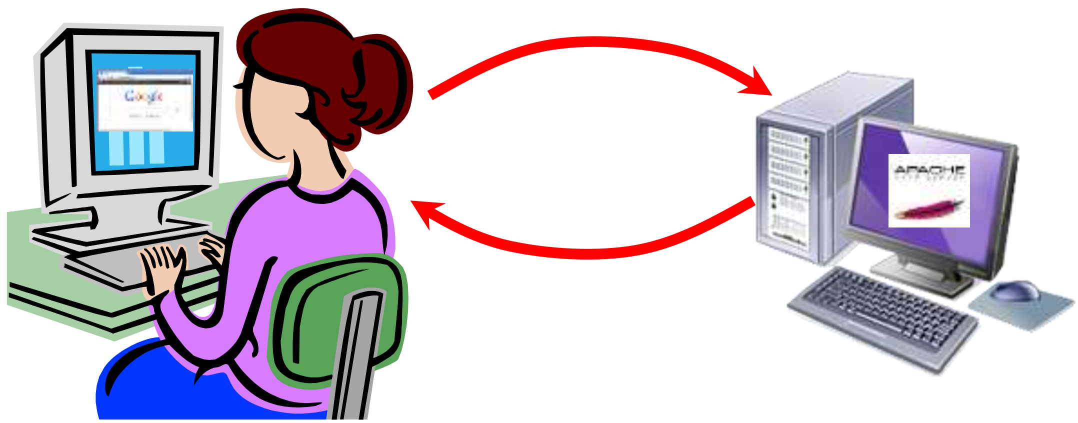
Network software & performance



latency

propagation | **transmit** | **queue**

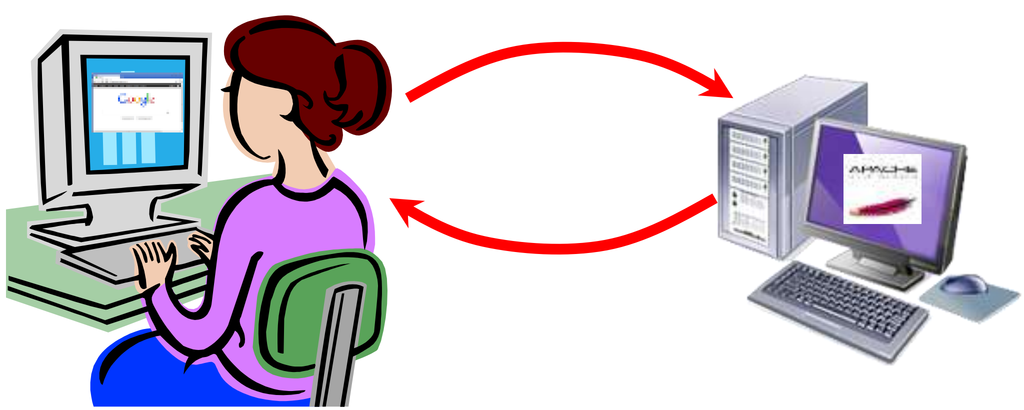
Overview

- How do we write network software?
  - Socket API
- How do we measure network performance?
  - Bandwidth
  - Propagation delay
  - What happens if bandwidth is ∞?
  - Effective throughput of a network

Clients and servers

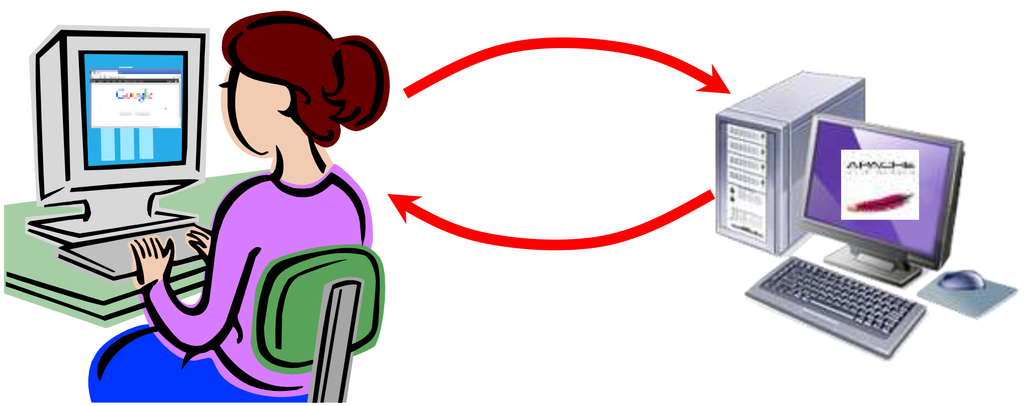
- Client program
  - Requests service
  - E.g. web browser, audio player, Twitter client

