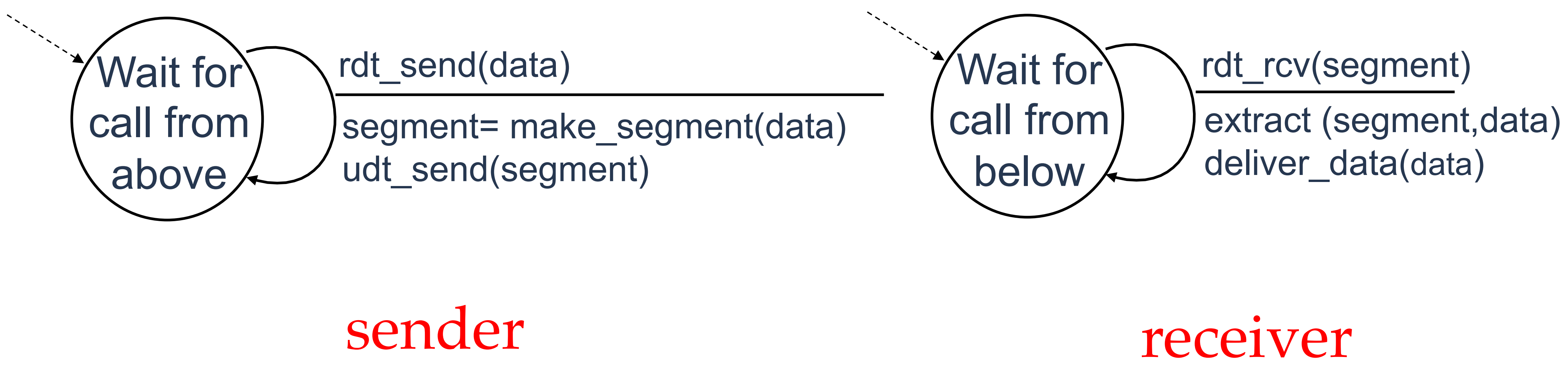


udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



What can go wrong (1)?

- Underlying channel may flip bits in segment
- Error detection:
  - Checksum to detect bit errors
- Recovering from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that segment received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that segment had errors
  - sender retransmits segment on receipt of NAK
- Stop-and-wait
  - Sender sends one segment, then waits for the receiver to respond
  - We will come back to this later

Handling duplicates

- What happens if ACK/NAK corrupted?
  - sender doesn't know what happened at receiver!
  - can't just retransmit: possible duplicate
- Sender retransmits current segment if ACK/NAK garbled
- Sender adds sequence number to each segment
  - For stop-and-go protocol a 1-bit sequence number with modulo-2 arithmetic is sufficient
- Receiver discards (does not deliver up to the application) duplicate segments

NAK-free Protocol

- It is possible to eliminate NAKs as negative acknowledgement
- instead of NAK, receiver sends ACK for last segment received OK
  - ➡ receiver must *explicitly* include sequence number of segment being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current packet*

What can go wrong (2)?

- Segments (data or ACK) may be lost
- Sender waits a “reasonable” amount of time for ACK
  - Retransmits if no ACK received in this time
  - If segment (data or ACK) simply delayed (not lost)
    - Retransmission will be duplicate, but use of sequence numbers already handles this
    - Receiver must specify the sequence number of segment being ACKed
  - Requires countdown timer
- Sequence numbers
  - For data: byte stream “number” of first byte in segment’s data
  - For ACKs: if pipelined, segment number of next byte expected from other side
    - Cumulative ACK

What can go wrong (3)?

- Data segments may come out of order
- TCP specification does not say what to do
- Two alternatives
  - ➡ Receiver discards out of order segments
    - ✦ Simplifies receiver design
    - ✦ Wastes network bandwidth
  - ➡ Receiver keeps out of order segments and fills in the missing ones when they arrive
    - ✦ Usually this is what is implemented

TCP sender events:

data rcvd from app:

- Create segment with sequence number
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

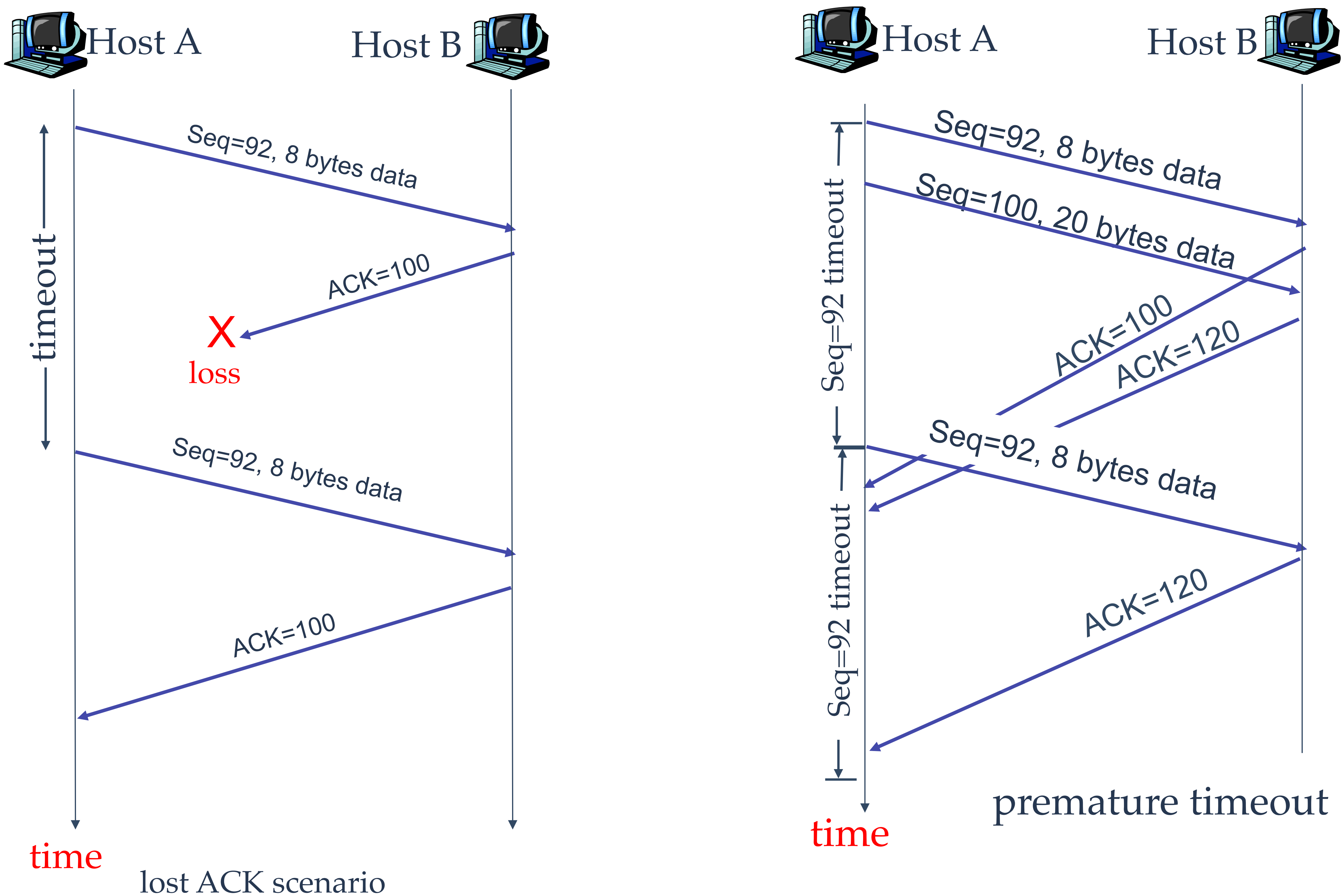
Ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

