

# Computer Networks and Applications

COMP 3331/COMP 9331

Week 5

## P2P + Transport Layer Part 1

**Reading Guide: Chapter 2, 2.5 + Chapter 3, Sections 3.1 – 3.4**

## 2. Application Layer: outline

### 2.1 principles of network applications

- app architectures
- app requirements

### 2.2 Web and HTTP

### 2.3 electronic mail

- SMTP, POP3, IMAP

### 2.4 DNS

### 2.5 P2P applications

### 2.6 video streaming and content distribution networks (CDNs)

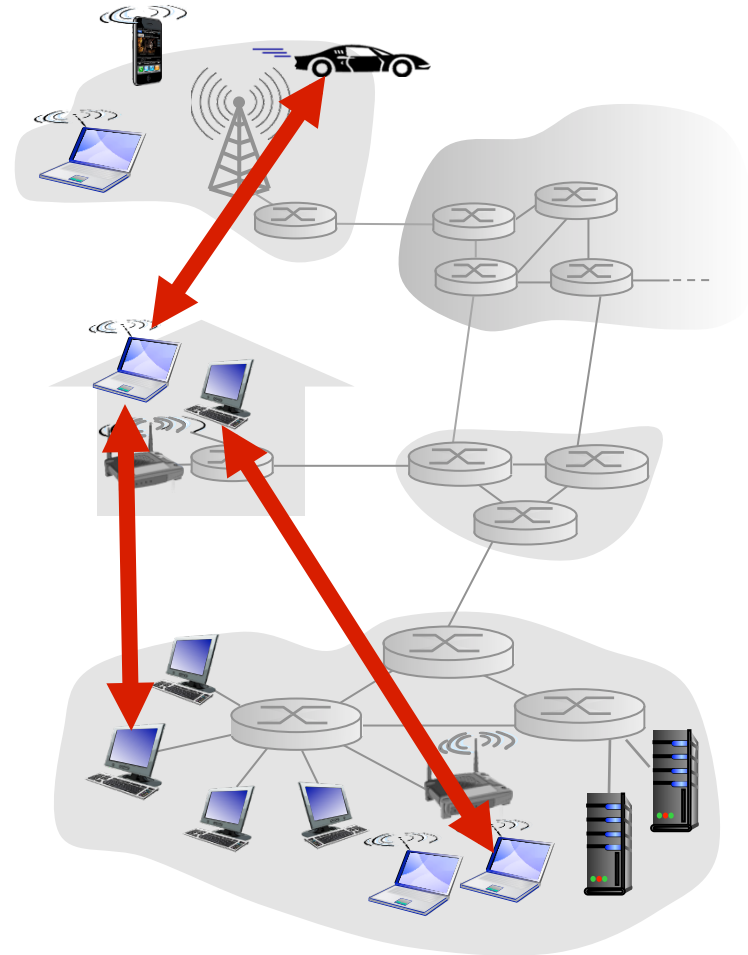
### 2.7 socket programming with UDP and TCP

# P2P architecture

- ❖ no always-on server
- ❖ arbitrary end systems directly communicate
- ❖ peers are intermittently connected and change IP addresses

## *examples:*

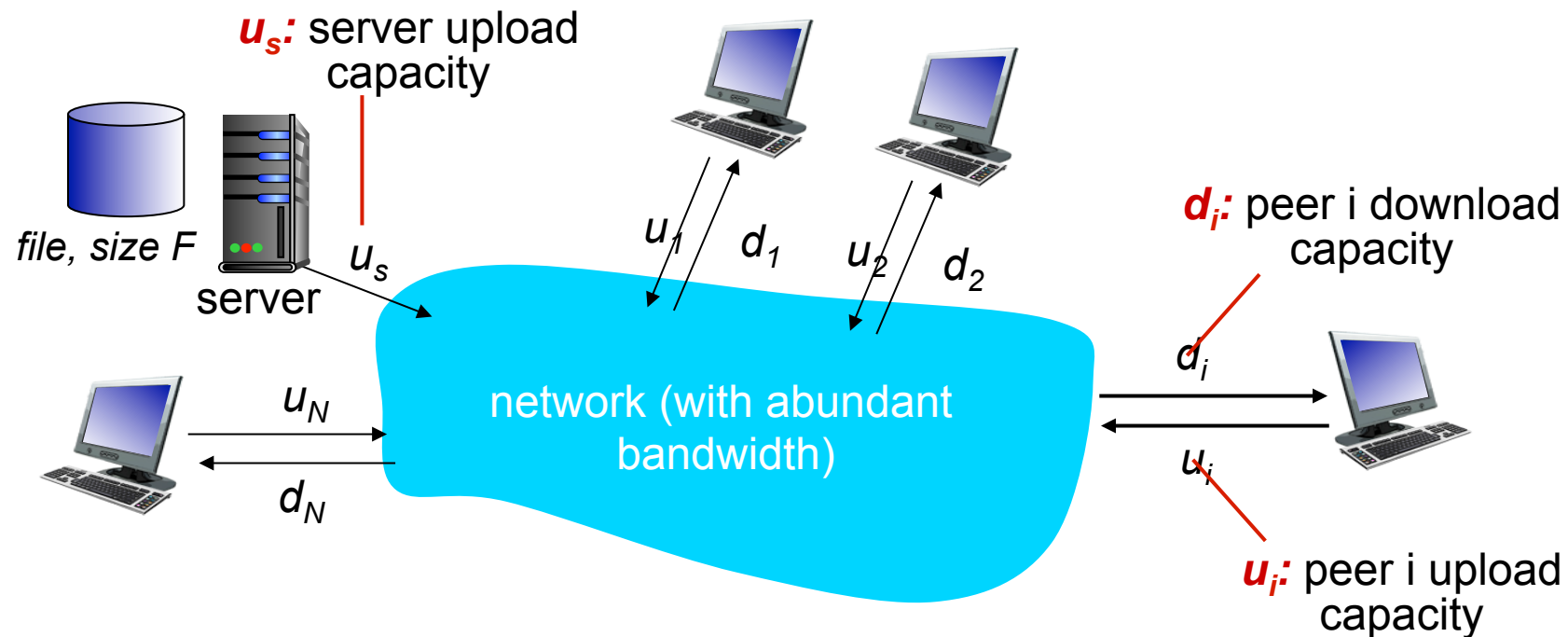
- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)



# File distribution: client-server vs P2P

Question: how much time to distribute file (size  $F$ ) from one server to  $N$  peers?

- peer upload/download capacity is limited resource



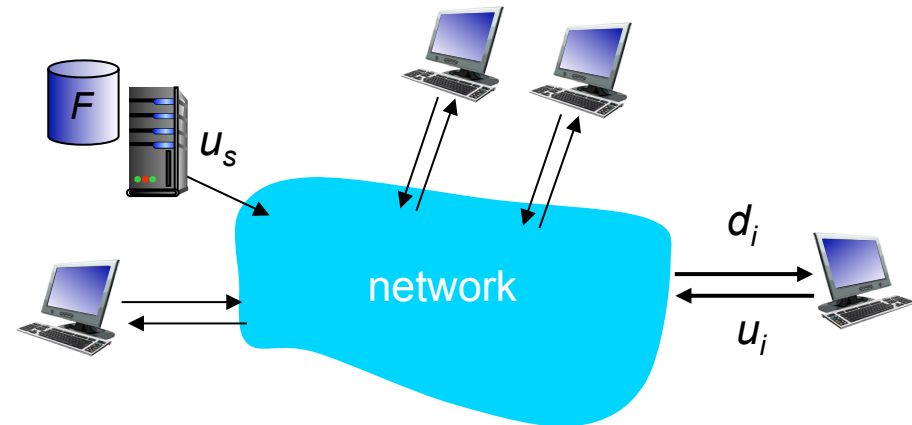
# File distribution time: client-server

❖ **server transmission:** must sequentially send (upload)  $N$  file copies:

- time to send one copy:  $F/u_s$
- time to send  $N$  copies:  $NF/u_s$

❖ **client:** each client must download file copy

- $d_{\min}$  = min client download rate
- min client download time:  $F/d_{\min}$



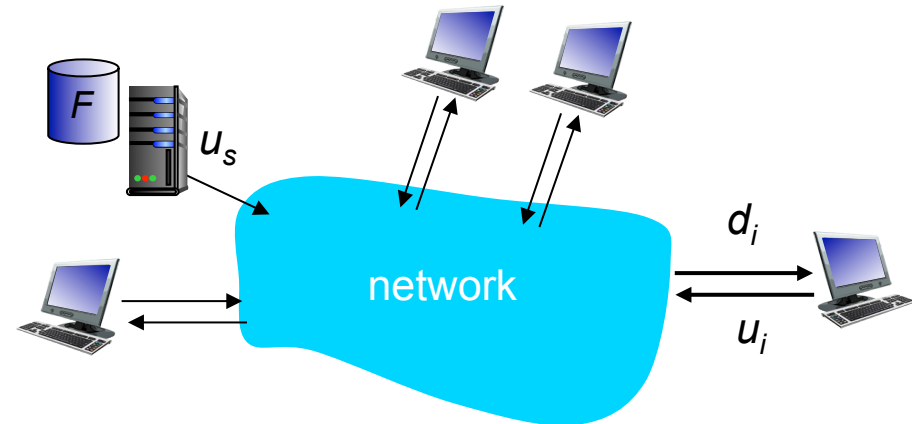
*time to distribute  $F$   
to  $N$  clients using  
client-server approach*

$$D_{c-s} \geq \max\{NF/u_s, F/d_{\min}\}$$

increases linearly in  $N$

# File distribution time: P2P

- ❖ **server transmission:** must upload at least one copy
  - time to send one copy:  $F/u_s$
- ❖ **client:** each client must download file copy
  - min client download time:  $F/d_{\min}$



- ❖ **clients:** as aggregate must download  $NF$  bits
  - max upload rate (limiting max download rate) is  $u_s + \sum u_i$

*time to distribute  $F$   
to  $N$  clients using  
P2P approach*

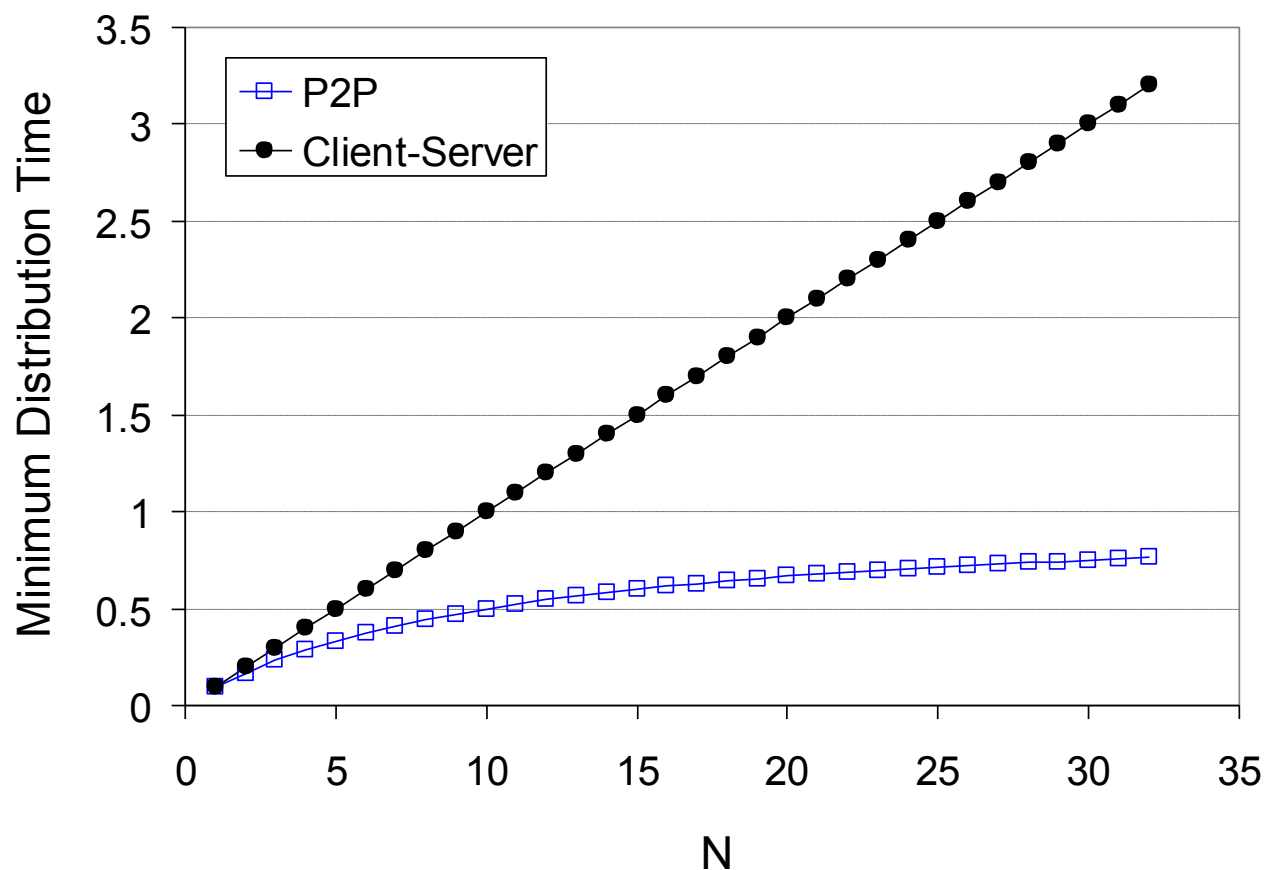
$$D_{P2P} \geq \max\{F/u_s, F/d_{\min}, NF/(u_s + \sum u_i)\}$$

increases linearly in  $N$  ...