- Server program
  - Provides service
  - E.g. web server, audio server at streaming station, server at Twitter



Clients and servers

- Client program
  - "sometimes"
  - Doesn't talk to other clients
  - Needs to know server's address
- Server program
  - "always on"
  - Serves requests from many clients
  - Needs fixed address



Communication steps

- Network
  - Gets data to the destination host
  - Uses destination IP address
- Operating system
  - Forwards data to a given “silo” based on port #
  - E.g. All port 80 request go the web server
- Application
  - Actually reads and writes to socket
  - Implement the application specific magic

Port numbers

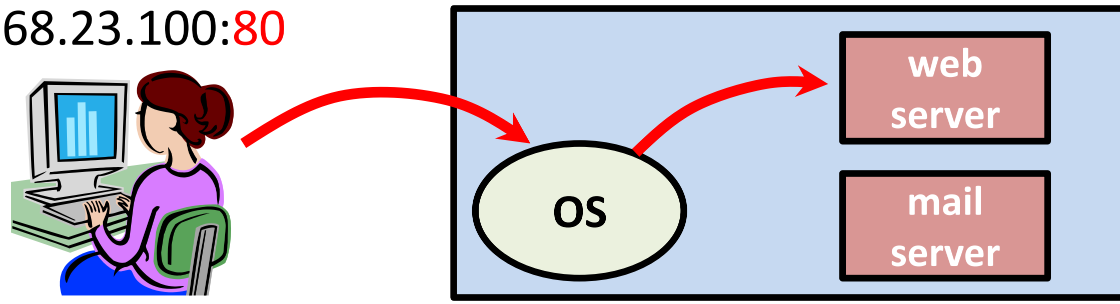
- Popular applications have known ports
  - Server uses a well-known port, 0 - 1023
  - Client uses a free temporary port, 1024 - 65535

Port	Service
21	File transfer protocol (FTP)
22	Secure shell (SSH)
23	Telenet
25	Simple mail transfer protocol (SMTP)
53	Domain name system (DNS)
80	Hypertext transfer protocol (HTTP)
110	Post office protocol (POP)
143	Internet message access protocol (IMAP)
443	HTTP secure (HTTPS)