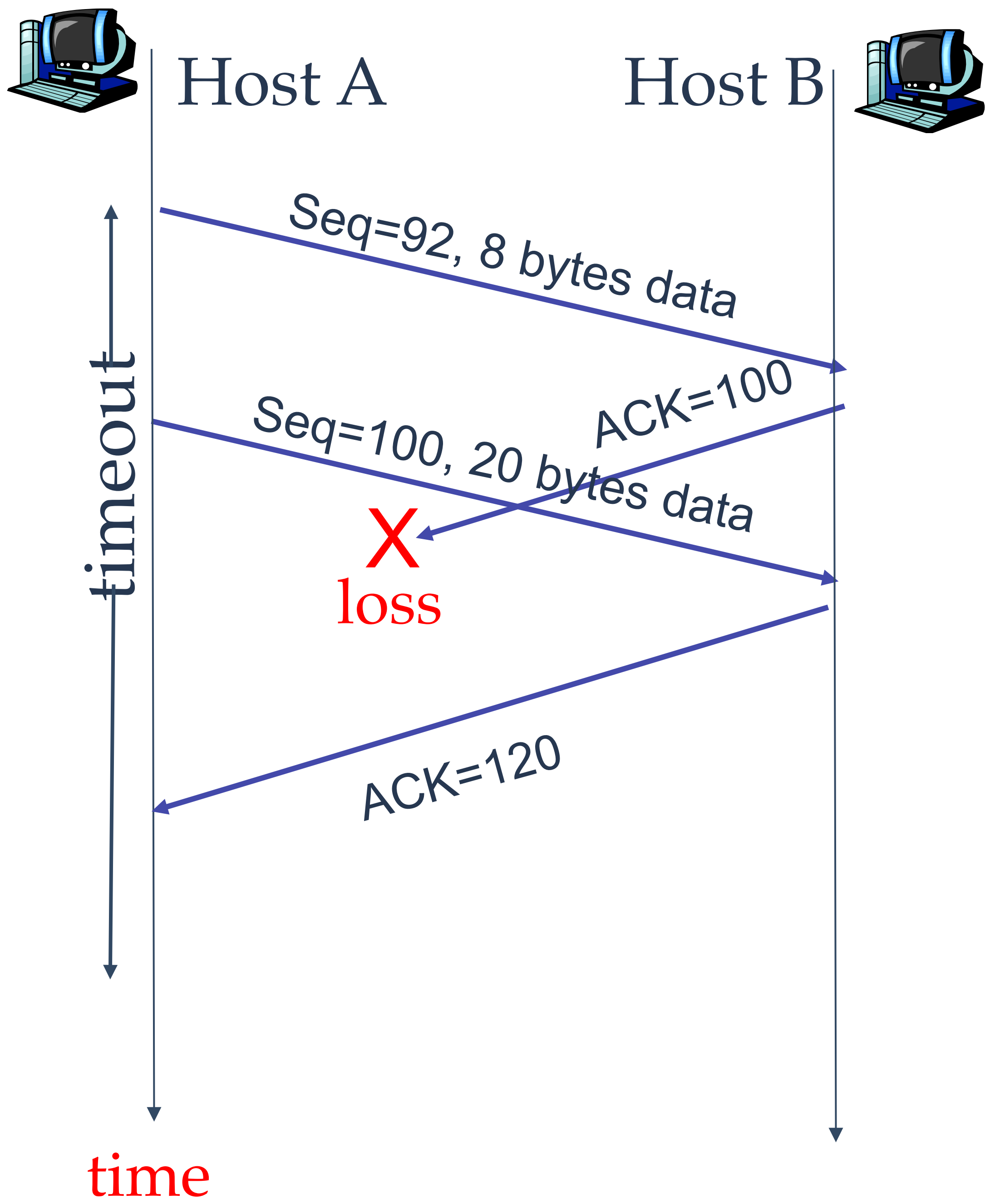
TCP sender
(simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
           smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
         if (there are currently not-yet-acknowledged segments)
              start timer
         }