... but so does this, as each peer brings service capacity

# Client-server vs. P2P: example

client upload rate =  $u$ ,  $F/u = 1$  hour,  $u_s = 10u$ ,  $d_{min} \geq u_s$

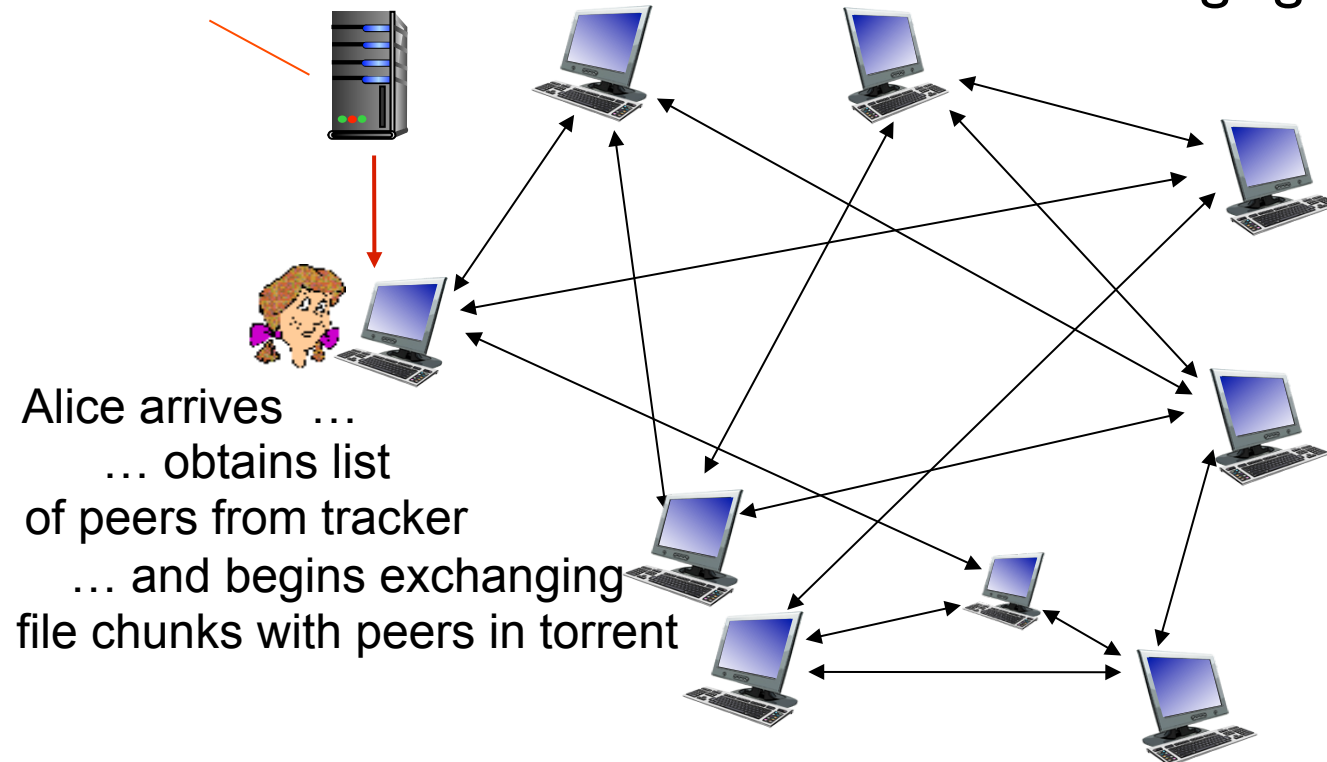


# P2P file distribution: BitTorrent

- ❖ file divided into 256Kb chunks
- ❖ peers in torrent send/receive file chunks

*tracker*: tracks peers participating in torrent

*torrent*: group of peers exchanging chunks of a file

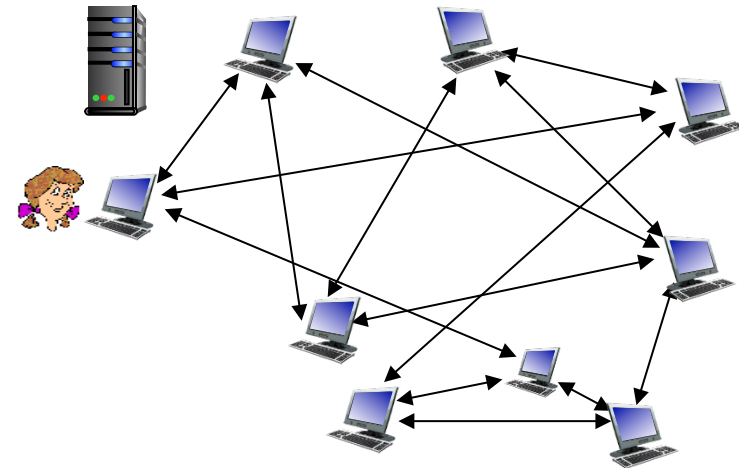




# P2P file distribution: BitTorrent

- ❖ peer joining torrent:

- has no chunks, but will accumulate them over time from other peers
- registers with tracker to get list of peers, connects to subset of peers (“neighbours”)



- ❖ while downloading, peer uploads chunks to other peers
- ❖ peer may change peers with whom it exchanges chunks
  - ❖ *churn*: peers may come and go
- ❖ once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

# BitTorrent: requesting, sending file chunks

## *requesting chunks:*

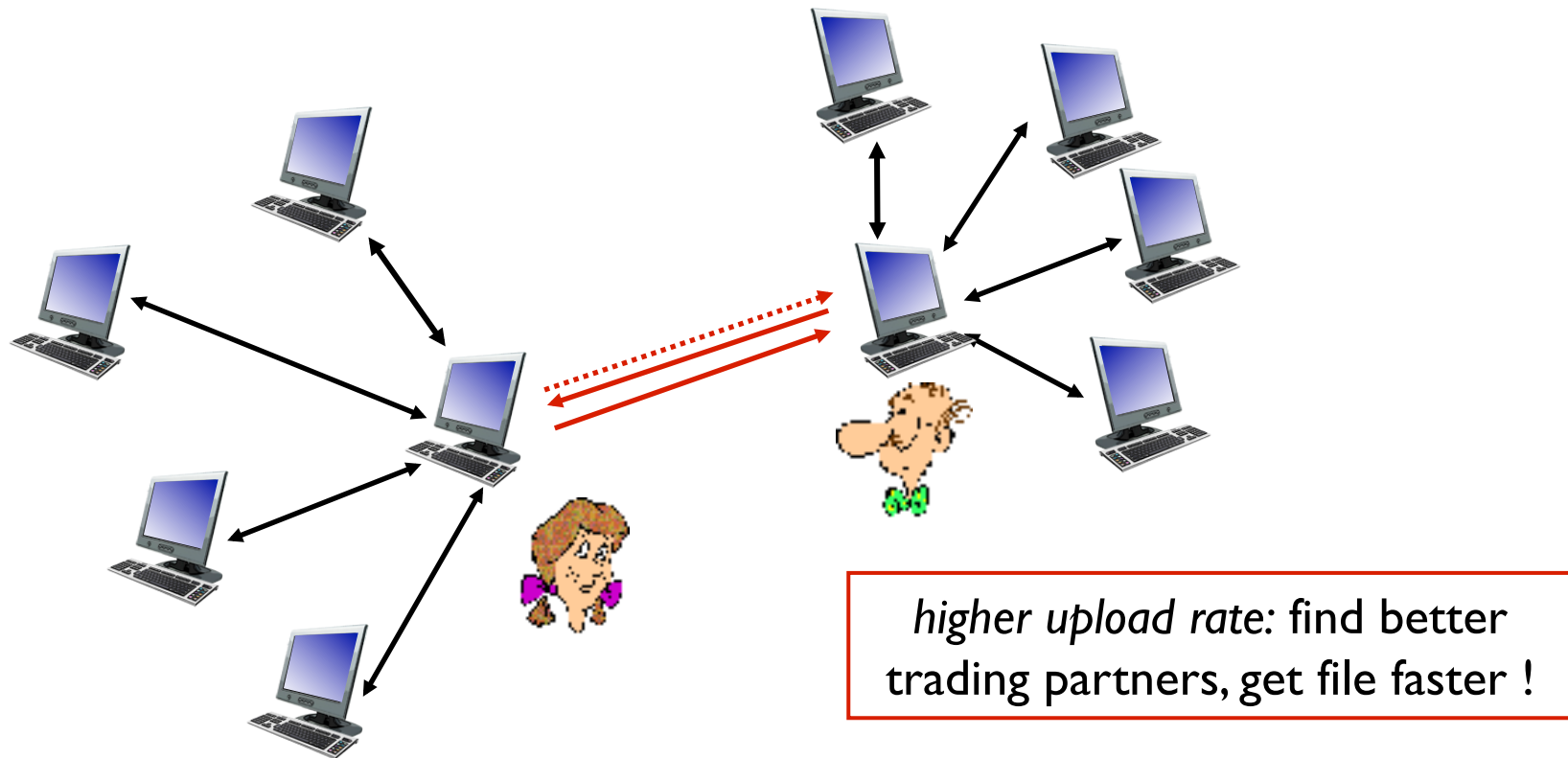
- ❖ at any given time, different peers have different subsets of file chunks
- ❖ periodically, Alice asks each peer for list of chunks that they have
- ❖ Alice requests missing chunks from peers, rarest first
- ❖ **Q:** Why rarest first?

## *sending chunks: tit-for-tat*

- ❖ Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- ❖ every 30 secs: randomly select another peer, starts sending chunks
  - “optimistically unchoke” this peer
  - newly chosen peer may join top 4

# BitTorrent: tit-for-tat

- (1) Alice “optimistically unchokes” Bob
- (2) Alice becomes one of Bob’s top-four providers; Bob reciprocates
- (3) Bob becomes one of Alice’s top-four providers



## Quiz: Free-riding



- ❖ Suppose Todd joins a BitTorrent torrent, but he does not want to upload any data to any other peers. Todd claims that he can receive a complete copy of the file that is shared by the swarm. Is Todd's claim possible? Why or Why not?

# Getting rid of the server/tracker

- ❖ Distribute the tracker information using a Distributed Hash Table (DHT)
- ❖ A DHT is a lookup structure
  - Maps keys to an arbitrary value
  - Works a lot like, well .... hash table

# Hash table - review

- ❖ (key,value) pairs
- ❖ Centralised hash table – all (key,value) pairs on 1 node
- ❖ Distributed hash tables – each node has a “section” of (key,value) pairs

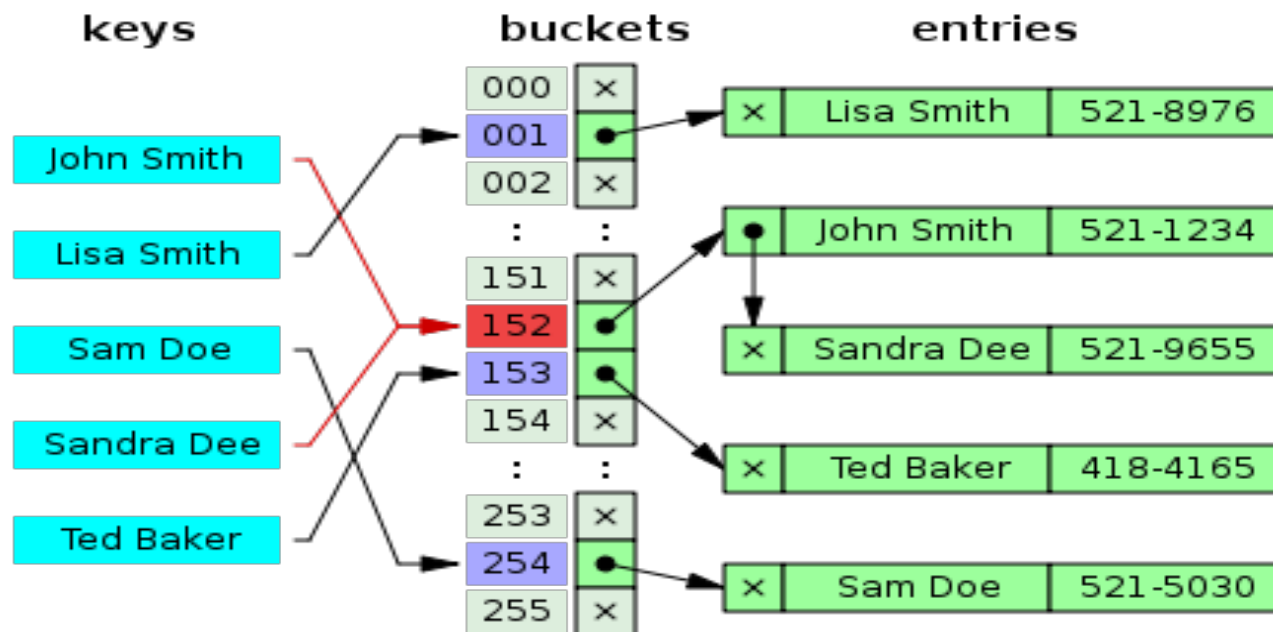


Figure src: [http://en.wikipedia.org/wiki/Hash\\_table](http://en.wikipedia.org/wiki/Hash_table)

# Distributed Hash Table (DHT)

- ❖ DHT: a *distributed P2P database*
- ❖ database has (key, value) pairs; examples:
  - key: TFN number; value: human name
  - key: file name; value: BT tracker peer(s)
- ❖ Distribute the (key, value) pairs over the (millions of peers)
- ❖ a peer **queries** DHT with key
  - DHT returns values that match the key
- ❖ peers can also **insert** (key, value) pairs

# Challenges

- ❖ How do we assign (key, value) pairs to nodes?
- ❖ How do we find them again quickly?
- ❖ What happens if nodes join/leave?