Use of port number

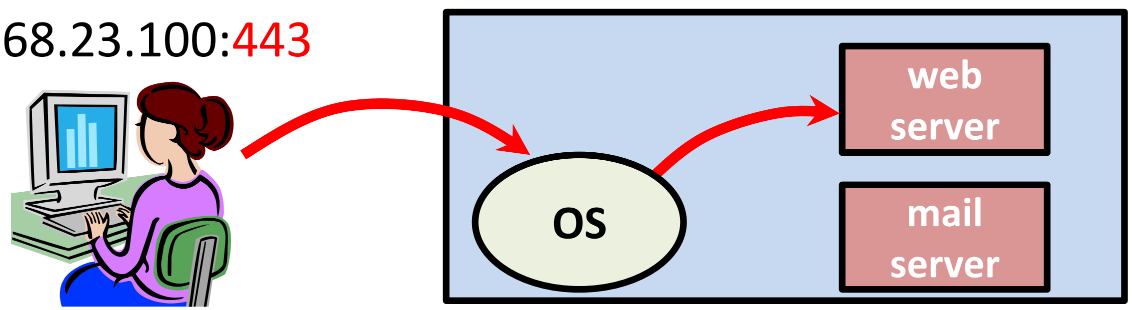
Requesting a non-secure web page

192.168.23.100:80



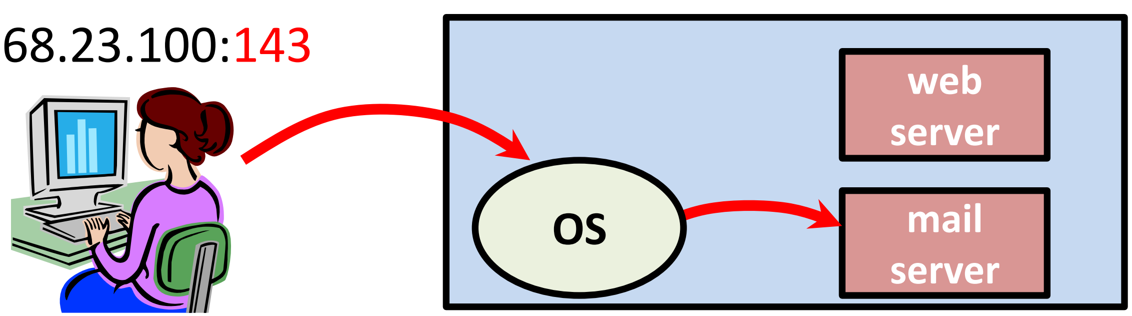
Requesting a secure web page

192.168.23.100:443



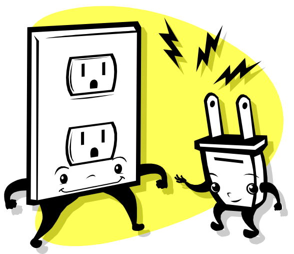
Requesting new email messages

192.168.23.100:143



Sockets

- Socket API (applications programming interface)
  - Originally in Berkeley Unix
    - Thus: Berkeley sockets, BSD sockets
  - De facto standard in all operating systems
  - Functions called by client, by server, or by both:
    - `socket()`, `bind()`, `connect()`, `listen()`, `accept()`, `send()`, `recv()`, `sendto()`, `recvfrom()`, `close()`
  - Use integer file descriptor (like reading/writing from a file)



High-level process

```
// Fire up connection
// to the server
getaddrinfo()
socket()
connect()

// Exchange data
while (!done)
{
  send()
  recv()
}

// Shutdown
close()
```

Client program

```
// Initial socket setup
getaddrinfo()
socket()
bind()
listen()
while (1)
{
  // Wait for new caller
  accept()

  // Exchange data
  while (!done)
  {
    recv()
    send()
  }
  // Disconnect
  close()
}
```

Server program

Client/Server: initial setup

- Prepare some stuff you'll need later

```
int getaddrinfo(const char *node,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);

node     - e.g. "www.example.com" or IP address
service  - e.g. "http" or port number (as a string)
hints    - already filled in struct with things like
           IPv4/IPv6 or stream/datagram
res      - result, needed by socket(), connect(), bind()
           NOTE: free up after use with freeaddrinfo()

Returns 0 on success.
```

Client: creation

- Creating a socket

```
int socket(int domain, int type, int protocol);

domain   - PF_INET for IPv4
type     - SOCK_STREAM for reliable byte stream (TCP)
protocol - normally set to 0

Returns -1 on failure.
```

domain	
PF_INET	Internet family (IPv4)
PF_UNIX	Unix pipe
PF_PACKET	Direct network access (bypasses TCP/IP stack)

type	
SOCK_STREAM	Reliable stream service
SOCK_DGRAM	Message oriented, such as UDP

Client: connecting

- Contact server for connection
  - Associate socket handle with server address + port
  - Obtain a local port number (assigned by OS)
  - Request a connection with server

```
int connect(int sockfd, struct sockaddr *serv_addr,
            int addrlen);

sockfd     - the socket descriptor
serv_addr  - struct contain server info
addrlen    - length of serv_addr struct

Returns -1 on failure.
```

Client: sending and receiving

- Finally let's exchange some data!

```
int send(int sockfd, const void *msg, int len, int flags);
int recv(int sockfd, void *buf, int len, int flags);

sockfd - socket descriptor
msg    - pointer to buffer to be sent/received
len    - length of buffer
flag   - normally 0

Returns bytes sent or received.  NOTE: send() may send fewer
bytes than requested for big messages!
```

Server: get ready to rock

- Create a socket
  - Server usually knows its port (nobody else better be using it)

```
int socket(int domain, int type, int protocol);
```

- Bind to address + port

```
int bind(int sockfd, struct sockaddr *my_addr,
         int addrlen);

sockfd  - description return by socket()
my_addr - struct contain info about address/port
addrlen - length of address

Returns -1 on failure.
```

