CS144 – Introduction to Computer Networking

Instructors: Philip Levis and David Mazières

CAs: Roger Liao and Samir Selman

Section Leaders: Saatvik Agarwal, Juan Batiz-Benet, and Tom Wiltzius

cs144-staff@scs.stanford.edu

http://cs144.scs.stanford.edu/

Networks class

- **Goal: Teach the concepts underlying networks**
  - How do networks work? What can one do with them?
  - Give you experience using and writing protocols
  - Give you tools to understand new protocols & applications
  - Not: train you on all the latest "hot" technologies
- **Prerequisites:**
  - CS110 or equiv; class assumes good knowledge of C, some socket programming helpful (e.g., CS110 web server)

Administrivia

- **All assignments are on the web page**
- **Text: Kurose & Ross, *Computer Networking: A Top-Down Approach, 4th or 5th edition***
  - Instructors working from 4th edition, either OK
  - Don't need lab manual or Ethereal (used book OK)
- **Syllabus on web page**
  - Gives which textbook chapters correspond to lectures (Lectures and book topics will mostly overlap)
  - Extra (not required) questions for further understanding
  - Papers sometimes, to make concepts more concrete (Read the papers before class for discussion)
  - Subject to change! (Reload before checking assignments)

Administrivia 2

- **Send all assignment questions to newsgroup**
  - Someone else will often have the same question as you
  - Newsgroup su.class.cs144 dedicated to class
  - For information on accessing Usenet, see `http://www.stanford.edu/services/usenet/`
- **Send all staff communication to `cs144-staff` list**
  - Goes to whole staff, so first available person can respond
  - CCing list ensures we give students consistent information
  - Also, some of us get lots of email. . . much easier for us to prioritize a specific mailing list

Grading

- **Exams: Midterm & Final**
- **Homework**
  - 5 lab assignments implemented in C
- **Grading**
  - Exam grade = $\max(\text{final}, (\text{final} + \text{midterm})/2)$
  - Final grade will be computed as:

$$(1 - r)\left(\frac{\text{exam} + \text{lab}}{2}\right) + r \cdot \max(\text{exam}, \text{lab})$$

  - r may vary per student, expect average to be $\sim 1/3$
- **Possible ideas for computing r**
  - Maybe a problem set, other kind of lab, or pop quizzes

Labs

- **Labs are due by the beginning of class**
  - Lab 1: Stop & wait
  - Lab 2: Reliable transport
  - Lab 3: Static routing
  - Lab 4: NAT
  - Lab 5: Dynamic routing
- **All assignments due at start of lecture**
  - Free extension to midnight if you come to lecture that day

Late Policy

- **No credit for late assignments w/o extension**
- **Contact** `cs144-staff` **if you need an extension**
  - We are nice people, so don't be afraid to ask
- **Most likely to get an extension when all of the following hold:**
  1. You ask *before* the original deadline,
  2. You tell us where you are in the project, and
  3. You tell us when you can finish by.

Topics

- **Network programming (sockets, RPC)**
- **Network (esp. Internet) architecture**
  - Switching, Routing, Congestion control, TCP/IP, Wireless networks
- **Using the network**
  - Interface hardware & low-level implementation issues, Naming (DNS), Error detection, compression
- **Higher level issues**
  - Encryption and Security, caching & content distribution, Peer-to-peer systems

Networks

- **What is a network?**
  - A system of lines/channels that interconnect
  - E.g., railroad, highway, plumbing, communication, telephone, computer
- **What is a *computer* network?**
  - A form of communication network—moves information
  - Nodes are general-purpose computers
- **Why study computer networks?**
  - Many nodes are general-purpose computers
  - *You* can program the nodes
  - Very easy to innovate and develop new uses of network
  - Contrast: Old PSTN – all logic is in the core