}  /* end of loop forever */
```

TCP: retransmission scenarios



TCP retransmission scenarios (more)



Cumulative ACK scenario

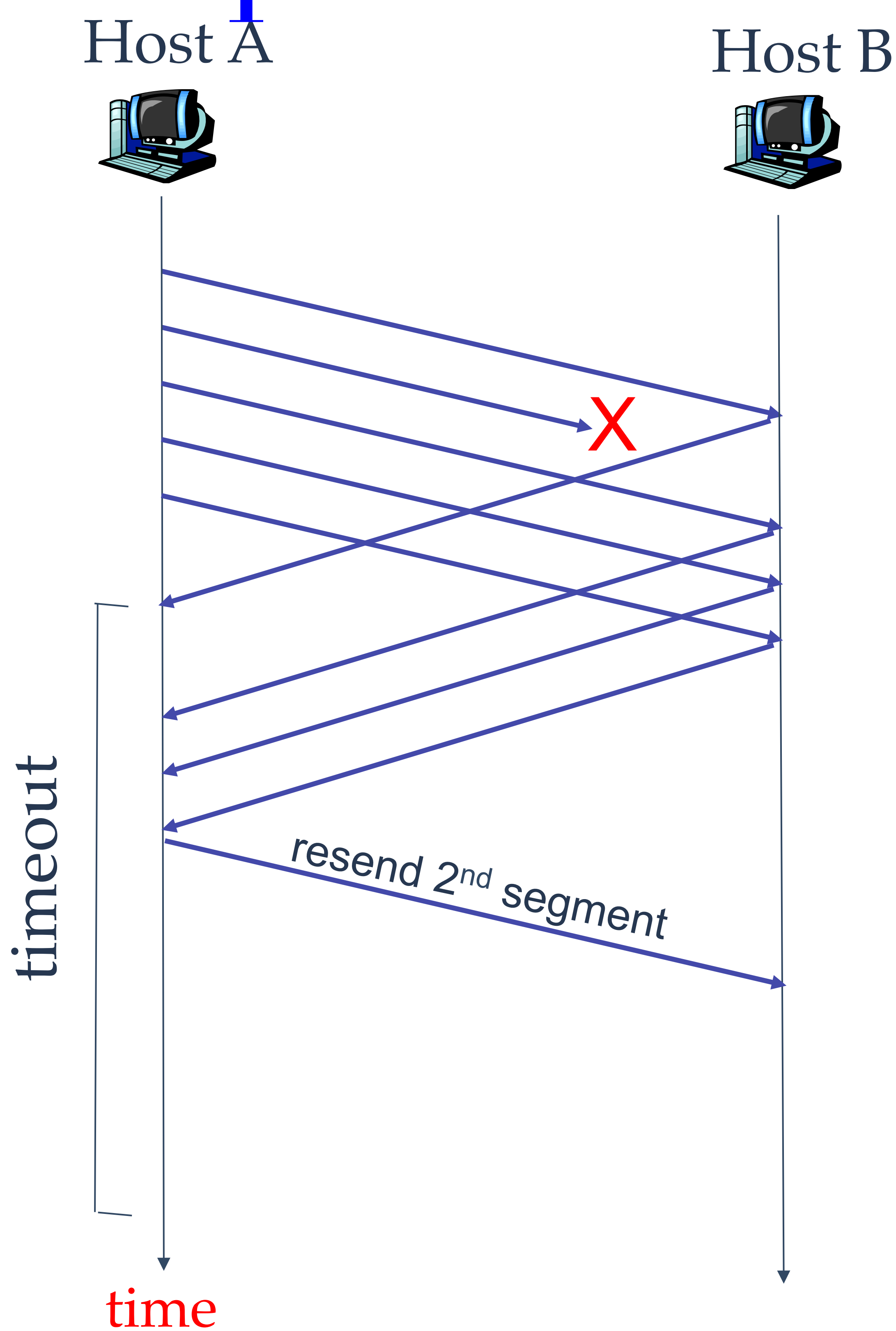
TCP ACK generation

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 starts at lower end of gap

Fast Retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.
- if sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Resending segment after triple duplicate ACK



Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
        if (y > SendBase) {
            SendBase = y
            if (there are currently not-yet-acknowledged segments)
                start timer
          }
        else {
             increment count of dup ACKs received for y
             if (count of dup ACKs received for y = 3) {
                 resend segment with sequence number y
             }
```