Server: maximum backlog

- Many clients may request service
  – Server can’t handle all at once
- Server specifies maximum pending

```
int listen(int sockfd, int backlog);

sockfd  - socket descriptor
backlog - maximum number of pending connections

Returns -1 on failure.
```

Server: accepting clients

- Server waits until client arrives
- Accept a new client connection

```
int accept(int sockfd, struct sockaddr *serv_addr,
            int addrlen);

sockfd      - the socket descriptor
serv_addr   - struct contain info about client
addrlen     - length of serv_addr struct

Returns new socket descriptor for the accepted connection.
```

Server: handling concurrency

- Server could serialize work
  - Service one client from start to finish
  - Move to the next one
  - Allow backlog to queue up waiting clients
- But client request could be long, resource bound, etc.
  - Spawn process/thread for each accepted client

Server: handling concurrency

```
// Initial socket setup
getaddrinfo()
socket()
bind()
listen()
while (1)
{
  // Wait for new caller
  accept()

  // Exchange data
  while (!done)
  {
    recv()
    send()
  }
  // Disconnect
  close()
}
```

thread 1

thread 2

thread 3

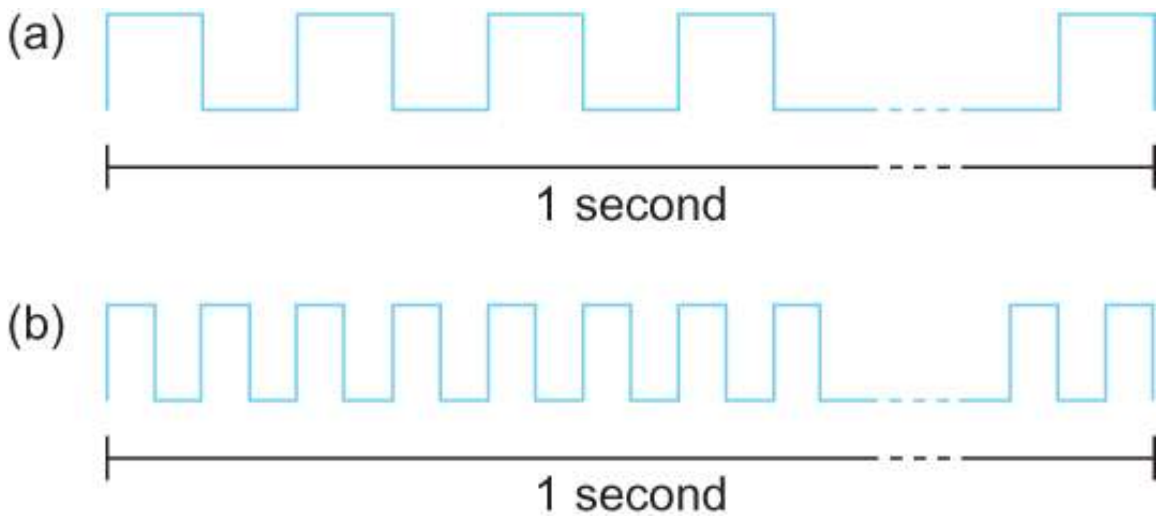
Server program

Performance

Bandwidth

- ***Bandwidth*** - measure of the frequency band
  - e.g. voice telephone line supports frequencies from 300 Hz - 3300 Hz, bandwidth = 3000 Hz
- ***Bandwidth*** - bits transmitted per unit time
  - 1 Mbps = 1 x 10^6 bits/second
  - e.g. 802.11g wireless has a bandwidth of 54 Mbps
    - Bandwidth, mega = 1 x 10^6 = 1000000
    - File size, mega = 2^{20} = 1048576
- ***Throughput*** - actual obtainable performance
  - e.g. 802.11g wireless has a throughput ~22 Mbps