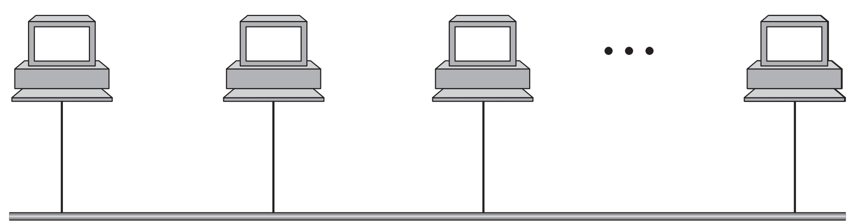
Building blocks

- **Nodes: Computers, dedicated routers, . . .**
- **Links: Coax, twisted pair, fibers, radio . . .**

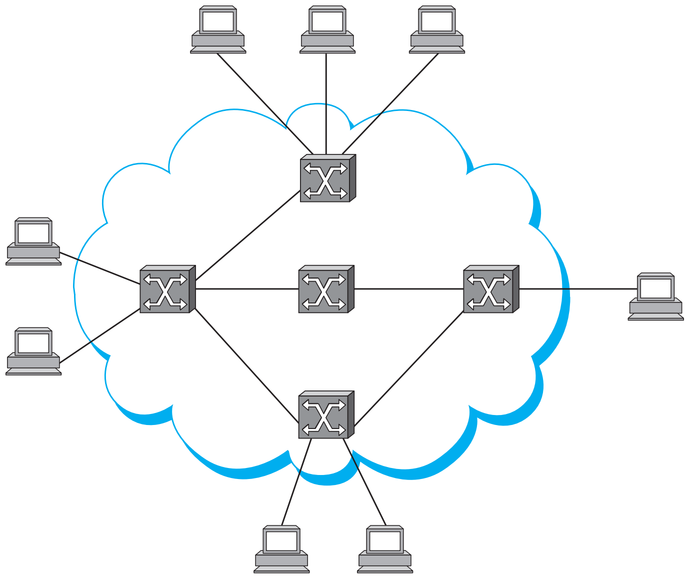
  (a) point-to-point

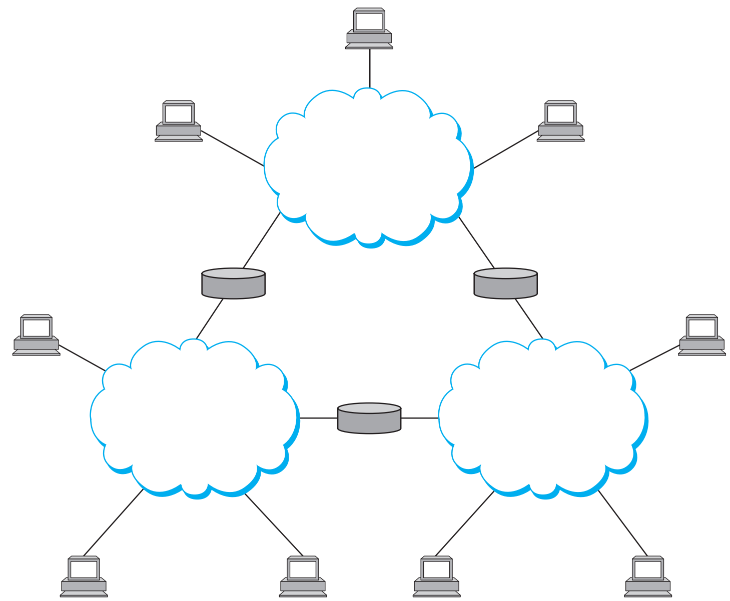
  (b) multiple access – every node sees every packet

(a)


(b)