# Q: how to assign keys to peers?

## ❖ basic idea:

- convert each key to an integer
- Assign integer to each peer
- put (key,value) pair in the peer that is **closest** to the key

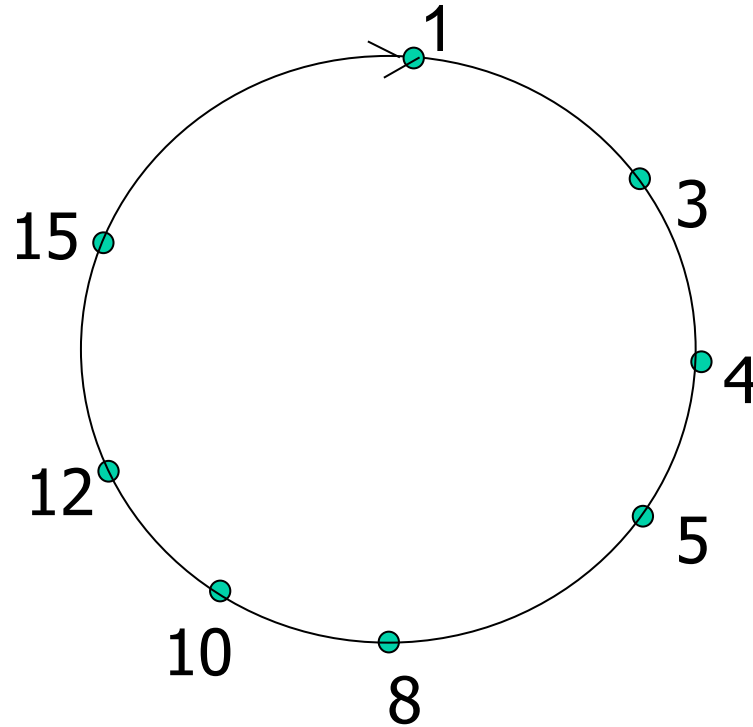
# DHT identifiers: Consistent Hashing

- ❖ assign integer identifier to each peer in range  $[0, 2^n - 1]$  for some  $n$ -bit hash function
  - E.g., node ID is hash of its IP address
- ❖ require each key to be an integer in same range
- ❖ to get integer key, hash original key
  - e.g., key = **hash**("House of Cards Season 4")
  - this is why its is referred to as a *distributed "hash" table*

# Assign keys to peers

- ❖ rule: assign key to the peer that has the *closest* ID.
- ❖ common convention: closest is the *immediate successor* of the key.
- ❖ e.g.,  $n=4$ ; peers: 1,3,4,5,8,10,12,14;
  - key = 13, then successor peer = 14
  - key = 15, then successor peer = 1

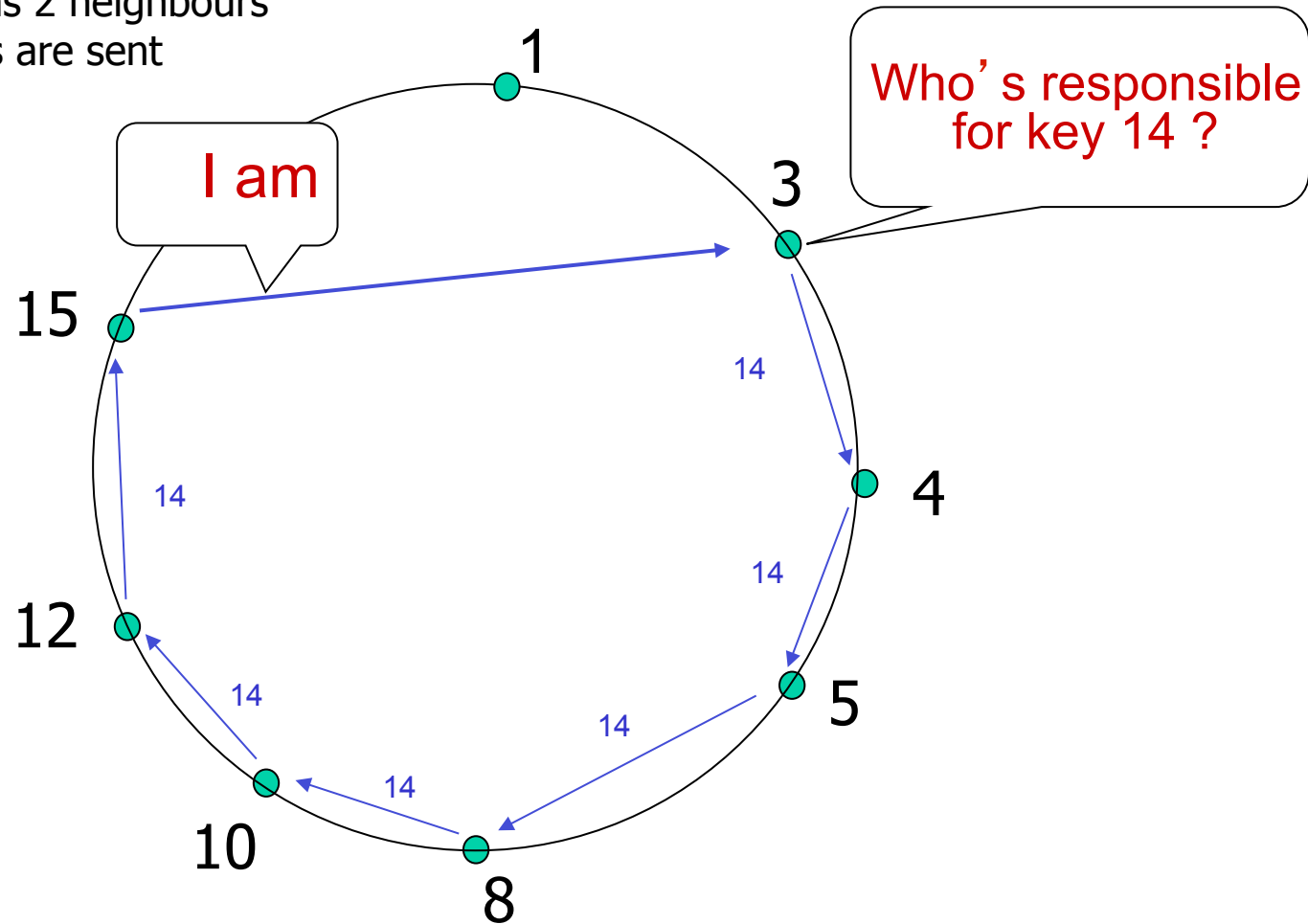
# Circular DHT (I)



- ❖ each peer *only* aware of immediate successor and predecessor.
- ❖ “overlay network”

# Circular DHT (2)

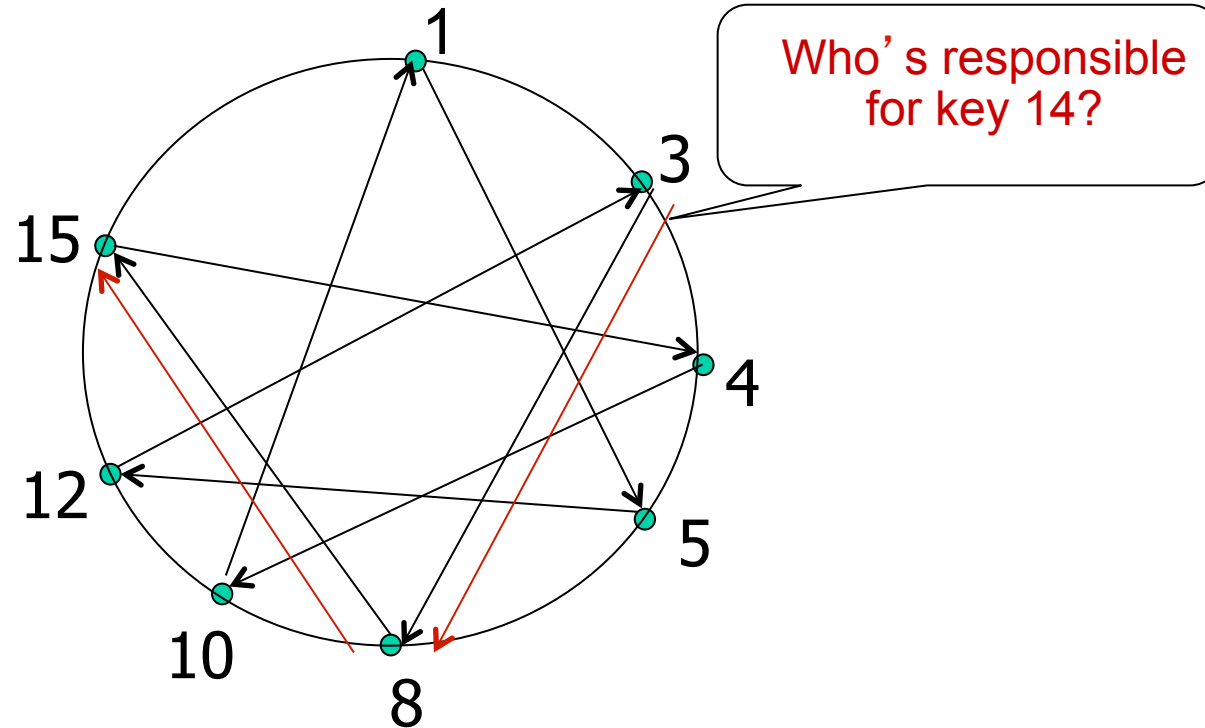
- Each peer maintains 2 neighbours
- 6 query messages are sent



# Tradeoff: #of neighbours vs. #of messages

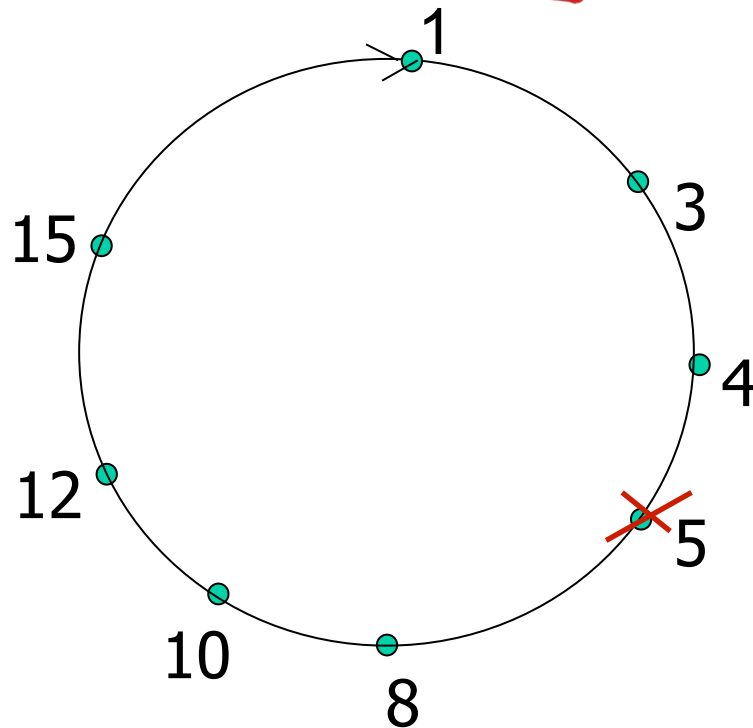
- The smaller the number of neighbours to maintain, the larger the number of messages have to be sent;
- Previous example: 2 neighbours, 6 messages
- Worst case:  $N$  messages
- Average:  $N/2$  messages

# Circular DHT with shortcuts



- ❖ each peer keeps track of IP addresses of predecessor, successor, short cuts
- ❖ reduced from 6 to 2 messages.
- ❖ possible to design shortcuts so  $O(\log N)$  neighbours,  $O(\log N)$  messages in query

# Peer churn



## handling peer churn:

- ❖ peers may come and go (churn)
- ❖ each peer knows address of its two successors
- ❖ each peer periodically pings its two successors to check aliveness
- ❖ if immediate successor leaves, choose next successor as new immediate successor

### *example: peer 5 abruptly leaves*

- ❖ peer 4 detects peer 5 departure; makes 8 its immediate successor; asks 8 who its immediate successor is; makes 8's immediate successor its second successor.



# More DHT info

---

- ❖ How do nodes join?
- ❖ How does cryptographic hashing work?
- ❖ How much state does each node store?

Research Articles (available from Moodle page):  
Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications  
Dynamo: Amazon's Highly Available Key-value Store

# DHT: Applications

- ❖ File sharing and backup [CFS, Ivy, OceanStore, PAST, Pastiche ...]
- ❖ Web cache and replica [Squirrel, Croquet Media Player]
- ❖ Censor-resistant stores [Eternity]
- ❖ DB query and indexing [PIER, Place Lab, VPN Index]
- ❖ Event notification [Scribe]
- ❖ Naming systems [ChordDNS, Twine, INS, HIP]
- ❖ Distributed BitTorrent tracker [Kademlia, Vuze]
- ❖ Communication primitives [I3, ...]
- ❖ Host mobility [DTN Tetherless Architecture]

# Transport Layer

## our goals:

- ❖ understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- ❖ learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

# Transport Layer Outline

## 3.1 transport-layer services

## 3.2 multiplexing and demultiplexing

## 3.3 connectionless transport: UDP

## 3.4 principles of reliable data transfer

## 3.5 connection-oriented transport: TCP

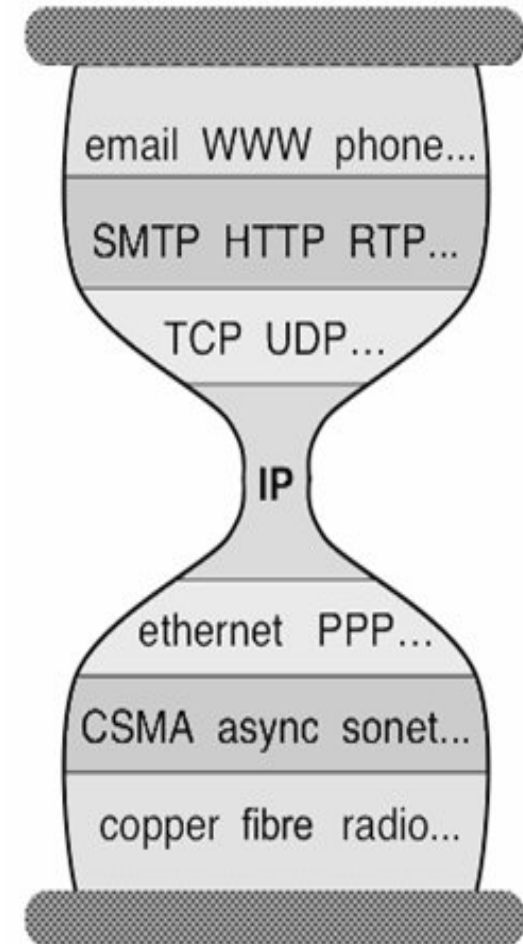
- segment structure
- reliable data transfer
- flow control
- connection management

## 3.6 principles of congestion control

## 3.7 TCP congestion control

# Transport layer

- ❖ Moving “down” a layer
- ❖ Current perspective:
  - Application is the boss....
  - Usually executing within the OS Kernel
  - The network layer is ours to command !!