Bandwidth



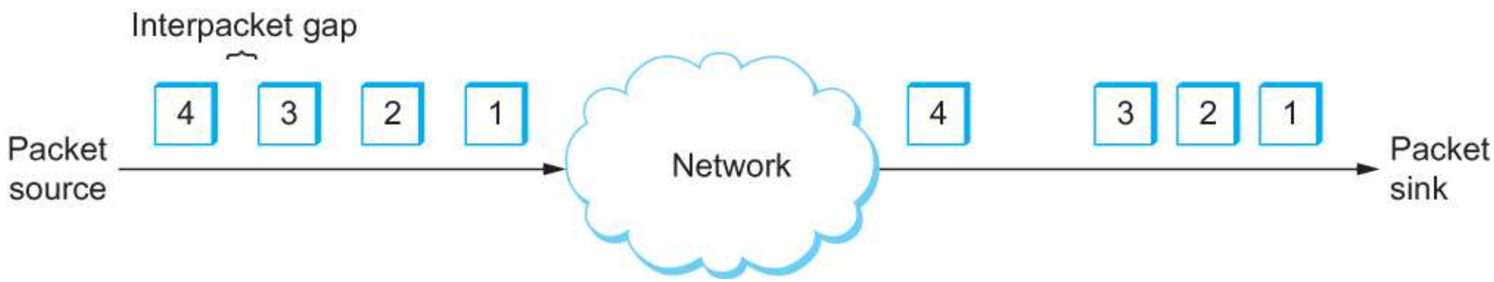
(a) bits transmitted at 1 Mbps (each bit is 1×10^{-6} seconds wide)
(b) bits transmitted at 2 Mbps (each bit is 0.5×10^{-6} seconds wide)

Watch your units!

- Bandwidth
  - gigabits (Gbps) = 10^9 bits/second
  - megabits (Mbps) = 10^6 bits/second
  - kilobits (Kbps) = 10^3 bits/second
- File sizes
  - 8 bits / byte
  - gigabyte (GB) = 2^{30} bytes
  - megabyte (MB) = 2^{20} bytes
  - kilobyte (KB) = 2^{10} bytes

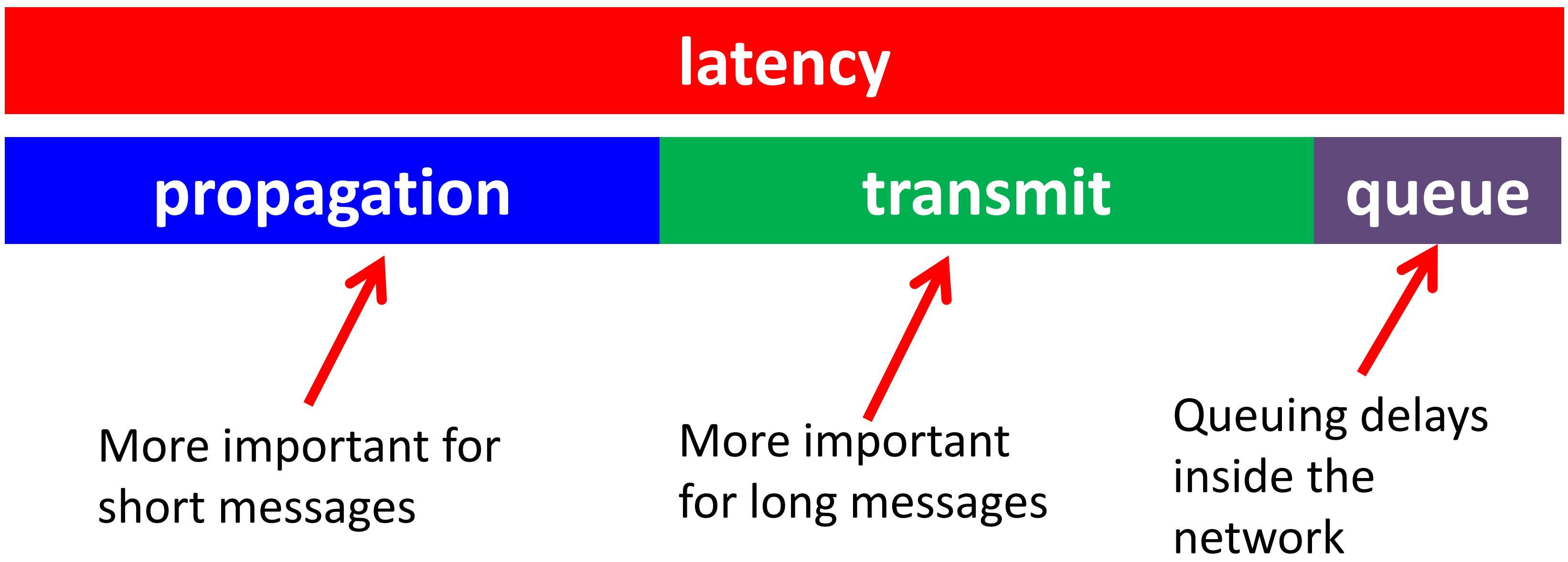
Latency

- ***Latency*** or ***delay*** - how long it takes a message to go from one end of network to other
  - Measured in units of time (often ms)
- ***Round-trip time (RTT)*** - how long from source to destination and back to source
- ***Jitter*** - variance in latency (affects time sensitive applications)