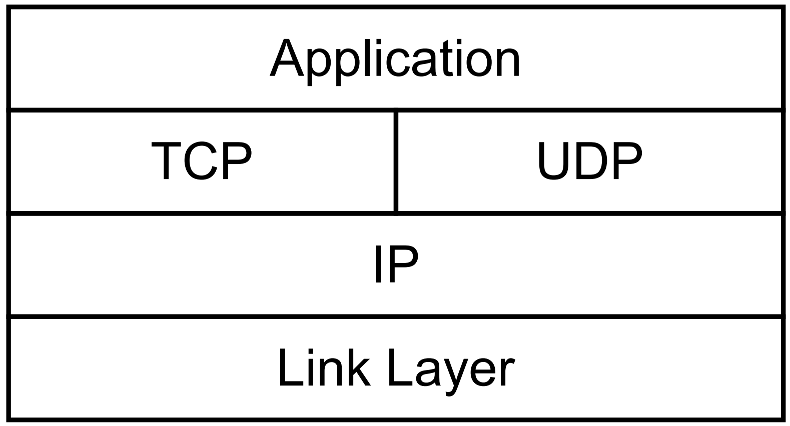
From Links to Networks





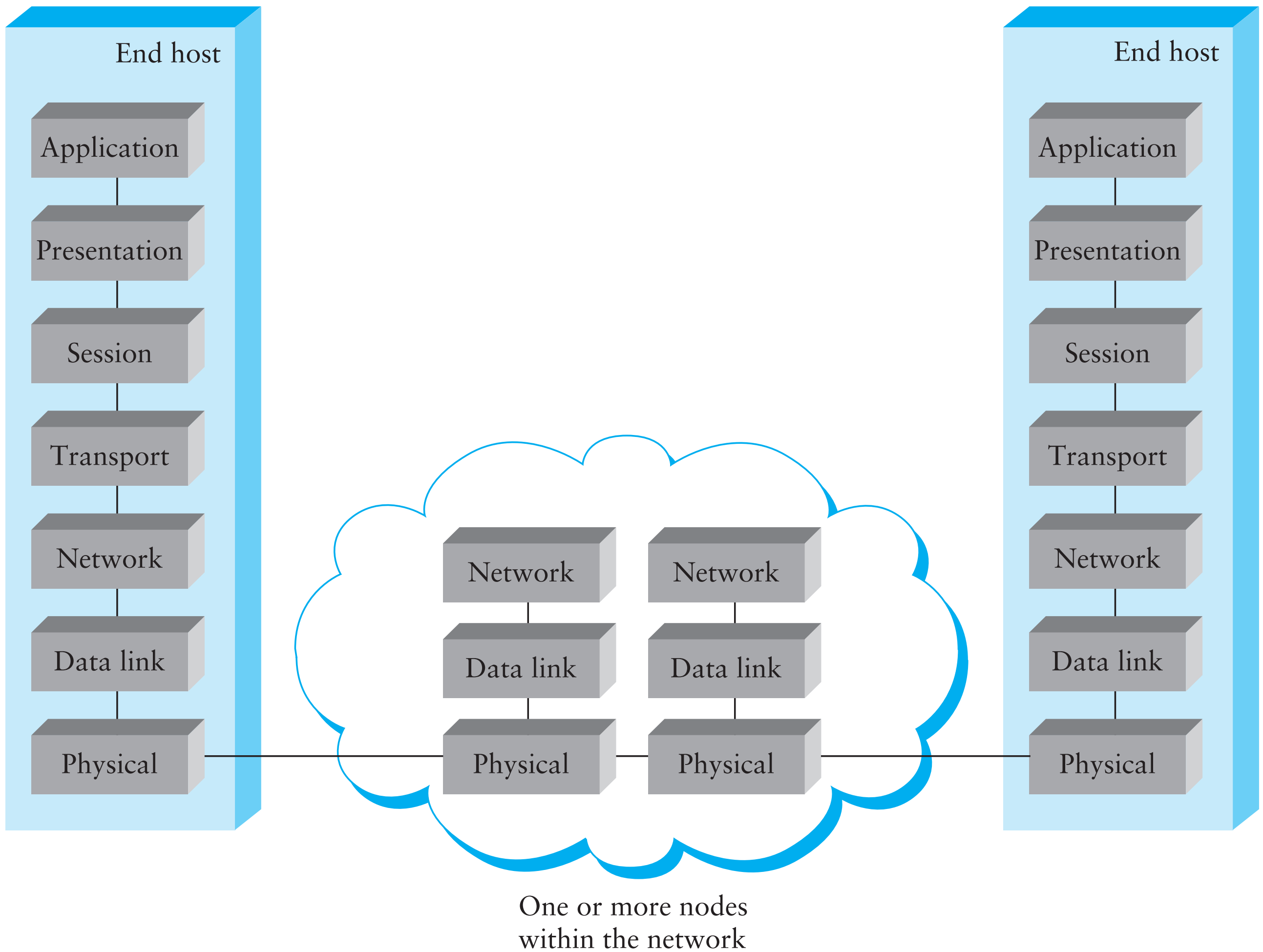
- **To scale to more nodes, use *switching***
  - nodes can connect multiple other nodes, or
  - Recursively, one node can connect multiple networks