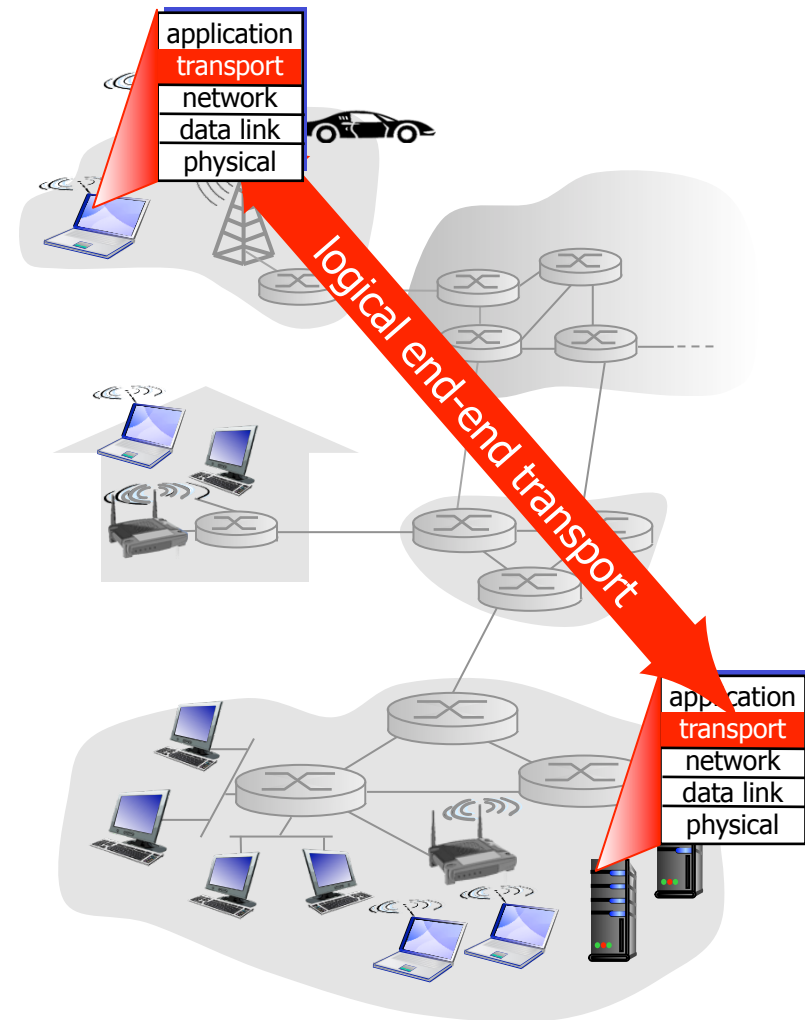


# Network layer (context)

- ❖ What it does: finds paths through network
  - Routing from one end host to another
- ❖ What it doesn't:
  - Reliable transfer: “best effort delivery”
  - Guarantee paths
  - Arbitrate transfer rates
- ❖ For now, think of the network layer as giving us an “API” with one function:  
*sendtohost(data, host)*
  - Promise: the data will go to that (usually!!)

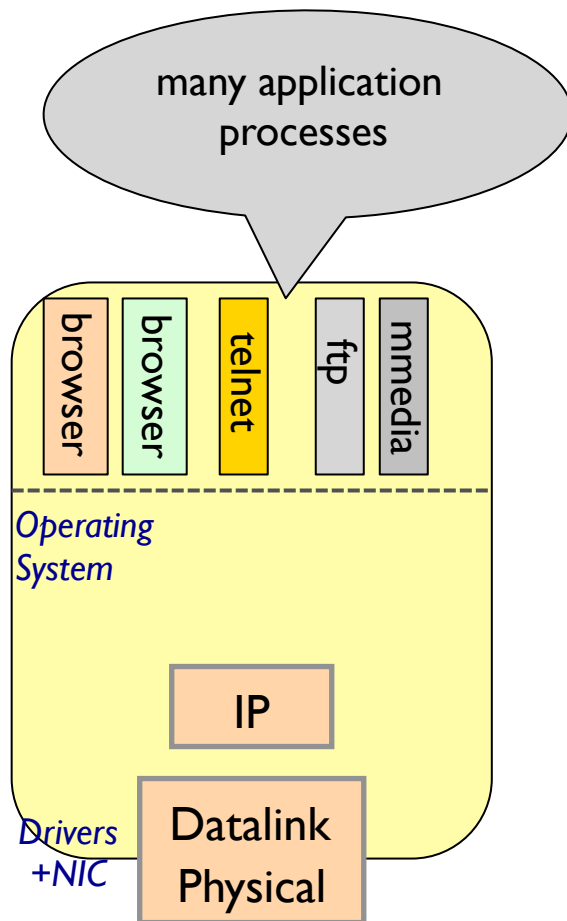
# Transport services and protocols

- ❖ 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
  - Exports services to application that network layer does not provide

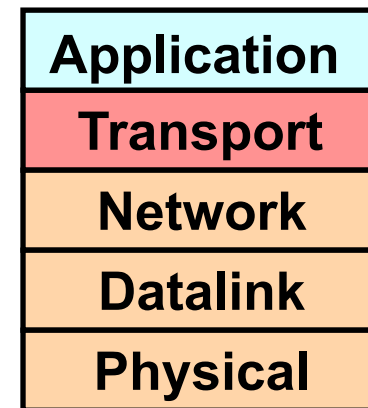


Network layer provides logical communication between hosts

# Why a transport layer?



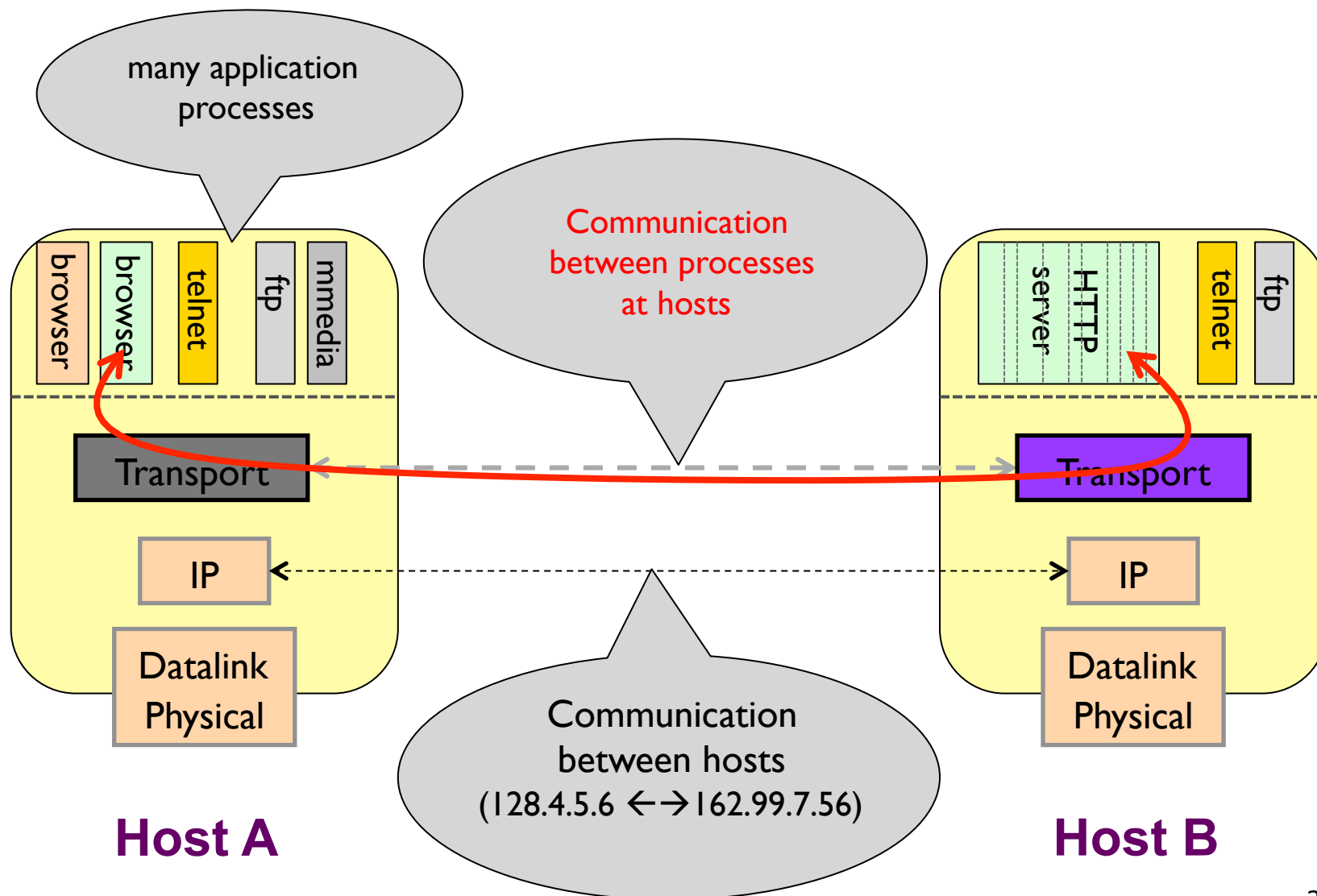
**Host A**



**Host B**



# Why a transport layer?



# Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

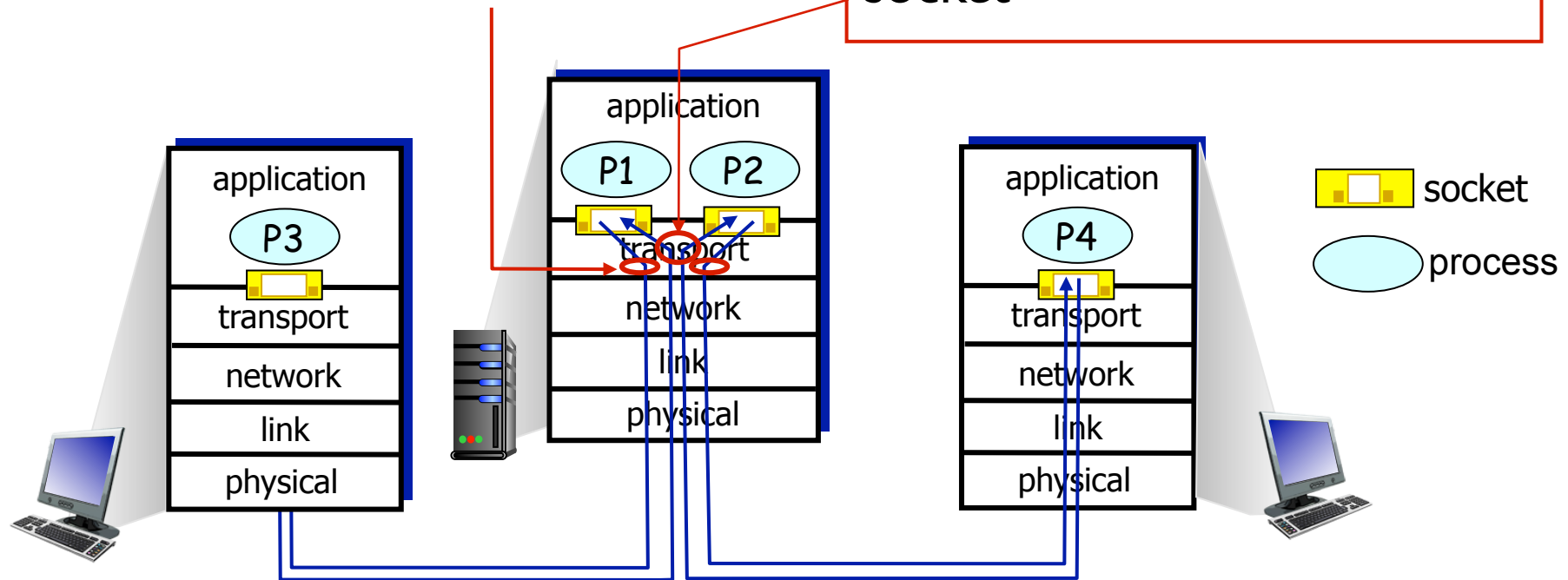
3.6 principles of congestion control

3.7 TCP congestion control

# Multiplexing/demultiplexing

*multiplexing at sender:*  
handle data from multiple sockets, add transport header (later used for demultiplexing)

*demultiplexing at receiver:*  
use header info to deliver received segments to correct socket



**Note:** The network is a shared resource. It does not care about your applications, sockets, etc.

# Connectionless demultiplexing

- ❖ *recall*: created socket has host-local port #:

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

- ❖ *recall*: when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- ❖ when host receives UDP segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #



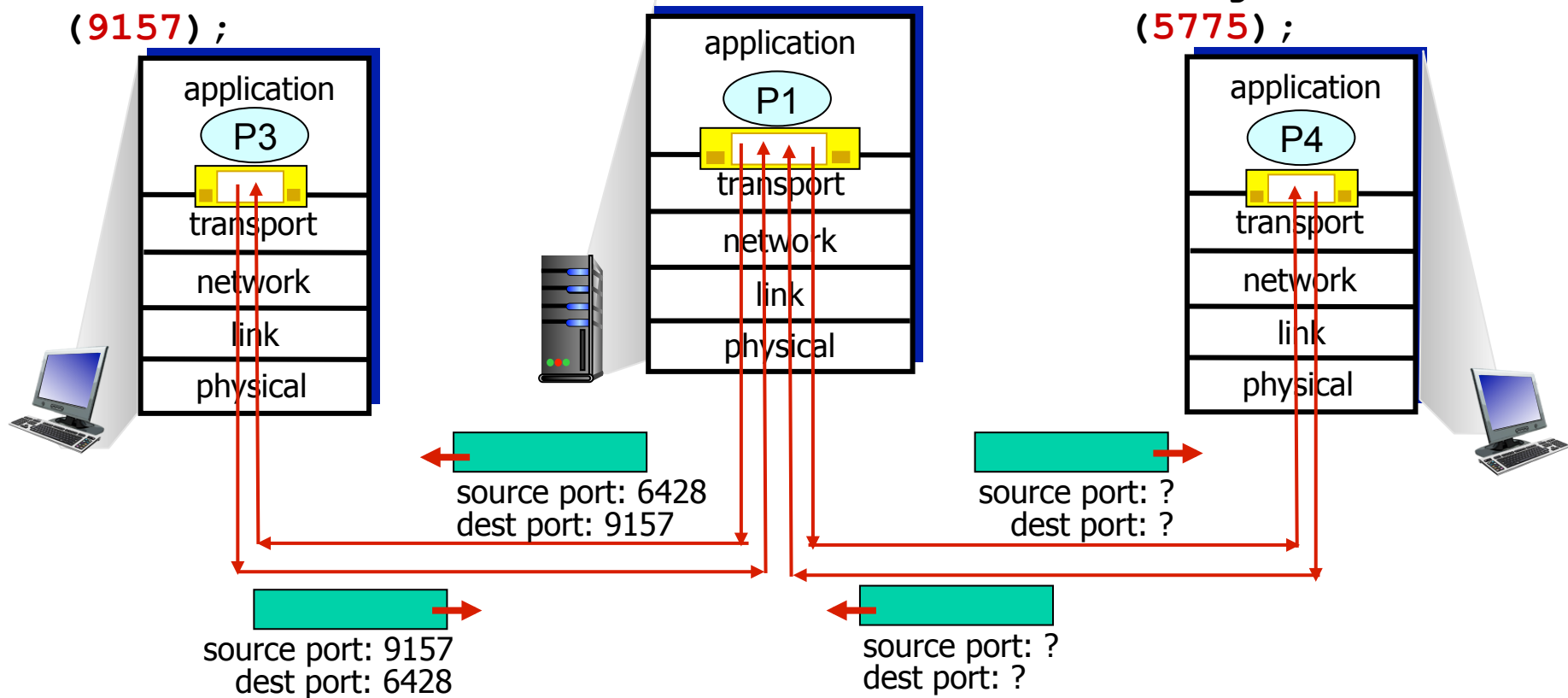
IP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

# Connectionless demux: example

```
DatagramSocket  
mySocket2 = new  
DatagramSocket  
(9157);
```

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

```
DatagramSocket  
mySocket1 = new  
DatagramSocket  
(5775);
```

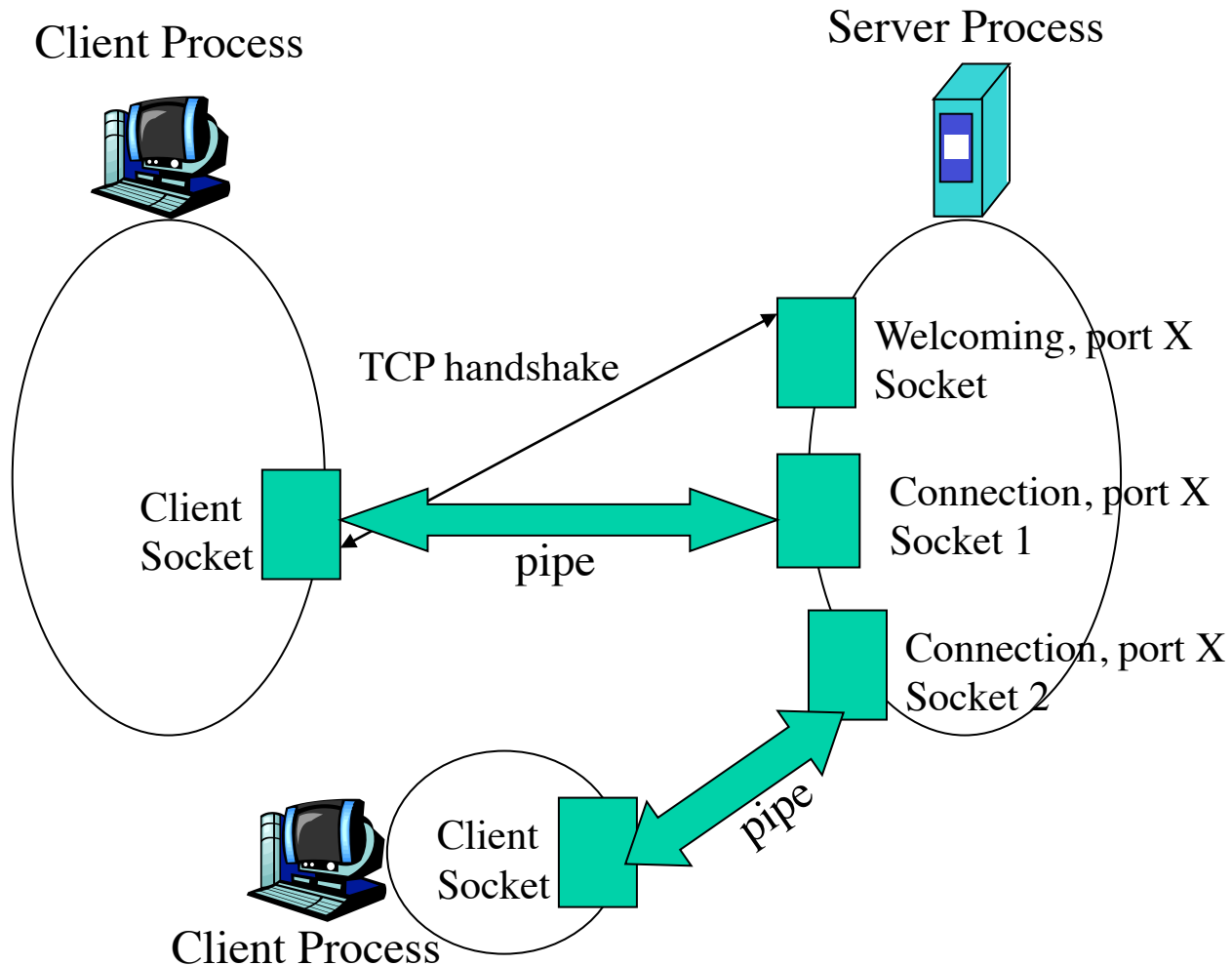


# Connection-oriented demux

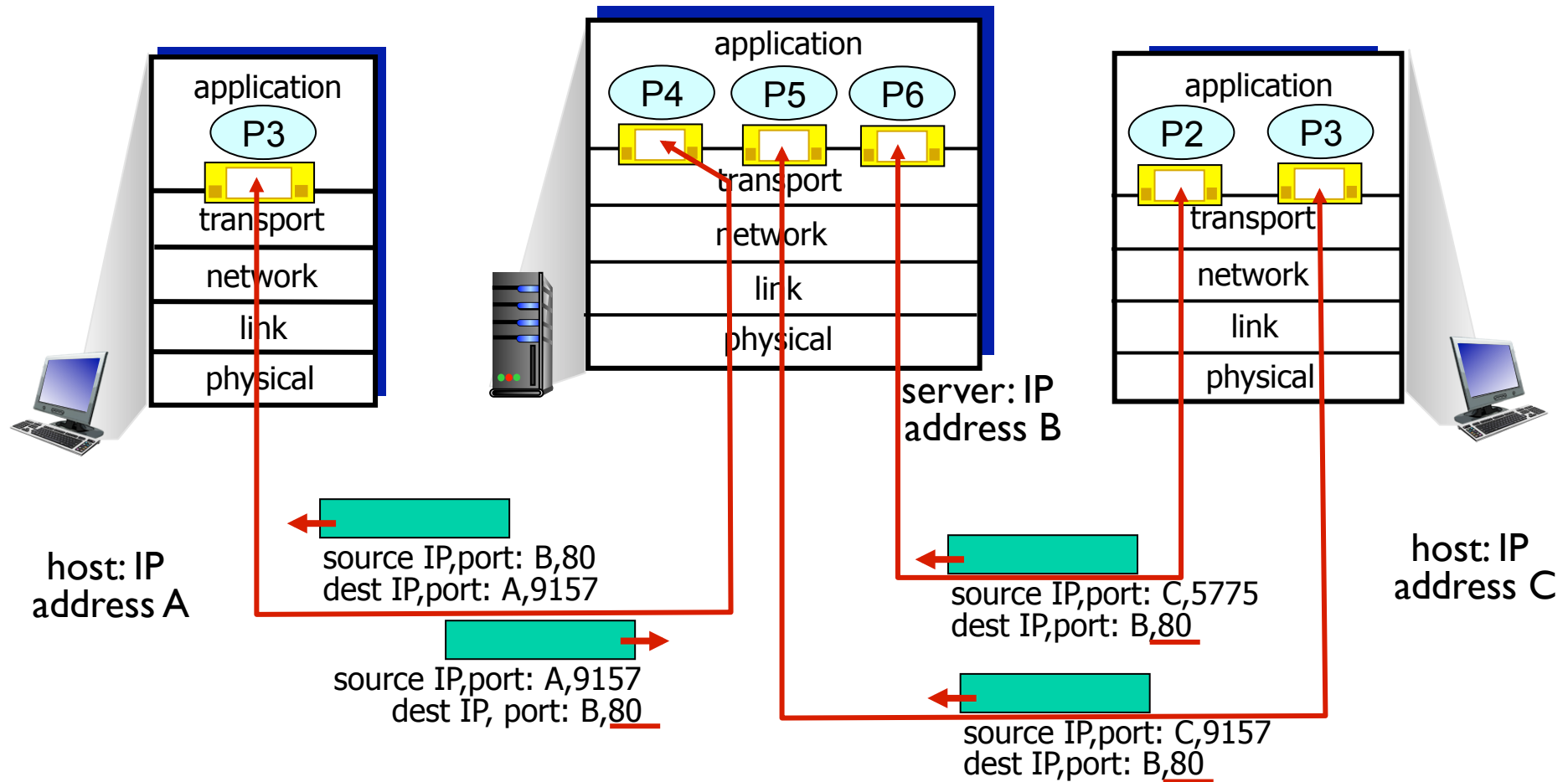
- ❖ TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- ❖ demux: receiver uses all four values to direct segment to appropriate socket
- ❖ server host may support many simultaneous TCP sockets:
  - each socket identified by its own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

# Revisiting TCP Sockets

Server process creates (spawns) two more sockets for two different TCP connections



# Connection-oriented demux: example



three segments, all destined to IP address: B,  
dest port: 80 are demultiplexed to *different* sockets



# Take-away message

- ❖ UDP is identified by 2-tuple
  - Server maintains a single socket for all incoming UDP packets
- ❖ TCP is identified by 4-tuple
  - Server creates new sockets for each TCP connection

# May I scan your ports?

<http://netsecurity.about.com/cs/hackertools/a/aa121303.htm>

- ❖ Servers wait at open ports for client requests
- ❖ Hackers often perform *port scans* to determine open, closed and unreachable ports on candidate victims
- ❖ Several ports are well-known
  - <1024 are reserved for well-known apps
  - Other apps also use known ports
    - MS SQL server uses port 1434 (udp)
    - Sun Network File System (NFS) 2049 (tcp/udp)
- ❖ Hackers can use exploit known flaws with these known apps
  - Example: Slammer worm exploited buffer overflow flaw in the SQL server
- ❖ How do you scan ports?
  - Nmap, Superscan, etc