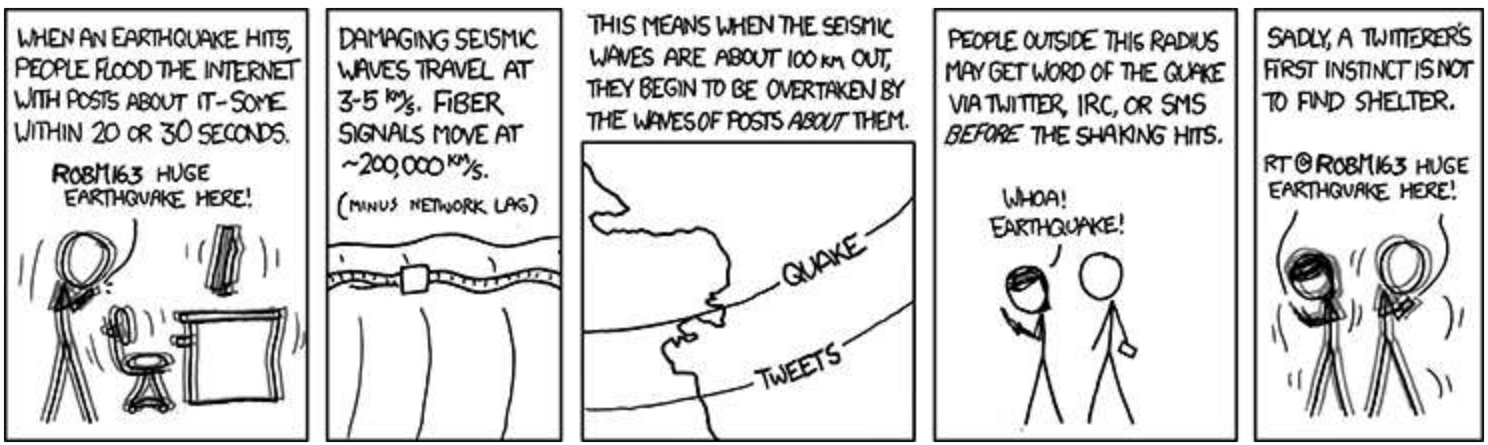
Latency

- latency = propagation + transmit + queue
- propagation = distance / speed of light
- transmit = size / bandwidth