Protocol layering



- **Can view network encapsulation as a stack**
- **A network packet from A to D must be put in link packets A to B, B to C, and C to D**
  - Each layer produces packets that become the payload of the lower-layer's packets
  - This is *almost* correct, but TCP/UDP "cheat" to detect certain errors in IP-level information like address

OSI layers



- **Layers typically fall into 1 of 7 categories**

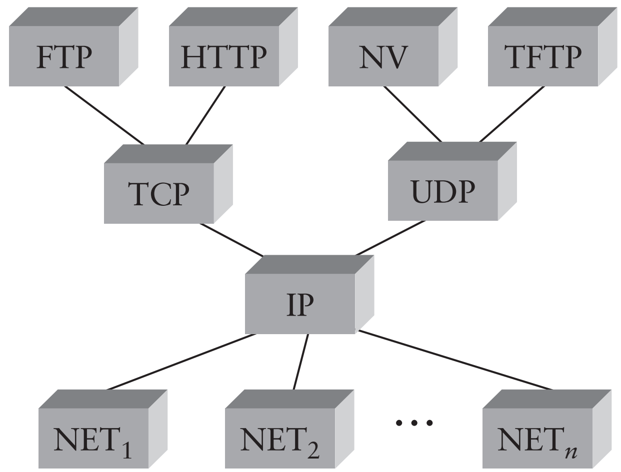
Layers

- Physical – sends individual bits
- Data link – sends *frames*, handles access control to shared media (e.g., coax)
- Network – delivers packets, using *routing*
- Transport – demultiplexes, provides reliability & flow control
- Session – can tie together multiple streams (e.g., audio & video)
- Presentation – crypto, conversion between representations
- Application – what end user gets, e.g., HTTP (web)

Addressing

- **Each node typically has unique *address***
  - (or at least is made to think it does when there is shortage)
- **Each layer can have its own addressing**
  - Link layer: e.g., 48-bit Ethernet address (interface)
  - Network layer: 32-bit IP address (node)
  - Transport layer: 16-bit TCP port (service)
- ***Routing* is process of delivering data to destination across multiple link hops**
- **Special addresses can exist for broadcast/multicast**

Hourglass



- **Many application protocols over TCP & UDP**
- **IP works over many types of network**
- **This is “Hourglass” philosophy of Internet**
  - Idea: If everybody just supports IP, can use many different applications over many different networks
  - In practice, some claim narrow waist is now network *and* transport layers, due to NAT (lecture 12)