a duplicate ACK for already ACKed segment

fast retransmit

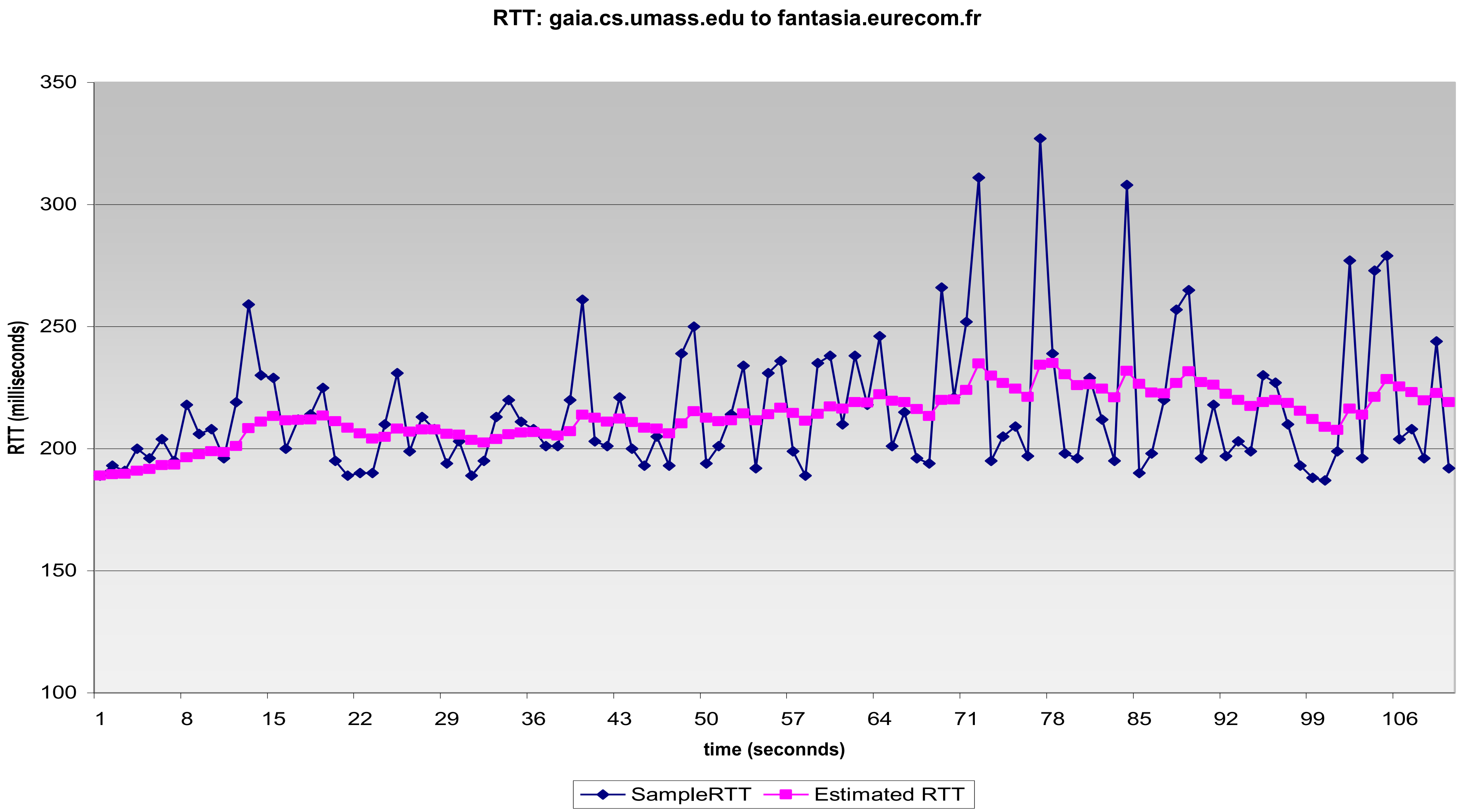
TCP Round Trip Time and Timeout

- How to set TCP timeout value?
  - longer than RTT
    - but RTT varies
  - too short: premature timeout
    - unnecessary retransmissions
  - too long: slow reaction to segment loss
- How to estimate RTT?
  - **`SampleRTT`**: measured time from segment transmission until ACK receipt
    - ignore retransmissions
  - **`SampleRTT`** will vary, want estimated RTT “smoother”
    - average several recent measurements, not just current **`SampleRTT`**

TCP Round Trip Time and Timeout

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: **α=** 0.125



TCP Round Trip Time and Timeout

Setting the timeout

- `EstimatedRTT` plus "safety margin"
  - ➡ large variation in `EstimatedRTT ->` larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

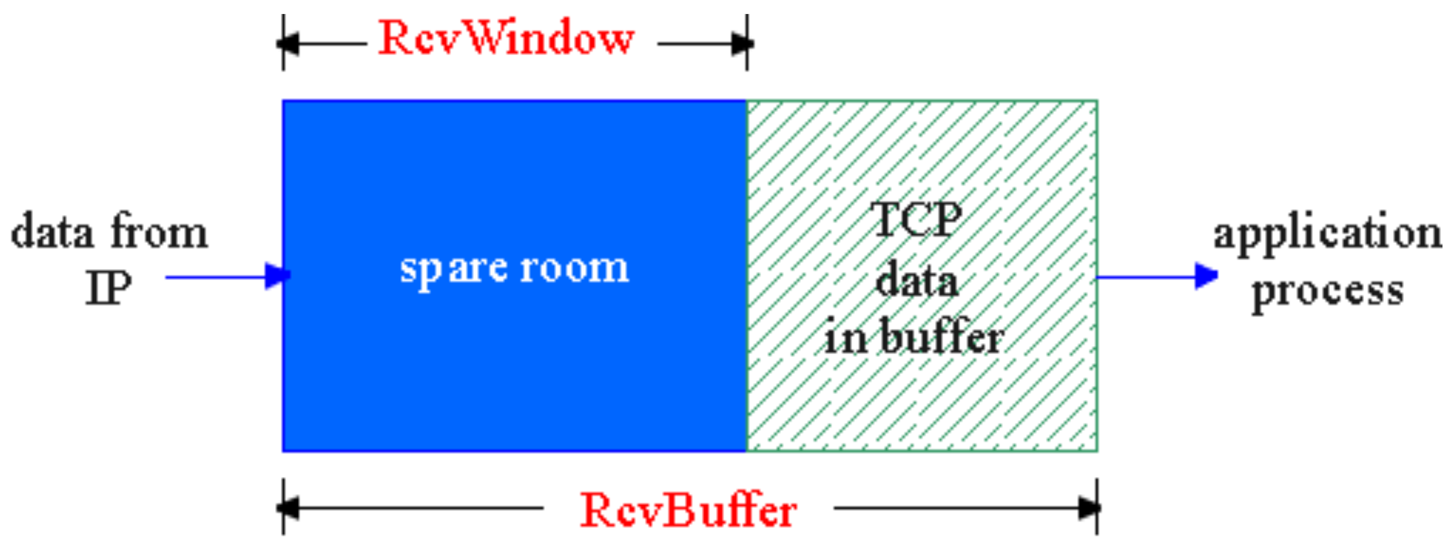
```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(typically, β= 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

TCP Flow Control

- receive side of TCP connection has a receive buffer:



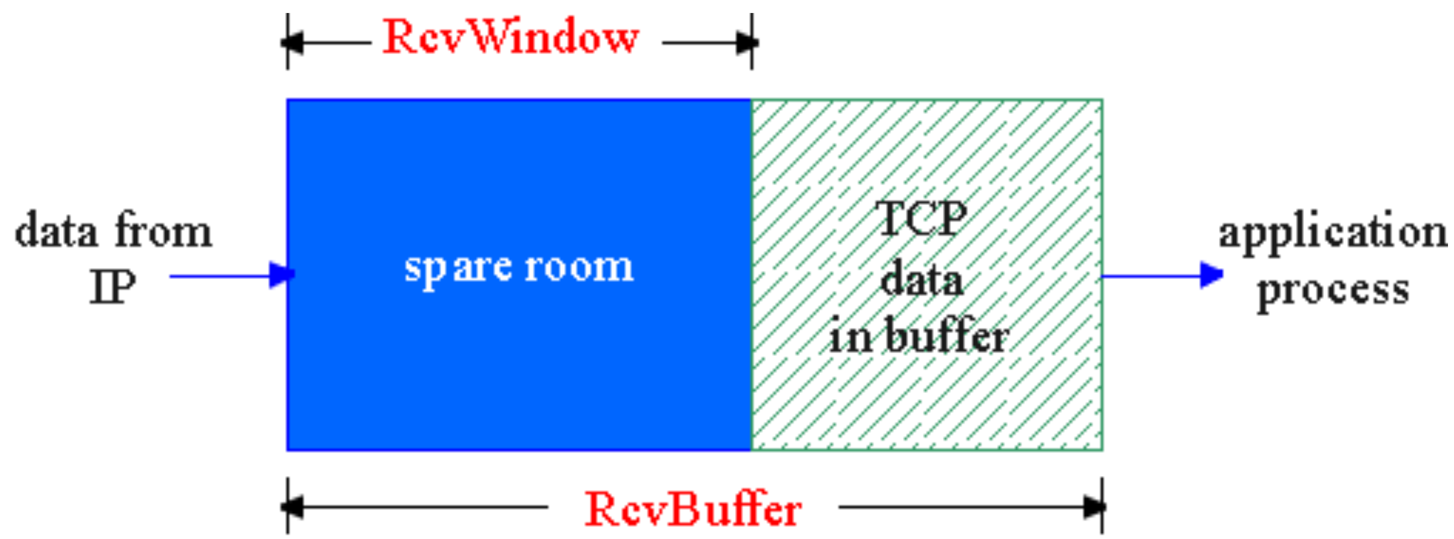
- app process may be slow at reading from buffer

flow control

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

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



- spare room in buffer (ignoring out-of-order segments)

  `RcvWindow = RcvBuffer-[LastByteRcvd - LastByteRead]`

- Receiver advertises spare room by including value of **`RcvWindow`** in segments
- Sender limits unACKed data to **`RcvWindow`**
  - ➡ guarantees receive buffer doesn't overflow

TCP Connection Management

Recall: TCP is a connection-oriented protocol

- sender, receiver establish "connection" before exchanging data segments
- initialize TCP variables:
  - ➡ Sequence numbers
  - ➡ buffers, flow control info (e.g. **`RcvWindow`**)
- *client:* connection initiator