<http://www.auditmypc.com/>

<https://www.grc.com/shieldsup>

# Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

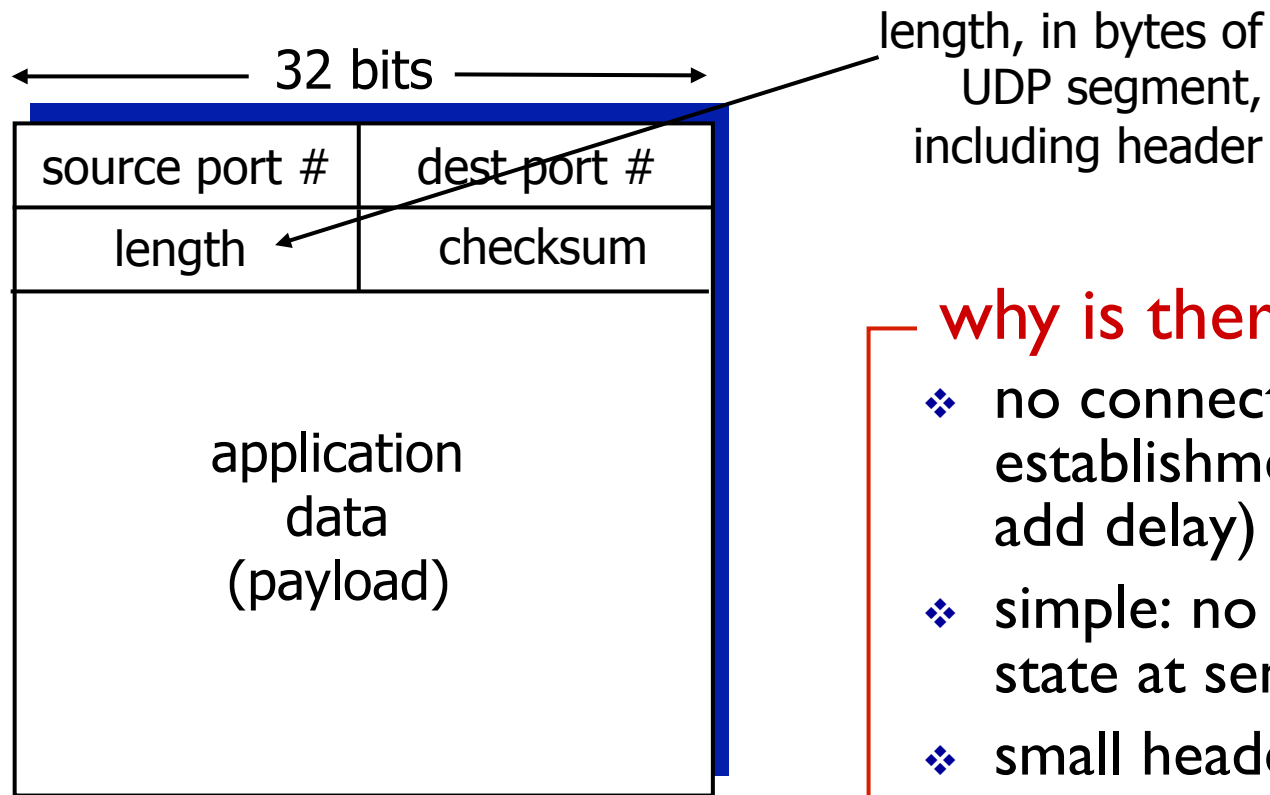
3.6 principles of congestion control

3.7 TCP congestion control

# UDP: User Datagram Protocol [RFC 768]

- ❖ “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

# UDP: segment header



UDP segment format

## why is there a UDP?

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

# UDP checksum

- **Goal:** detect “errors” (e.g., flipped bits) in transmitted segment
  - Router memory errors
  - Driver bugs
  - Electromagnetic interference

## sender:

- ❖ treat segment contents, including header fields, as sequence of 16-bit integers
- ❖ checksum: addition (one's complement sum) of segment contents
- ❖ sender puts checksum value into UDP checksum field

## receiver:

- ❖ Add all the received together as 16-bit integers
- ❖ Add that to the checksum
- ❖ If the result is not 1111 1111 1111, there are errors !

# Internet checksum: example

example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

*Note:* when adding numbers, a carryout from the most significant bit needs to be added to the result

# UDP Applications

- ❖ Latency sensitive/time critical
  - ❖ Quick request/response (DNS, DHCP)
  - ❖ Network management (SNMP)
  - ❖ Routing updates (RIP)
  - ❖ Voice/video chat
  - ❖ Gaming (especially FPS)
- ❖ Error correction unnecessary (periodic messages)



# Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

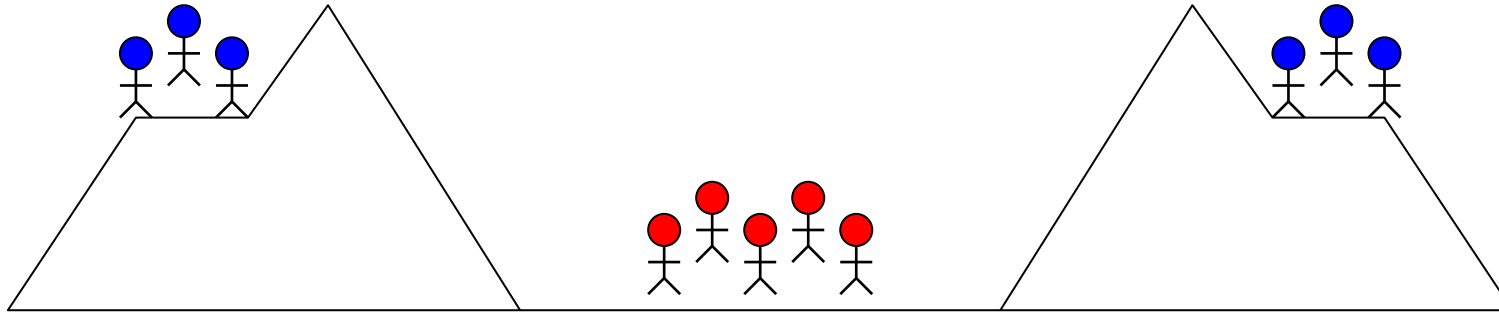
3.6 principles of congestion control

3.7 TCP congestion control

# Reliable Transport

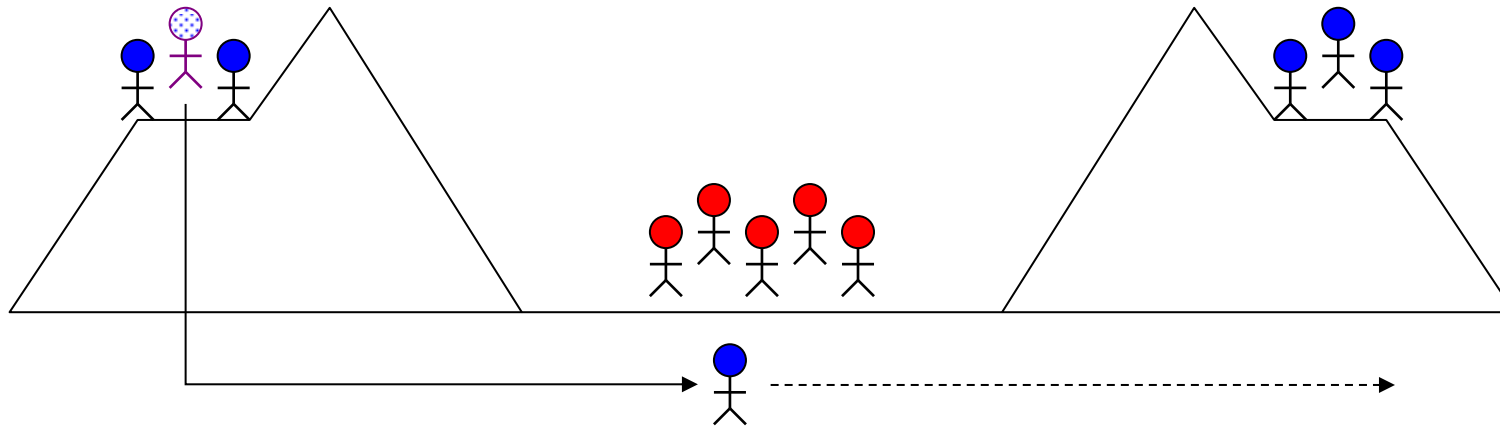
- In a perfect world, reliable transport is easy
- All the bad things best-effort can do
  - a packet is corrupted (bit errors)
  - a packet is lost
  - a packet is delayed (*why?*)
  - packets are reordered (*why?*)
  - a packet is duplicated (*why?*)

# The Two Generals Problem



- ❖ Two army divisions (blue) surround enemy (red)
  - Each division led by a general
  - Both must agree when to simultaneously attack
  - If either side attacks alone, defeat
- ❖ Generals can only communicate via messengers
  - Messengers may get captured (unreliable channel)

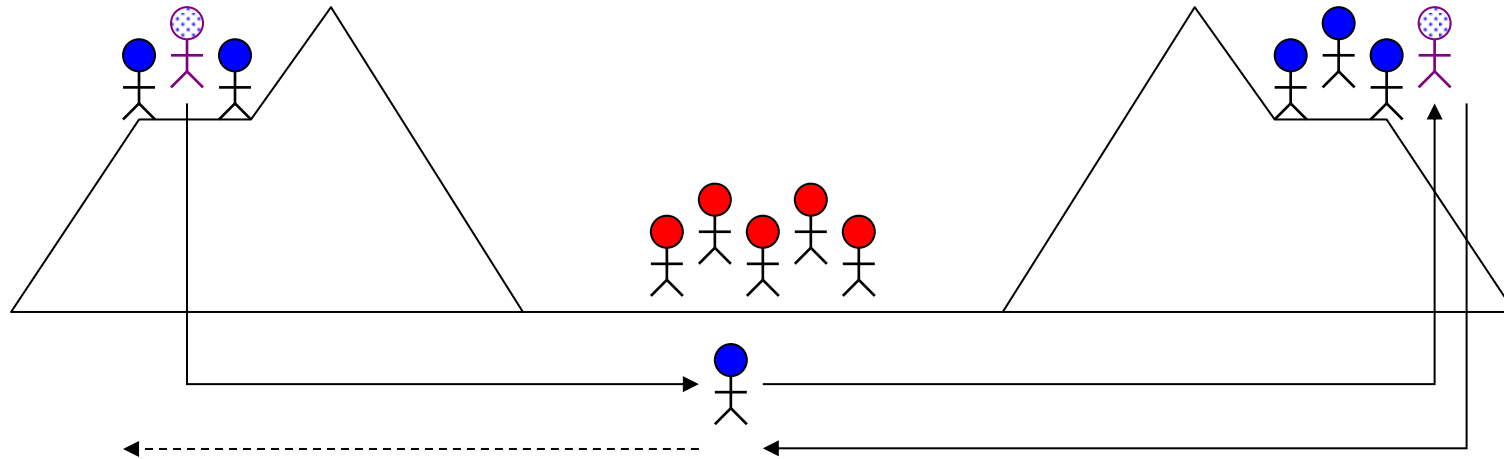
# The Two Generals Problem



## ❖ How to coordinate?

- Send messenger: “Attack at dawn”
- What if messenger doesn’t make it?

# The Two Generals Problem



- ❖ How to be sure messenger made it?
  - Send acknowledgement: “We received message”

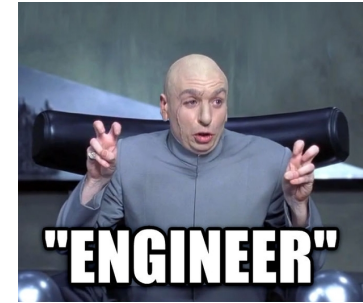
# Engineering

## ❖ Concerns

- Message corruption
- Message duplication
- Message loss
- Message reordering
- Performance

## ❖ Our toolbox

- Checksums
- Timeouts
- Acks and Nacks
- Sequence numbering
- Pipelining

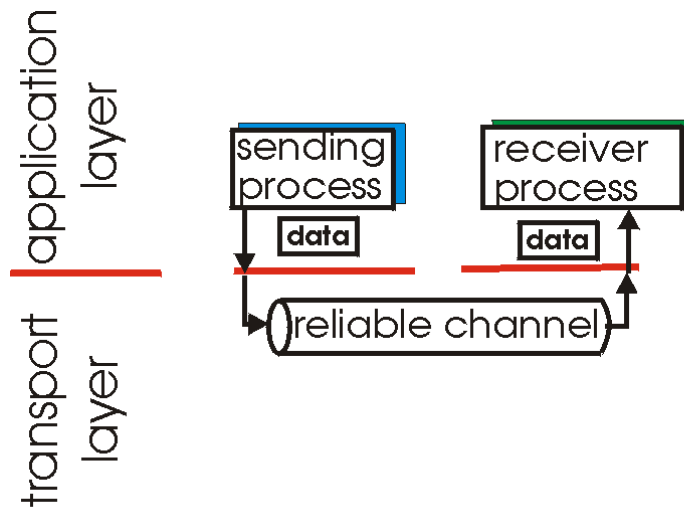


We will use these to build **Automatic Repeat Request (ARQ)** protocols

- Stop-and-wait
- Pipelining
  - Go-back-N
  - Selective Repeat

# Principles of reliable data transfer

- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!

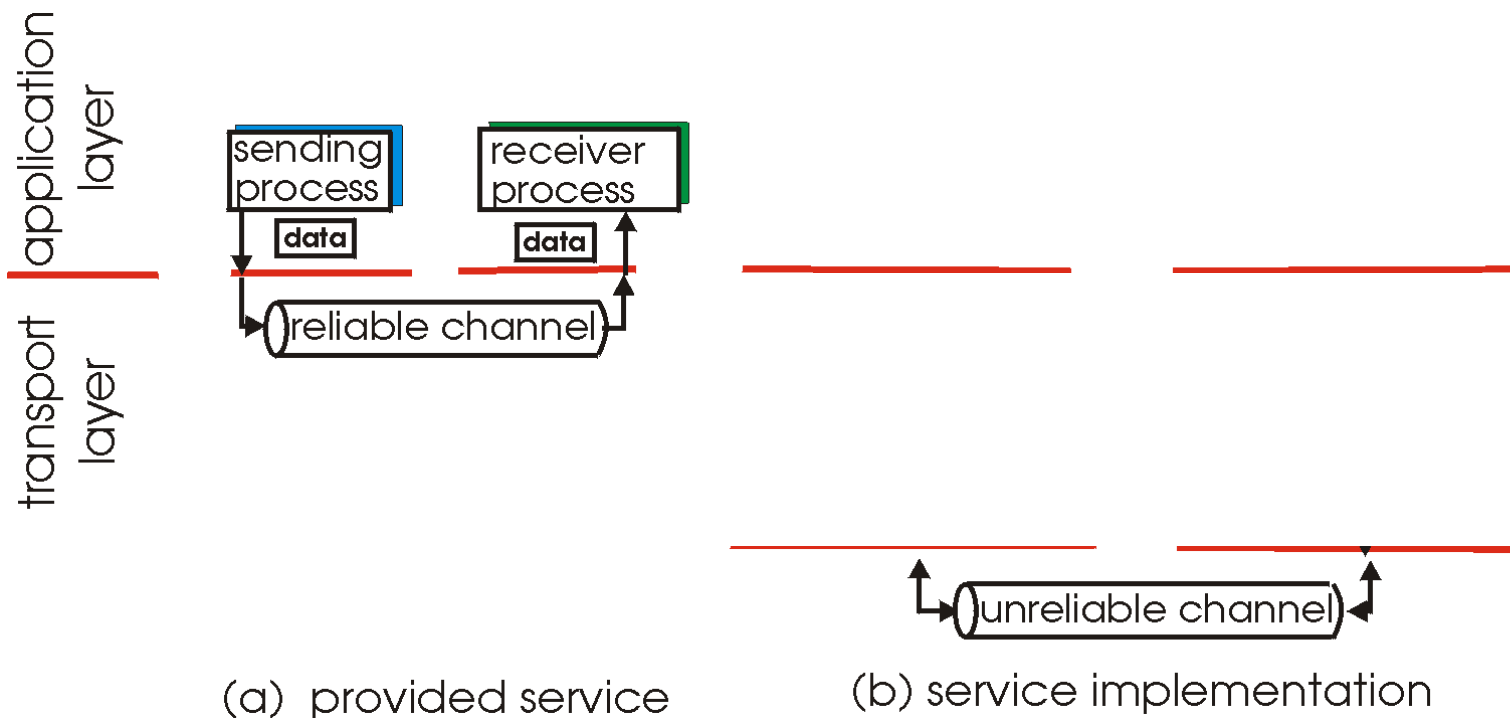


(a) provided service

- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Principles of reliable data transfer

- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!