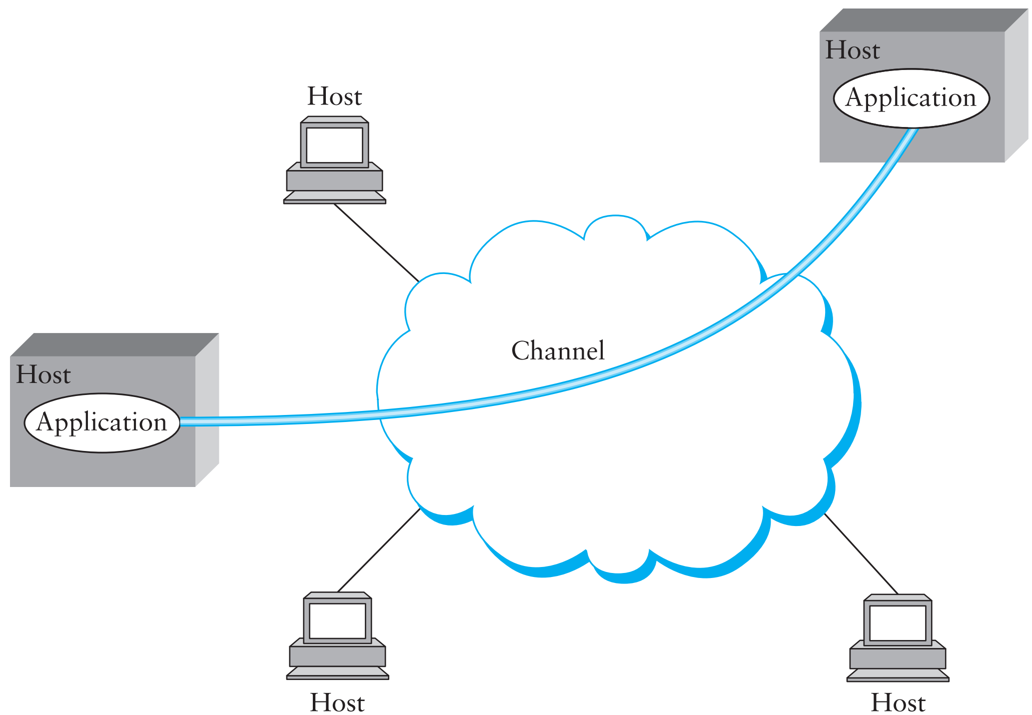
Internet protocol

- **Most computer nets connected by Internet protocol**
  - Runs over a variety of physical networks, so can connect Ethernet, Wireless, people behind modem lines, etc.
- **Every host has[a] a unique 4-byte IP address**
  - E.g., `www.ietf.org` → 132.151.6.21
  - Given a node's IP address, the network knows how to route a packet (lectures 3+4)
  - Next generation IPv6 uses 16-byte host addresses
- **But how do you build something like the web?**
  - Need naming (look up www.ietf.org) – DNS (lecture 8)
  - Need API for browser, server (CS110/this lecture)
  - Need demultiplexing within a host—E.g., which packets are for web server, which for mail server, etc.? (lecture 4)

---

[a] or thinks it has

Inter-process communication



- **Want abstraction of inter-process (not just inter-node) communication**
- **Solution: *Encapsulate* another protocol within IP**

UDP and TCP

- **UDP and TCP most popular protocols on IP**
  - Both use 16-bit *port* number as well as 32-bit IP address
  - Applications *bind* a port & receive traffic to that port
- **UDP – unreliable datagram protocol**
  - Exposes packet-switched nature of Internet
  - Sent packets may be dropped, reordered, even duplicated (but generally not corrupted)
- **TCP – transmission control protocol**
  - Provides illusion of a reliable "pipe" between to processes on two different machines (lecture 5)
  - Handles congestion & flow control (lecture 6)

Uses of TCP

- **Most applications use TCP**
  - Easier interface to program to (reliability, lecture 5)
  - Automatically avoids congestion (don't need to worry about taking down network, lecture 6)
- **Servers typically listen on well-known ports**
  - SSH: 22
  - Email: 25
  - Finger: 79
  - Web / HTTP: 80
- **Example: Interacting with www.stanford.edu**

Programming Sockets

- **Book has Java source code**
- **CS144 is in C**
  - Many books and internet tutorials
- **Berkeley sockets API**
  - Bottom-level OS interface to networking
  - Important to know and do once
  - Higher-level APIs build on them

Quick CS110 review: System calls

- **System calls invoke code in the OS kernel**
  - Kernel runs in a more privileged mode than application
  - Can execute special instructions that application cannot
  - Can interact directly with devices such as network card
- **Higher-level functions built on syscall interface**
  - `printf`, `scanf`, `gets`, etc. all user-level code

File descriptors

- **Most IO done on file descriptors**
  - Small integers referencing per-process table in the kernel
- **Examples of system calls with file descriptors:**
  - `int open(char *path, int flags, ...);`
    - Returns new file descriptor bound to file `path`
  - `int read (int fd, void *buf, int nbytes);`
    - Returns number of bytes read
    - Returns 0 bytes at end of file, or -1 on error
  - `int write (int fd, void *buf, int nbytes);`
    - Returns number of bytes written, -1 on error
    - (Never returns 0 if `nbytes` $>$ 0)
  - `int close (int fd);`
    - Deallocates file descriptor (not underlying I/O resource)