```
Socket clientSocket = new Socket("hostname","port number");
```

- *server:* contacted by client

```
Socket connectionSocket = welcomeSocket.accept();
```

Three-Way Handshake

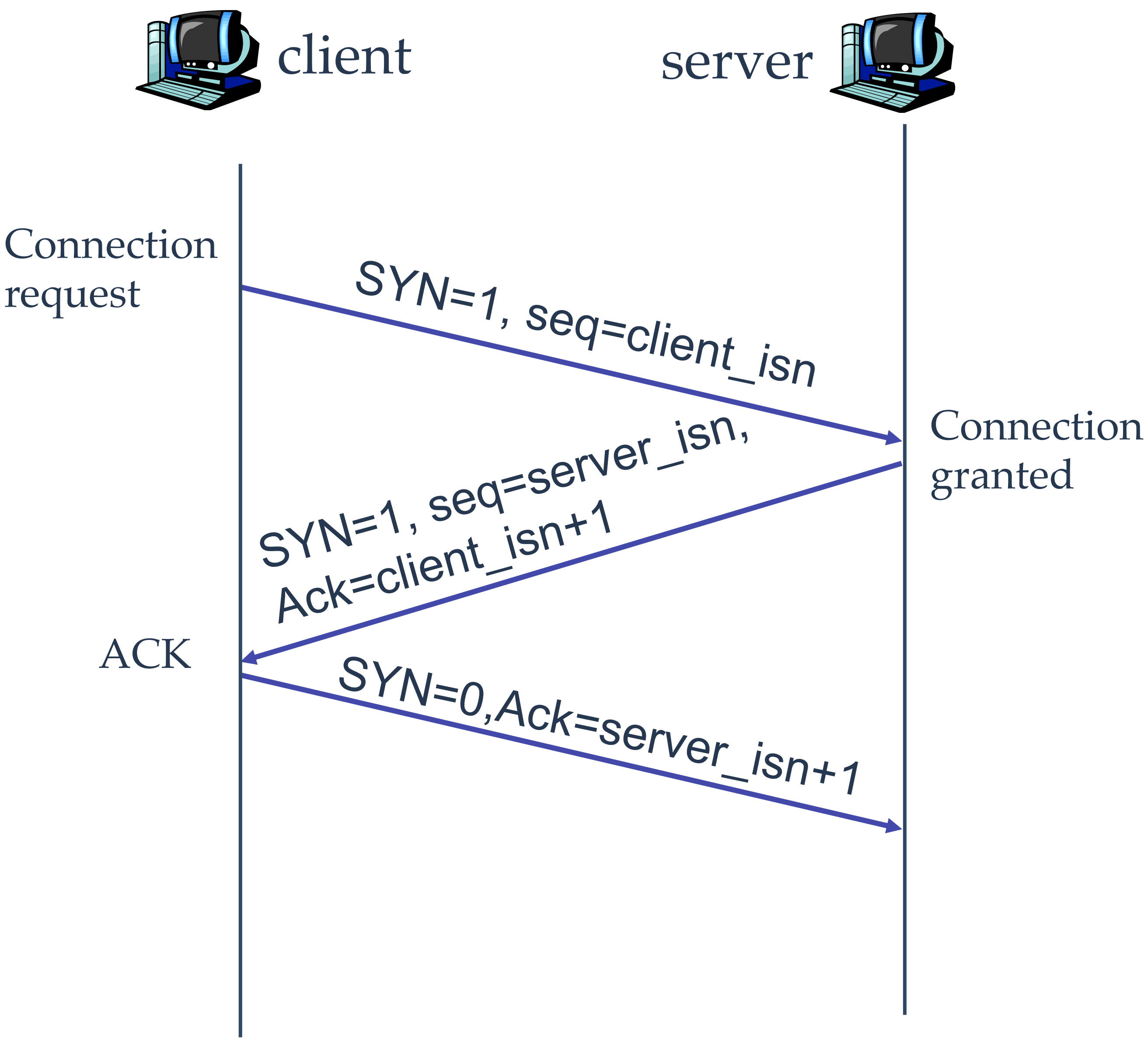
Step 1: client host sends TCP SYN segment to server

- specifies initial sequence number
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial sequence number

Step 3: client receives SYNACK, replies with ACK segment, which may contain data



Closing a TCP Connection

client closes socket:

```
clientSocket.close();
```

Step 1: client end system sends TCP FIN control segment to server

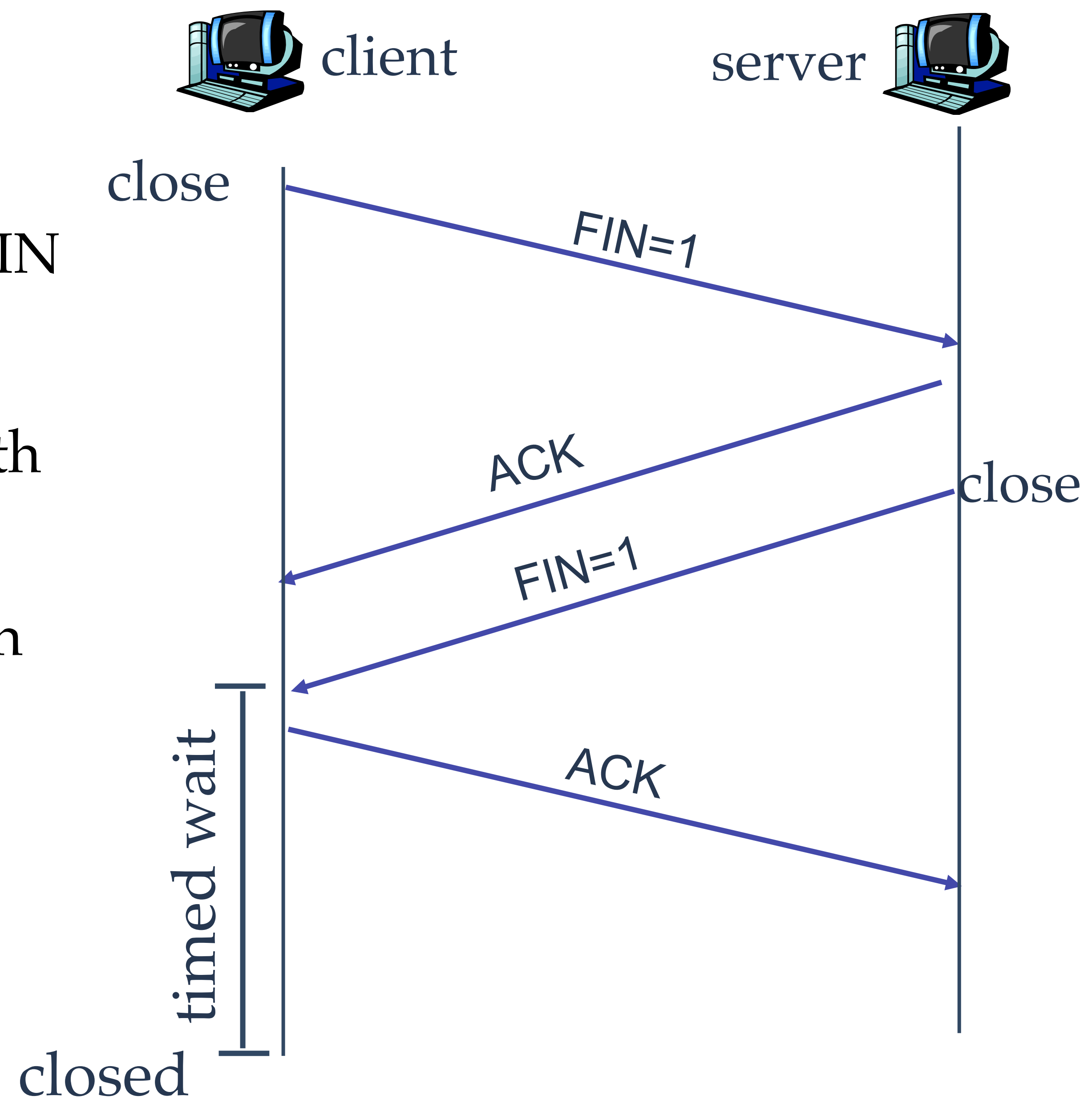
Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.

Step 3: client receives FIN, replies with ACK.

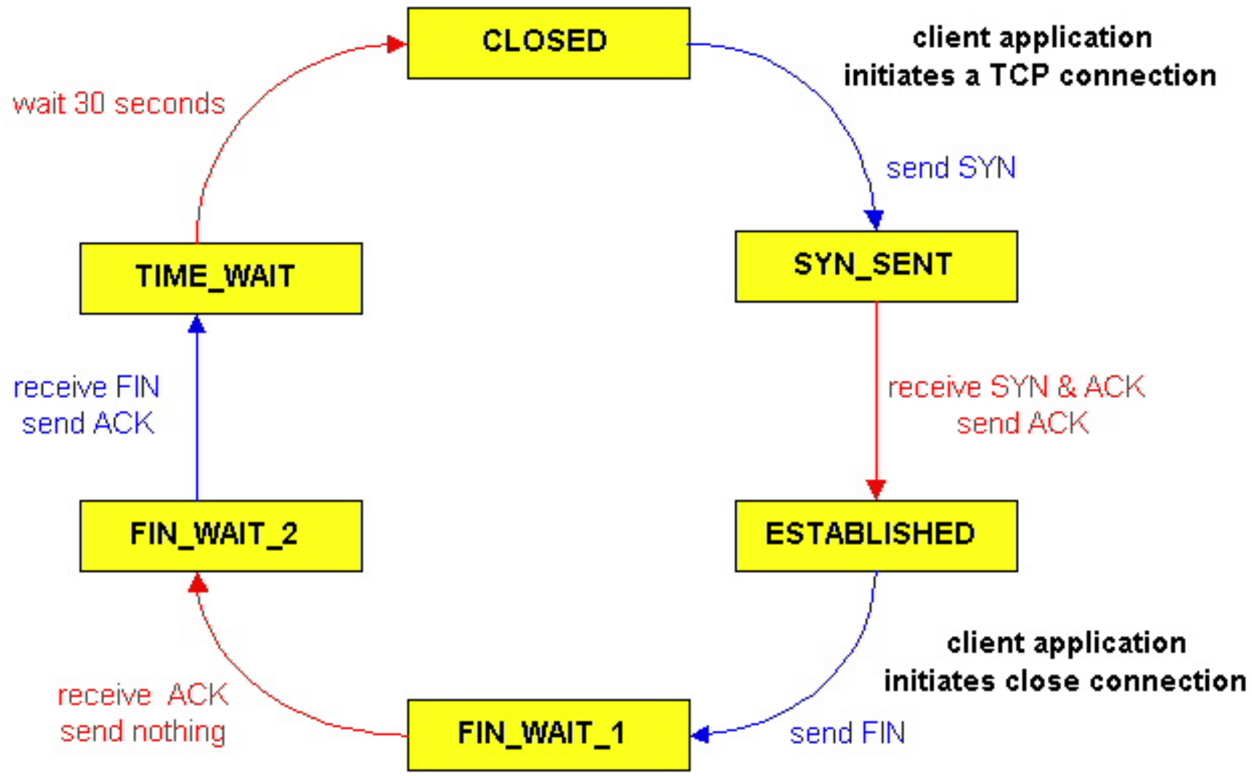
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

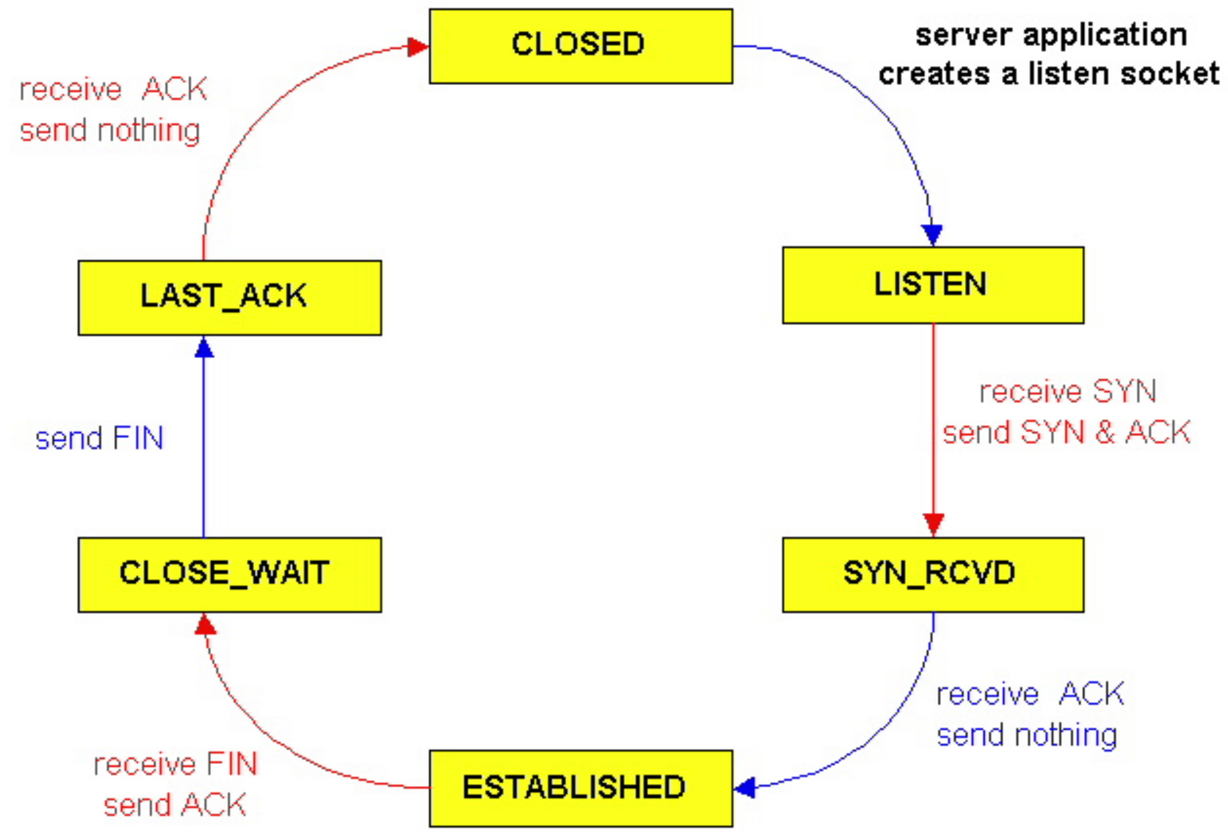


TCP Connection States



TCP client lifecycle

TCP server lifecycle

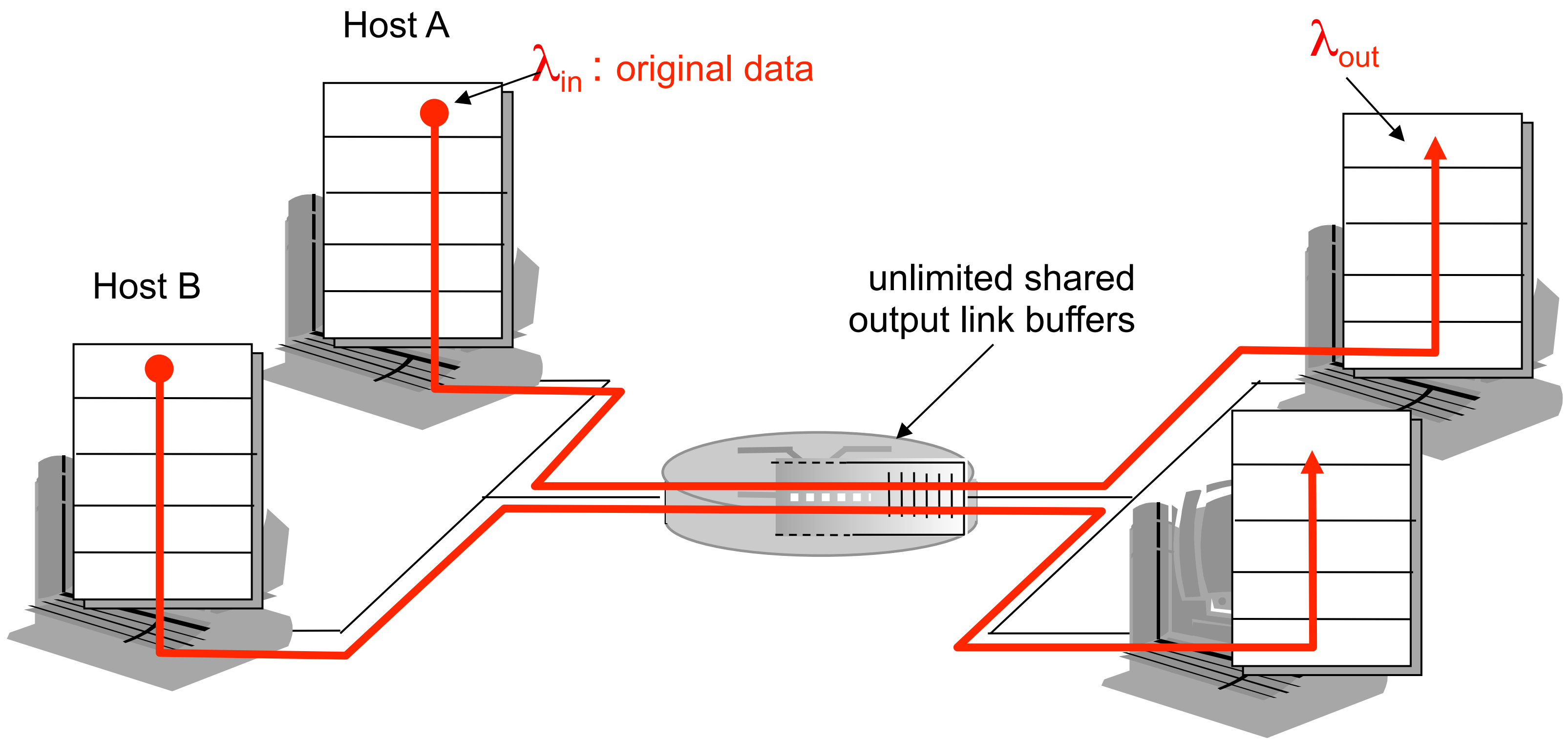


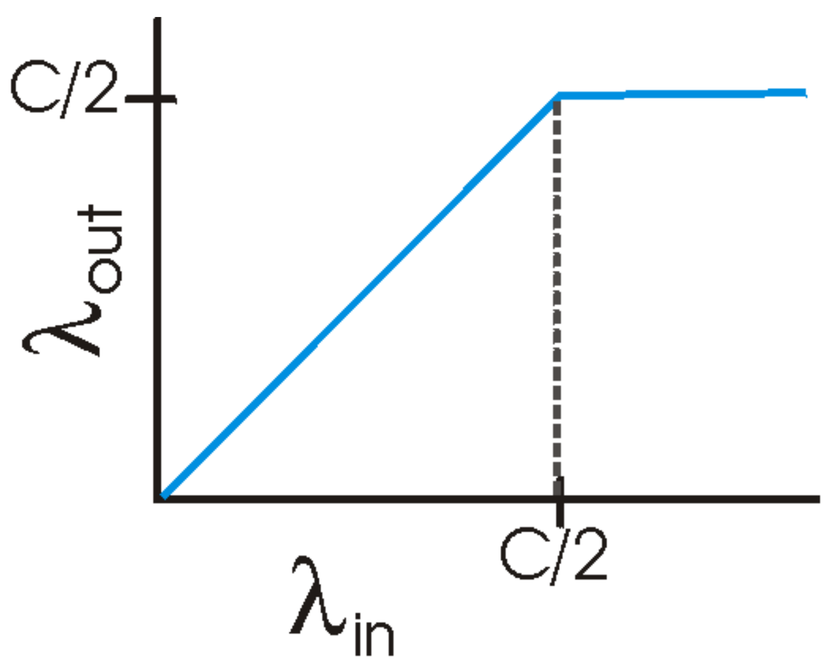
Principles of Congestion Control

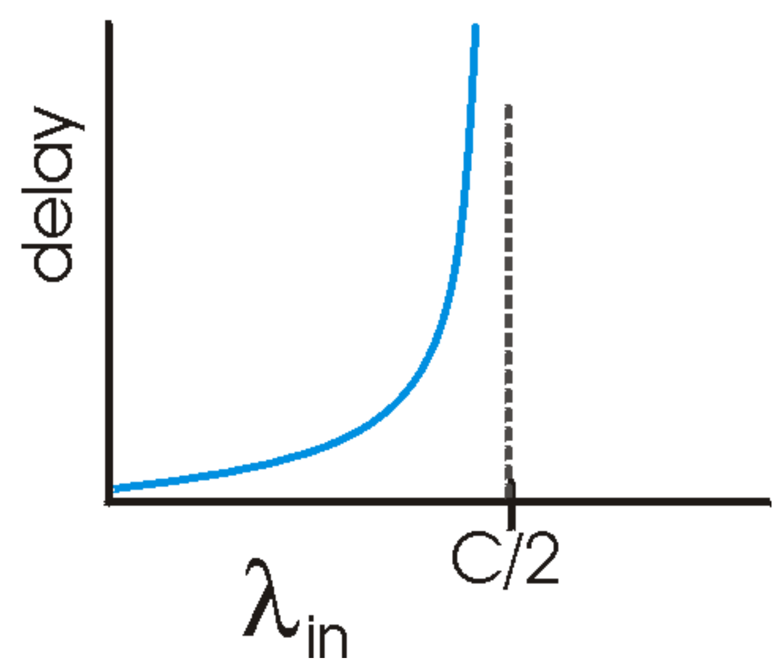
- Congestion: informally, “too many sources sending too much data too fast for the *network* to handle”
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

Causes/costs of congestion: Simple Scenario

- two senders, two receivers
- one router, infinite buffers
- no retransmission







- large delays when congested