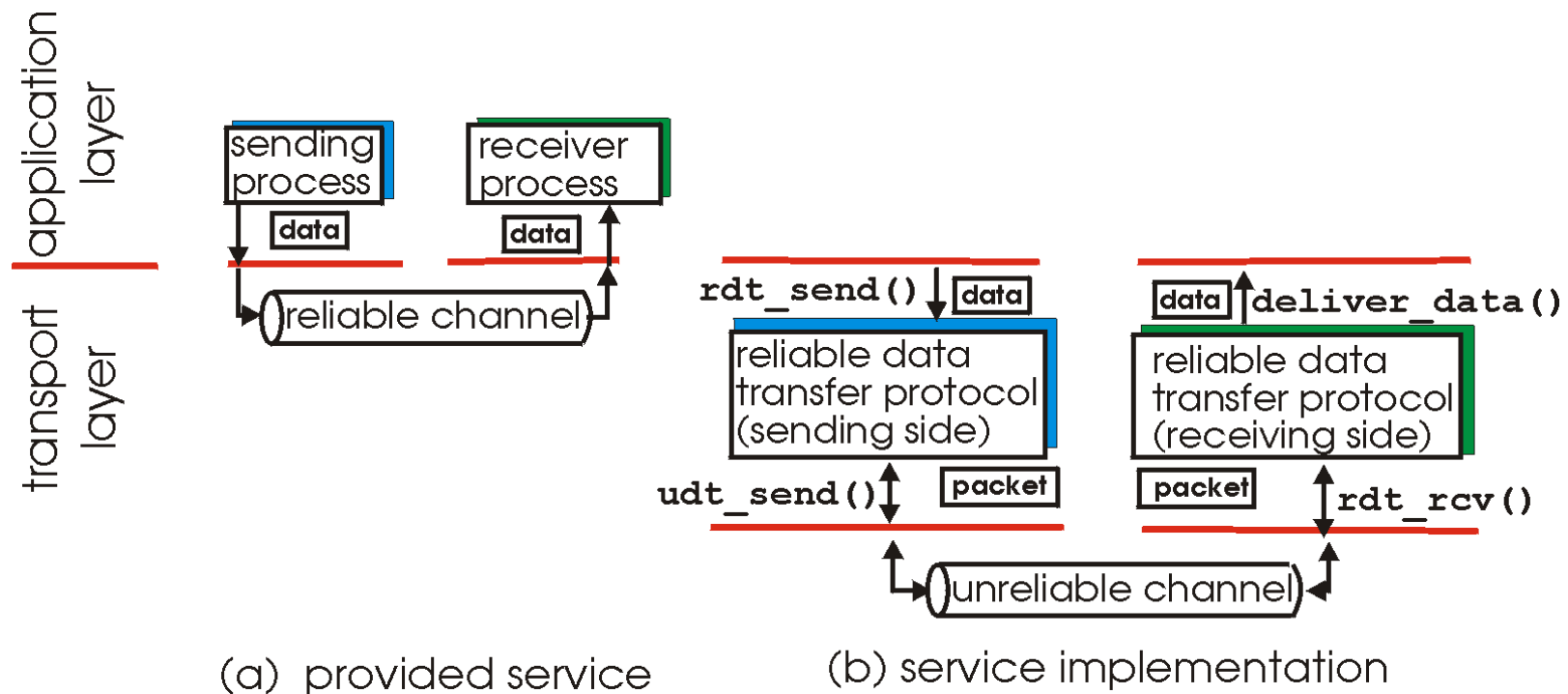


- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)



# Principles of reliable data transfer

- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!



- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

## rdt1.0: reliable transfer over a reliable channel

- ❖ underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Transport layer does not have to do anything !!

# rdt2.0: channel with bit errors

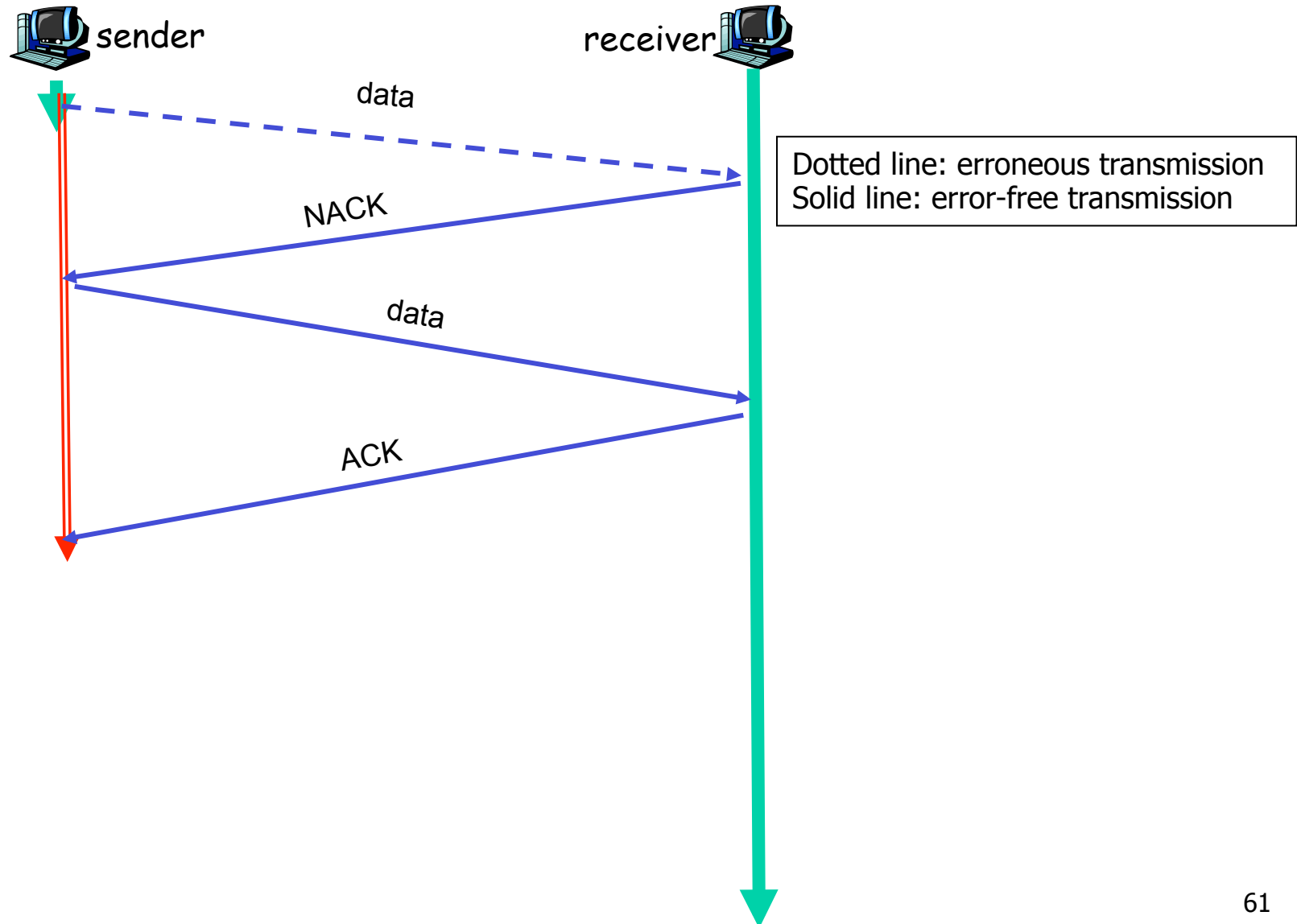
- ❖ underlying channel may flip bits in packet
  - checksum to detect bit errors
- ❖ *the question: how to recover from errors:*

*How do humans recover from “errors”  
during conversation?*

# rdt2.0: channel with bit errors

- ❖ underlying channel may flip bits in packet
  - checksum to detect bit errors
- ❖ *the question: how to recover from errors:*
  - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- ❖ new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender

# Global Picture of rdt2.0



# rdt2.0 has a fatal flaw!

## what happens if ACK/ NAK corrupted?

- ❖ sender doesn't know what happened at receiver!
- ❖ can't just retransmit: possible duplicate

## handling duplicates:

- ❖ sender retransmits current pkt if ACK/NAK corrupted
- ❖ sender adds *sequence number* to each pkt
- ❖ receiver discards (doesn't deliver up) duplicate pkt

## stop and wait

sender sends one packet,  
then waits for receiver  
response

# rdt2.1: discussion

## sender:

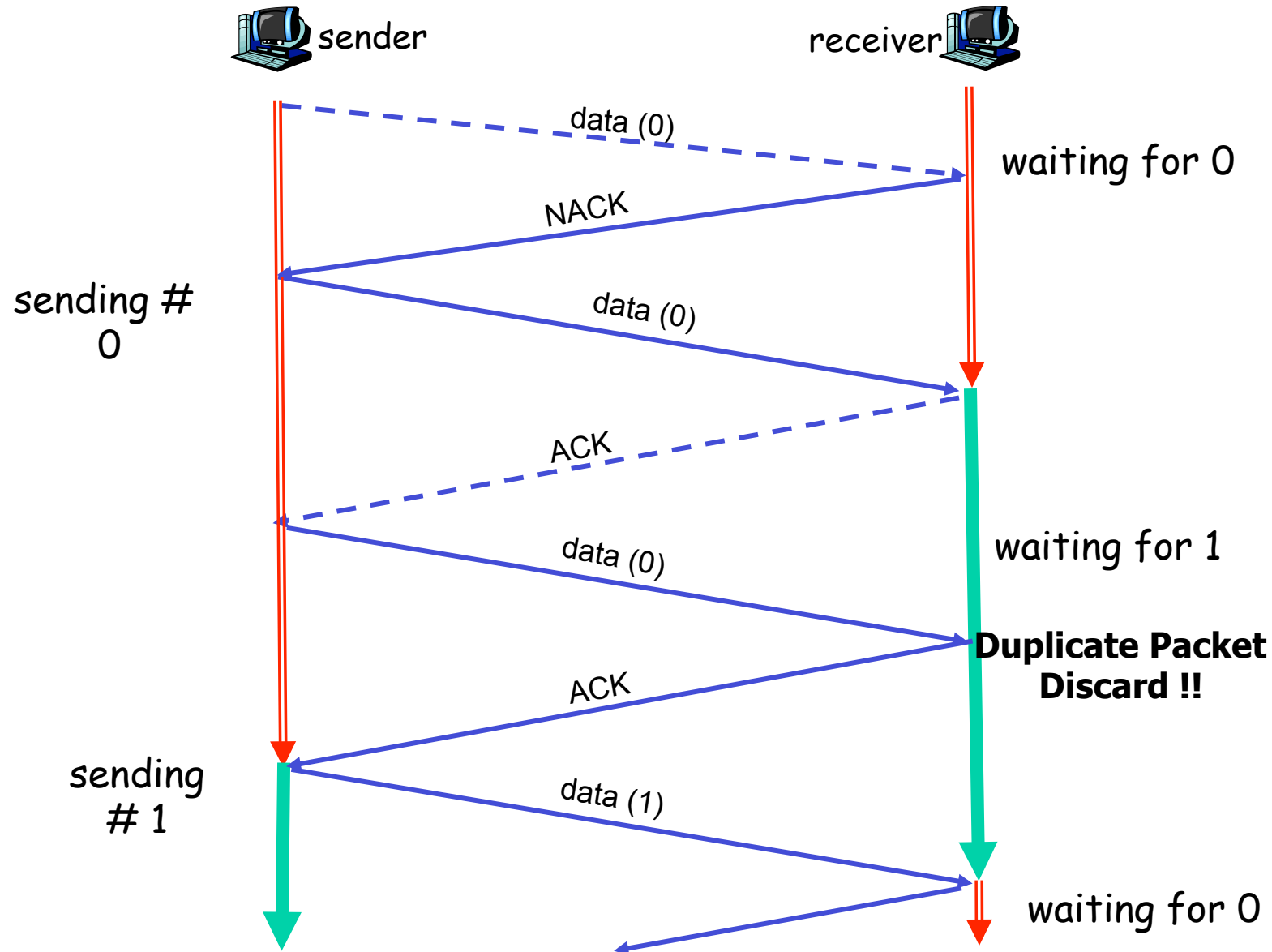
- ❖ seq # added to pkt
- ❖ two seq. #'s (0,1) will suffice. Why?
- ❖ must check if received ACK/NAK corrupted
- ❖ twice as many states
  - state must “remember” whether “expected” pkt should have seq # of 0 or 1

## receiver:

- ❖ must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- ❖ note: receiver can *not* know if its last ACK/NAK received OK at sender