Error returns

- **What if syscall failes? E.g.** `open` **non-existent file?**
  - Returns -1 (invalid fd number)
- **Most system calls return -1 on failure**
  - Always check for errors when invoking system calls
  - Specific kind of error in global int `errno`
    (But `errno` will be unchanged if syscall did not return -1)
- `#include <sys/errno.h>` **for possible values**
  - 2 = `ENOENT` "No such file or directory"
  - 13 = `EACCES` "Permission Denied"
- `perror` **function prints human-readable message**
  - `perror ("initfile");`
    → "`initfile: No such file or directory`"

Sockets: Communication between machines

- **Network sockets are file descriptors too**
- **Datagram sockets: Unreliable message delivery**
  - With IP, gives you UDP
  - Send atomic messages, which may be reordered or lost
  - Special system calls to read/write: `send`/`recv`, `sendto`/`recvfrom`, and `sendmsg`/`recvmsg` (most general)
- **Stream sockets: Bi-directional pipes**
  - With IP, gives you TCP
  - Bytes written on one end read on the other
  - Reads may not return full amount requested—must re-read

Socket naming

- **Recall how TCP & UDP name communication endpoints**
  - 32-bit IP address specifies machine
  - 16-bit TCP/UDP port number demultiplexes within host
  - Well-known services "listen" on standard ports: finger—79, HTTP—80, mail—25, ssh—22
  - Clients connect from arbitrary ports to well known ports
- **A *connection* can be named by 5 components**
  - Protocol (TCP), local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP

System calls for using TCP

Client	Server
	`socket` – make socket
	`bind` – assign address
	`listen` – listen for clients
`socket` – make socket	
`bind`* – assign address	
`connect` – connect to listening socket	
	`accept` – accept connection

*This call to `bind` is optional; `connect` can choose address & port.

Socket address structures

- **Socket interface supports multiple network types**
- **Most calls take a generic `sockaddr`:**

```
struct sockaddr {
  uint16_t  sa_family;   /* address family */
  char      sa_data[14]; /* protocol-specific address */
};                       /* (may be longer than this) */

int connect(int fd, const struct sockaddr *, socklen_t);
```

- **Cast `sockaddr *` from protocol-specific struct, e.g.:**

```
struct sockaddr_in {
  short   sin_family;        /* = AF_INET */
  u_short sin_port;          /* = htons (PORT) */
  struct  in_addr sin_addr;  /* 32-bit IPv4 address */
  char    sin_zero[8];
};
```

Dealing with address types [RFC 3493]

- **All values in network byte order (big endian)**
  - `htonl` converts 32-bit value from host to network order
  - `ntohl` converts 32-bit value from network to host order
  - `ntohs`/`htons` same for 16-bit values
- **All address types begin with family**
  - `sa_family` in `sockaddr` tells you actual type
- **Unfortunately, not address types the same size**
  - E.g., `struct sockaddr_in6` is typically 28 bytes, yet generic `struct sockaddr` is only 16 bytes
  - So most calls require passing around socket length
  - Can simplify code with new generic `sockaddr_storage` big enough for all types (but have to cast between *3* types now)

Looking up a socket address w. `getaddrinfo`

```
struct addrinfo hints, *ai;
int err;

memset (&hints, 0, sizeof (hints));
hints.ai_family = AF_UNSPEC;     /* or AF_INET or AF_INET6 */
hints.ai_socktype = SOCK_STREAM; /* or SOCK_DGRAM for UDP */

err = getaddrinfo ("www.stanford.edu", "http", &hints, &ai);
if (err)
  fprintf (stderr, "%s\n", gia_strerror (err));
else {
  /* ai->ai_family  = address type (AF_INET or AF_INET6) */
  /* ai->ai_addr    = actual address cast to (sockaddr *) */
  /* ai->ai_addrlen = length of actual address */
  freeaddrinfo (ai); /* must free when done! */
}
```

Address lookup details

- `getaddrinfo` **notes:**
  - Can specify port as service name or number (e.g., `"80"` or `"http"`, allows possibility of dynamically looking up port)
  - May return multiple addresses (chained with `ai_next` field)
  - You must free structure with `freeaddrinfo`
- **Other useful functions to know about**
  - `getnameinfo` – Lookup hostname based on address
  - `inet_ntop` – convert IPv4 or 6 address to printable form
  - `inet_pton` – convert string to IPv4 or 6 address