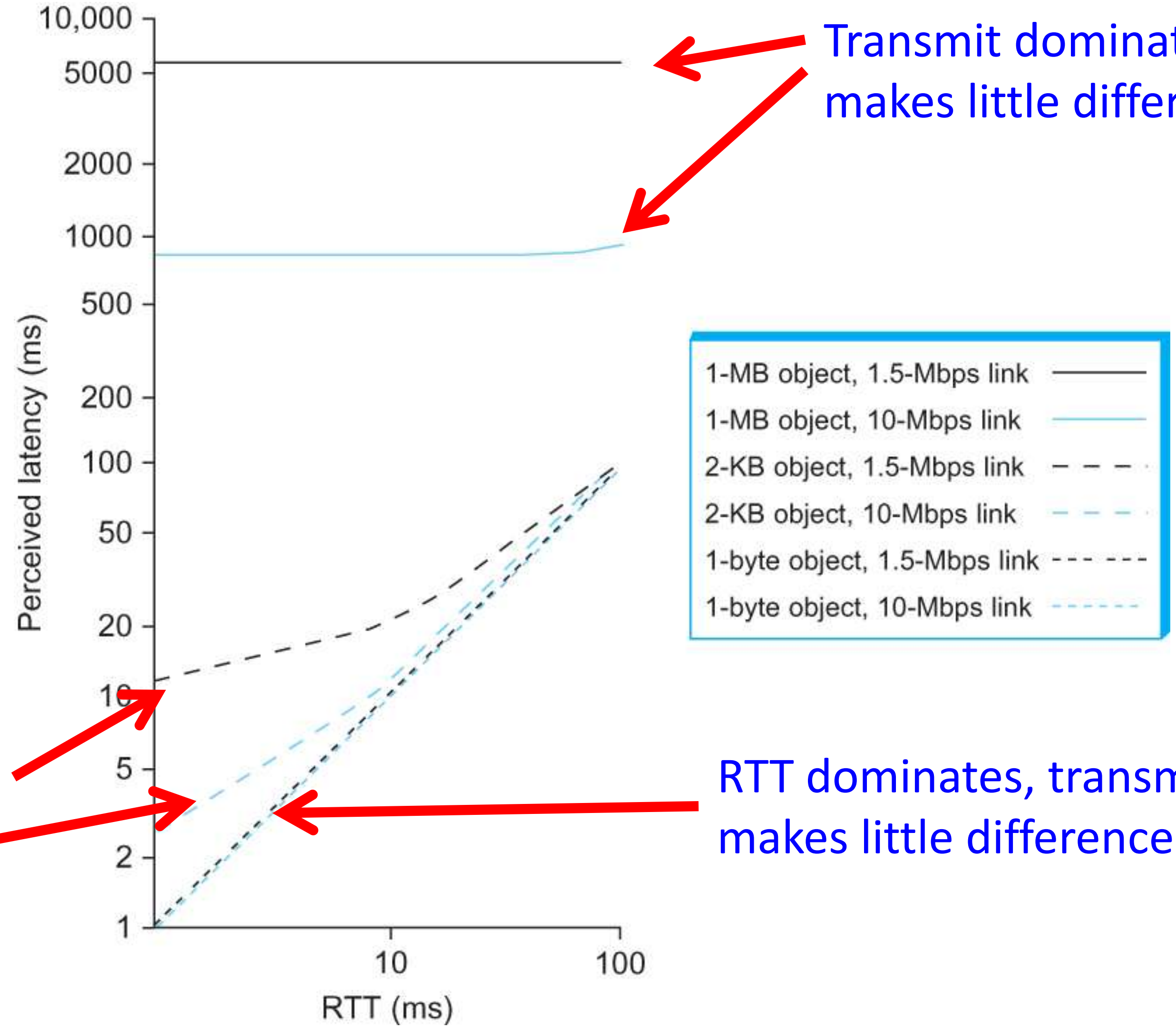
Speed of light

Medium	Speed of light
Vacuum	3.0×10^8 m/s
Copper cable	2.3×10^8 m/s
Optical fiber	2.0×10^8 m/s



http://xkcd.com/723/

Latency example



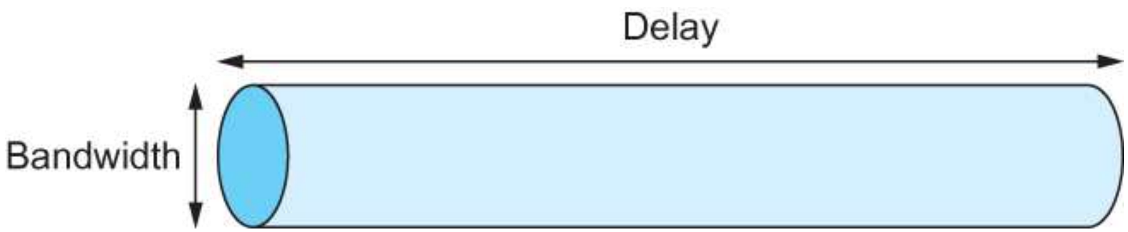
Transmit dominates, RTT makes little difference

Transmit important until RTT gets big

RTT dominates, transmit makes little difference

Delay x Bandwidth

- Channel is like a hollow pipe
- Latency is length, bandwidth is width



- delay = 50 ms, bandwidth of 45 Mbps

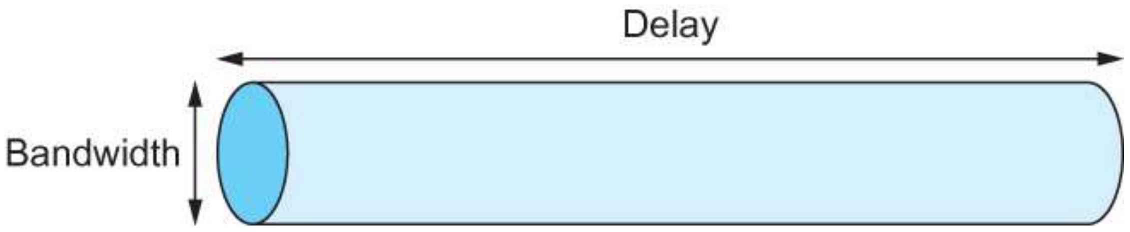
  $(50 \times 10^{-3}$ sec$) \times (45 \times 10^{6}$ bits/sec$)$