# Another Look at rdt2.1

Dotted line: erroneous transmission  
Solid line: error-free transmission



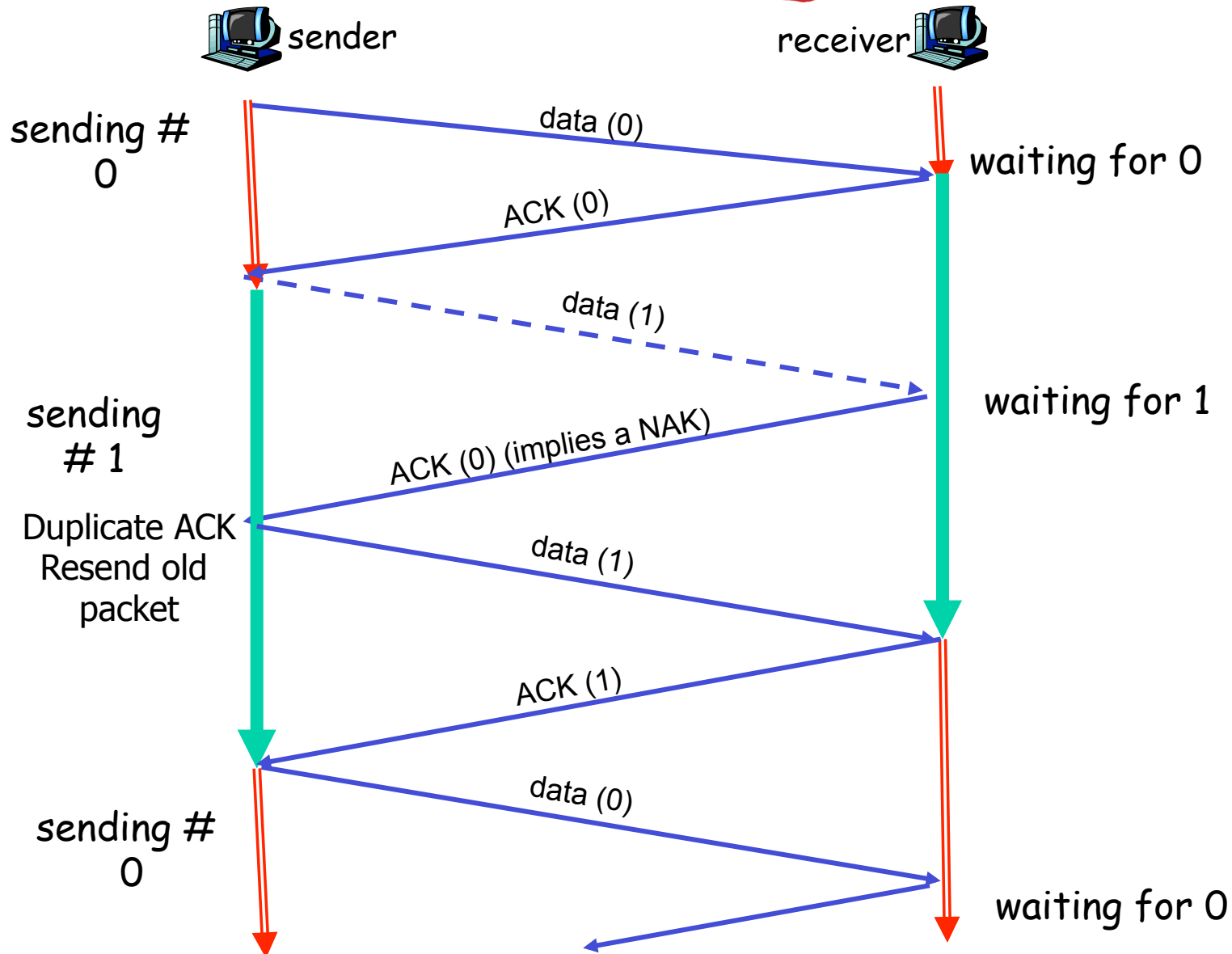


## rdt2.2: a NAK-free protocol

- ❖ same functionality as rdt2.1, using ACKs only
- ❖ instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- ❖ duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

# rdt2.2: Example

Dotted line: erroneous transmission  
Solid line: error-free transmission



## rdt3.0: channels with errors *and* loss

### new assumption:

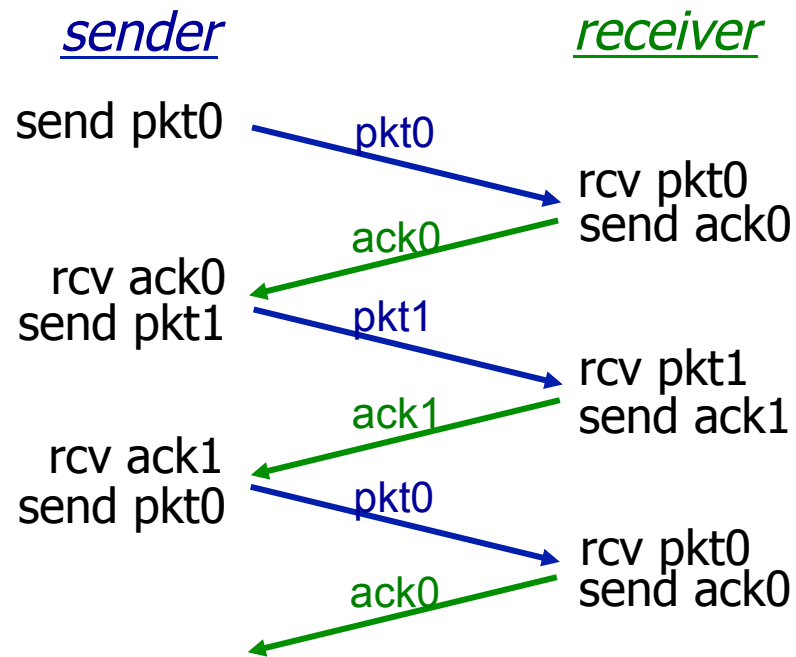
underlying channel can also lose packets (data, ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help ... but not enough

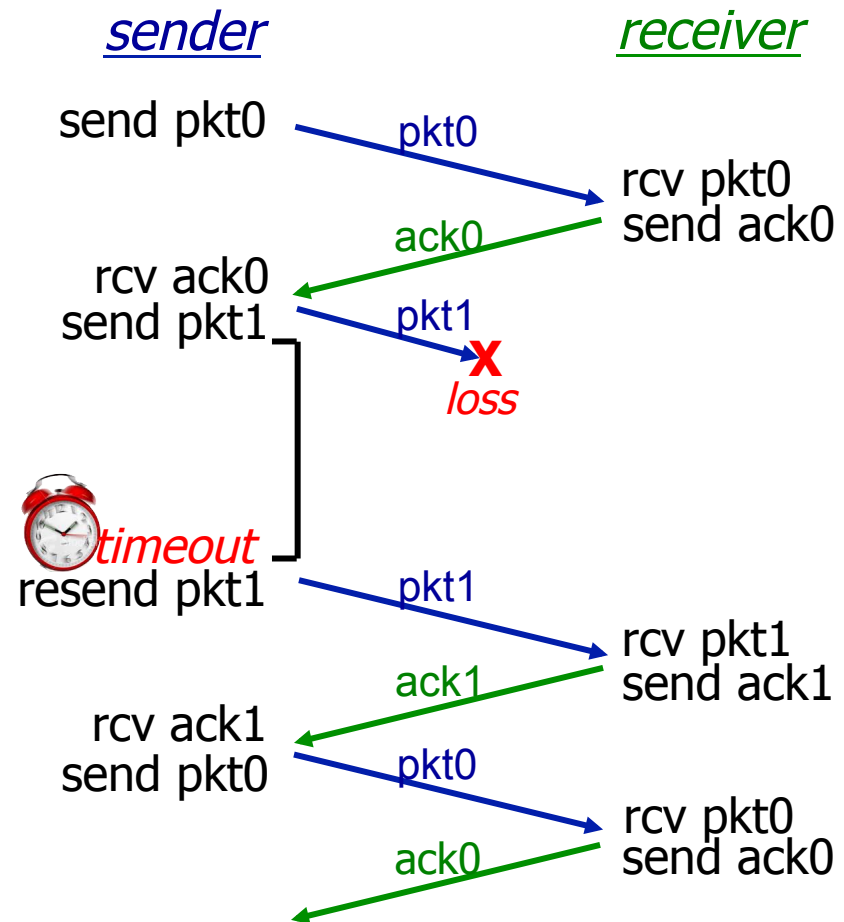
approach: sender waits “reasonable” amount of time for ACK

- ❖ retransmits if no ACK received in this time
- ❖ if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- ❖ requires countdown timer

# rdt3.0 in action

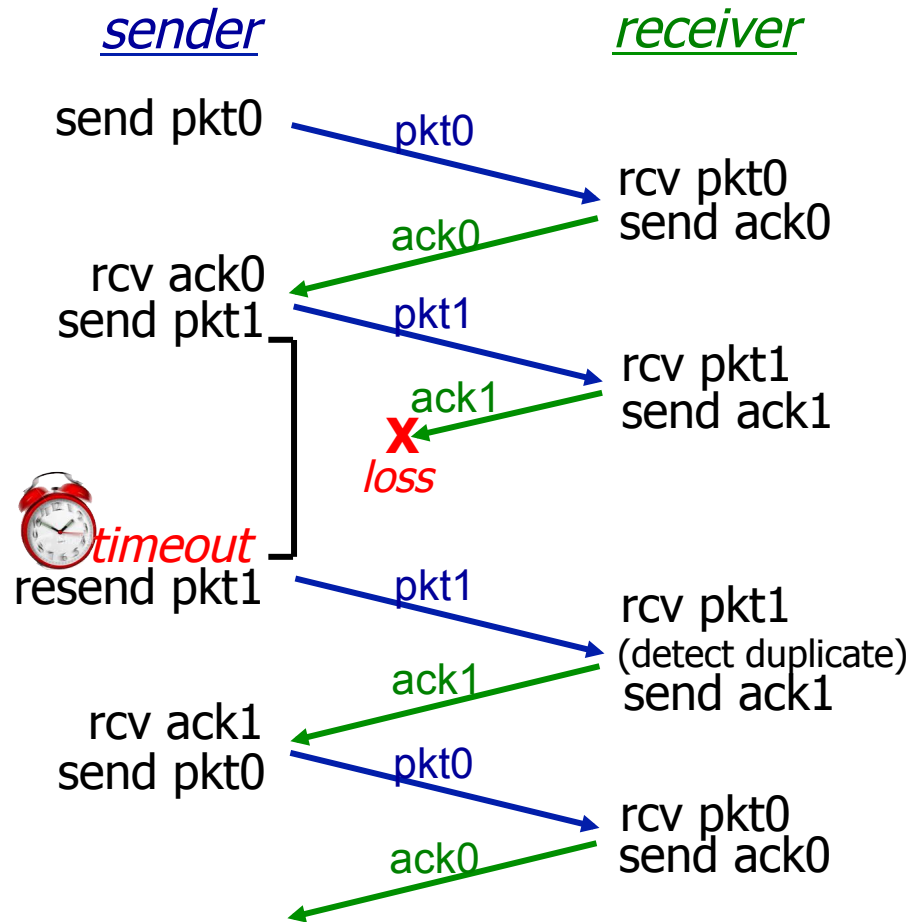


(a) no loss

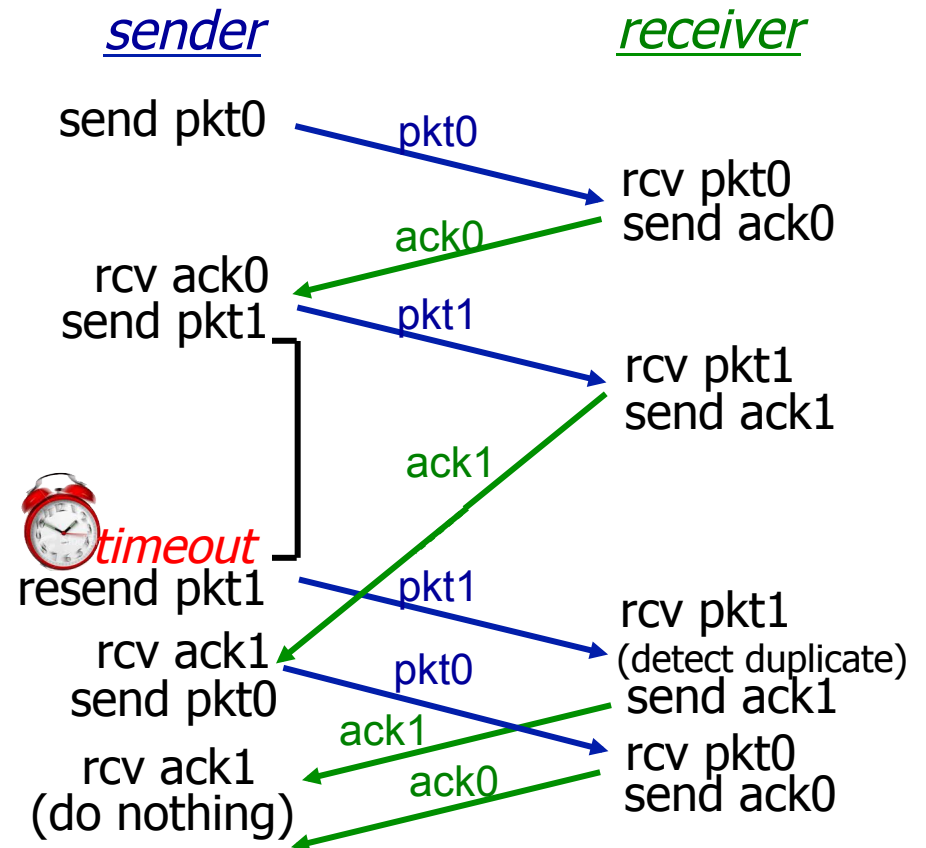


(b) packet loss

# rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

# Transport Part I: Summary

- ❖ principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer

- ❖ UDP

- ❖ **Next Week:**

- Pipelined Protocols for reliable data transfer
- TCP
  - TCP Flow Control
  - TCP Connection Management
  - TCP Congestion Control