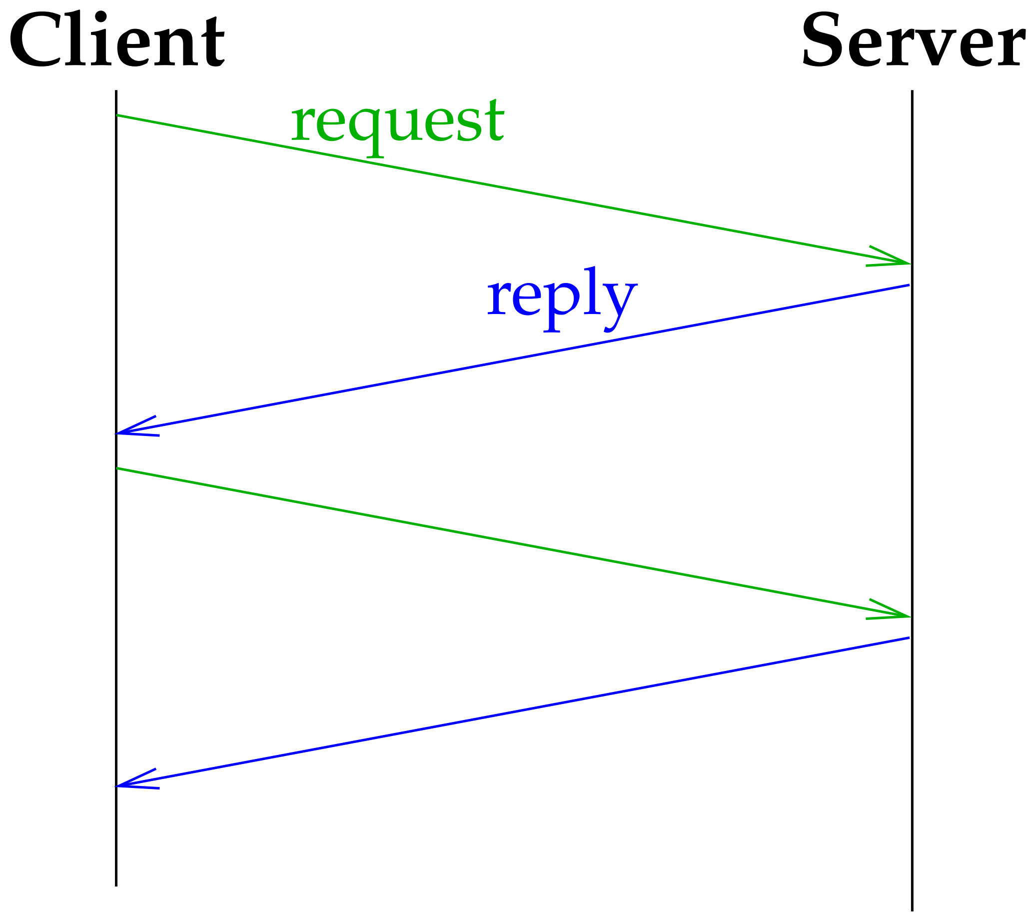
EOF in more detail

- **Simple client-server application**
  - Client sends request
  - Server reads request, sends response
  - Client reads response
- **What happens when you're done?**
  - Client wants server to read EOF to say request is done
  - But still needs to be able to read server reply – fd is not closed!