  2.25×10^{6} bits $\times$ (1 byte/8 bits) $\times$ (1 KB/2^{10} bytes) =

  275 KB data (how much fits in the pipe)

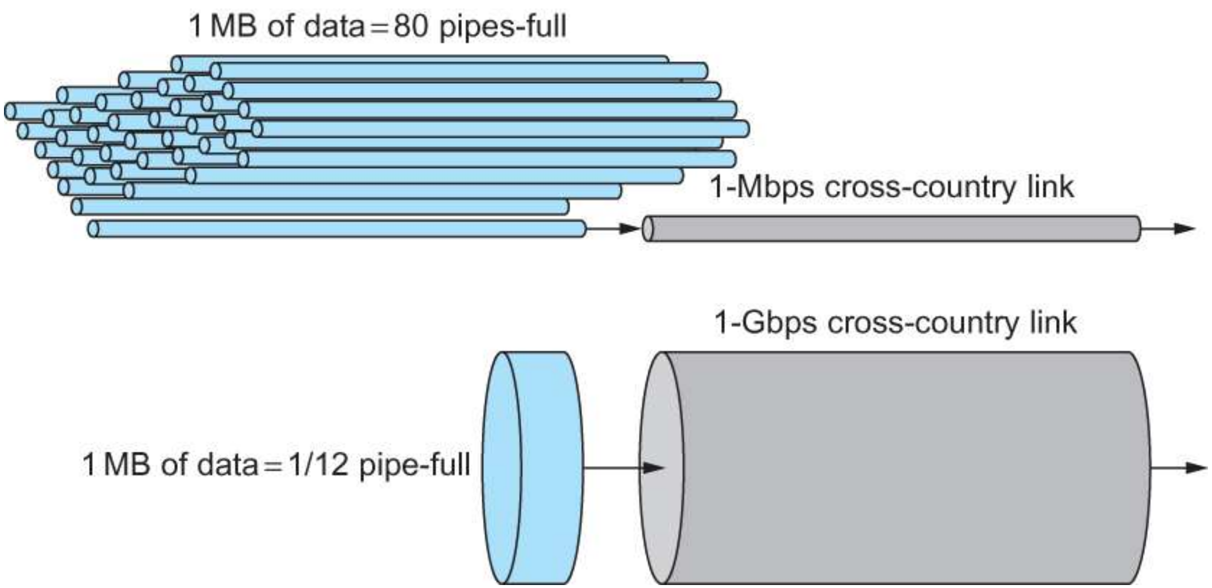
Delay x Bandwidth

- Often we consider RTT as the delay
  - Takes RTT = 2 x latency to hear back from receiver
- If sender wants to keep pipe full:
  - Delay x Bandwidth = # bits transmitted before hearing from receiver all is well, “bits in flight”
  - Delay x Bandwidth = # bits sent before waiting for signal from receiver



High speed networks

- Bandwidth increasing dramatically
- But speed of light is constant



1 MB file, 1-Mbps link with RTT of 100ms, 80 full pipes
1 MB file, 1-Gbps link with RTT of 100ms, 1/12 of a full pipe

High speed networks

- Throughput = Transfer size / Transfer time
- Transfer time = RTT + 1/Bandwidth x Transfer size

File size (MB)	RTT	Bandwidth (Gbps)	Transmit time (ms)	Transfer time (ms)	Throughput (Mbps)
0.25	100	1	2.1	102.1	19.6
0.50	100	1	4.2	104.2	38.4
1	100	1	8.4	108.4	73.8
2	100	1	16.8	116.8	137.0
4	100	1	33.6	133.6	239.6
8	100	1	67.1	167.1	383.0
16	100	1	134.2	234.2	546.5

Summary

- Overview of socket API
  - Very common thing to use
- Measuring network performance
  - Bandwidth, how frequently bits can be sent
  - Latency, how long the bits take to get there
  - High speed networks
    - RTT starts to dominate