- maximum achievable throughput

Causes and Effects of Congestion

- With finite buffers at routers, packets may be dropped as λ_{in} increases
  - ➡ Retransmission needed
  - ➡ Offered load $\lambda'_{in} > \lambda_{in}$
  - – More work (retransmissions) for given λ_{out}
  - – Unneeded retransmissions: link carries multiple copies of segment
- With multi-hop connections, upstream routers receive two types of traffic:
  - ➡ Forwarded traffic from downstream routers
  - ➡ Traffic they may receive directly from hosts
  - ➡ As λ'_{in} increases, more dropped packages and more transmissions
    - ✦ λ_{out} will approach 0
  - – When packet dropped, any upstream transmission capacity used for that packet was wasted!

Approaches to Congestion Control

Two broad approaches towards congestion control:

1. end-end congestion control:
   - no explicit feedback from network
   - congestion inferred from end-system observed loss, delay
   - approach taken by TCP
2. network-assisted congestion control:
   - routers provide feedback to end systems
     - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
     - explicit rate sender should send at

TCP congestion control

- Approach: increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - Additive increase: increase `cwnd` by 1 MSS (maximum segment size) every RTT until loss detected
  - Multiplicative decrease: cut `cwnd` in half after loss
- Algorithm has three components:
  - Slow start
    - Initially set `cwnd` = 1 MSS
    - double **`cwnd`** every RTT
    - done by incrementing **`cwnd`** for every ACK received
  - Congestion avoidance: After timeout or 2 duplicate ACKS
    - **`cwnd`** is cut in half
    - window then grows linearly
  - Fast recovery: After a timeout
    - **`cwnd`** set to 1 MSS;
    - window then grows exponentially
    - to a threshold, then grows linearly