shutdown

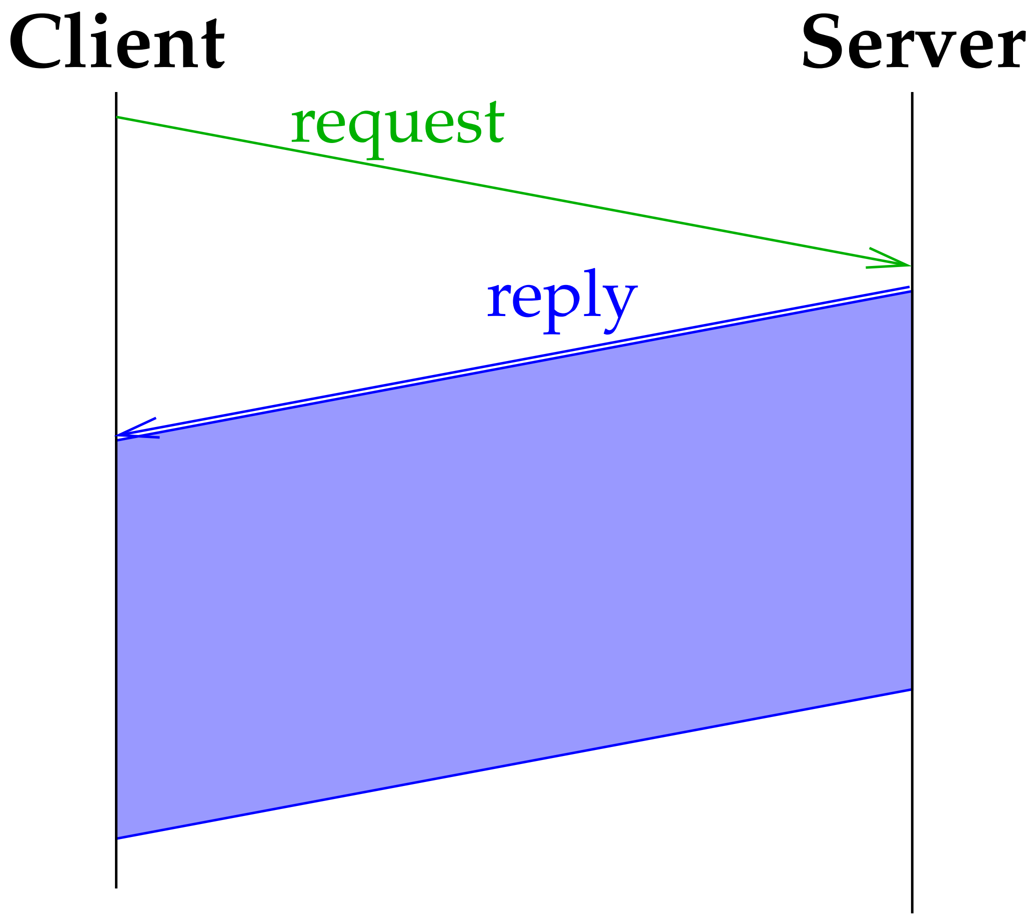
- `int shutdown (int fd, int how);`
  - Shuts down a socket w/o closing file descriptor
  - `how`: 0 = reading, 1 = writing, 2 = both
  - Note: Applies to *socket*, not descriptor—so copies of descriptor (through `dup` or `fork` affected)
  - Note 2: With TCP, can't detect if other side shuts for reading
- **Many network applications detect & use EOF**
  - Common error: "leaking" file descriptor via `fork`, so not closed (and no EOF) when you exit

Small request/reply protocol



- **Small message protocols typically dominated by latency**

Large reply protocol



- **For bulk tranfer, throughput is most important**

Performance definitions

- **Throughput – Number of bits/time you can sustain at the receiver**
  - Improves with technology
- **Latency – How long for message to cross network**
  - Propagation + Transmit + Queue
  - We are stuck with speed of light…
    10s of milliseconds to cross country
- **Goodput – TransferSize/Latency**
- **Jitter – Variation in latency**
- **What matters most for your application?**
  - We'll look at network applications next week

Today's Lecture

- **Basic networking abstractions**
  - Protocols
  - OSI layers and the Internet Hourglass
- **Transport protocols: TCP and UDP**
- **Review of file descriptors**
- **Some functions from the socket API**
- **Protocol performance tradeoffs**
- **Next lecture: Transport & reliability**

Structure of Rest of Class

- **IP and above (5 weeks)**
  - Application layers
  - Network layer: IP and routing, multicast
  - Transport layer: TCP and congestion control
  - Naming, address translation, and content distribution
- **Below IP (2 weeks)**
  - Network address translation (NAT)
  - Link and physical layers
- **Advanced topics (2 weeks)**
  - Multimedia
  - Network coding
  - Security