CSCI-1680
Application Interface

Rodrigo Fonseca

Based partly on lecture notes by David Mazières, Phil Levis, John Jannotti

Administrivia

- **Today: Network Programming mini-course! 368, 8-10pm**
- **Signup for Snowcast milestone**
  - Must sign up for a slot by Fri 3pm
  - http://bit.ly/snowcast12

Review

- **Multiplexing**
- **Layering and Encapsulation**
- **IP, TCP, UDP**

- **Today:**
  - Performance Metrics
  - Socket API
  - Concurrent servers

Circuit Switching

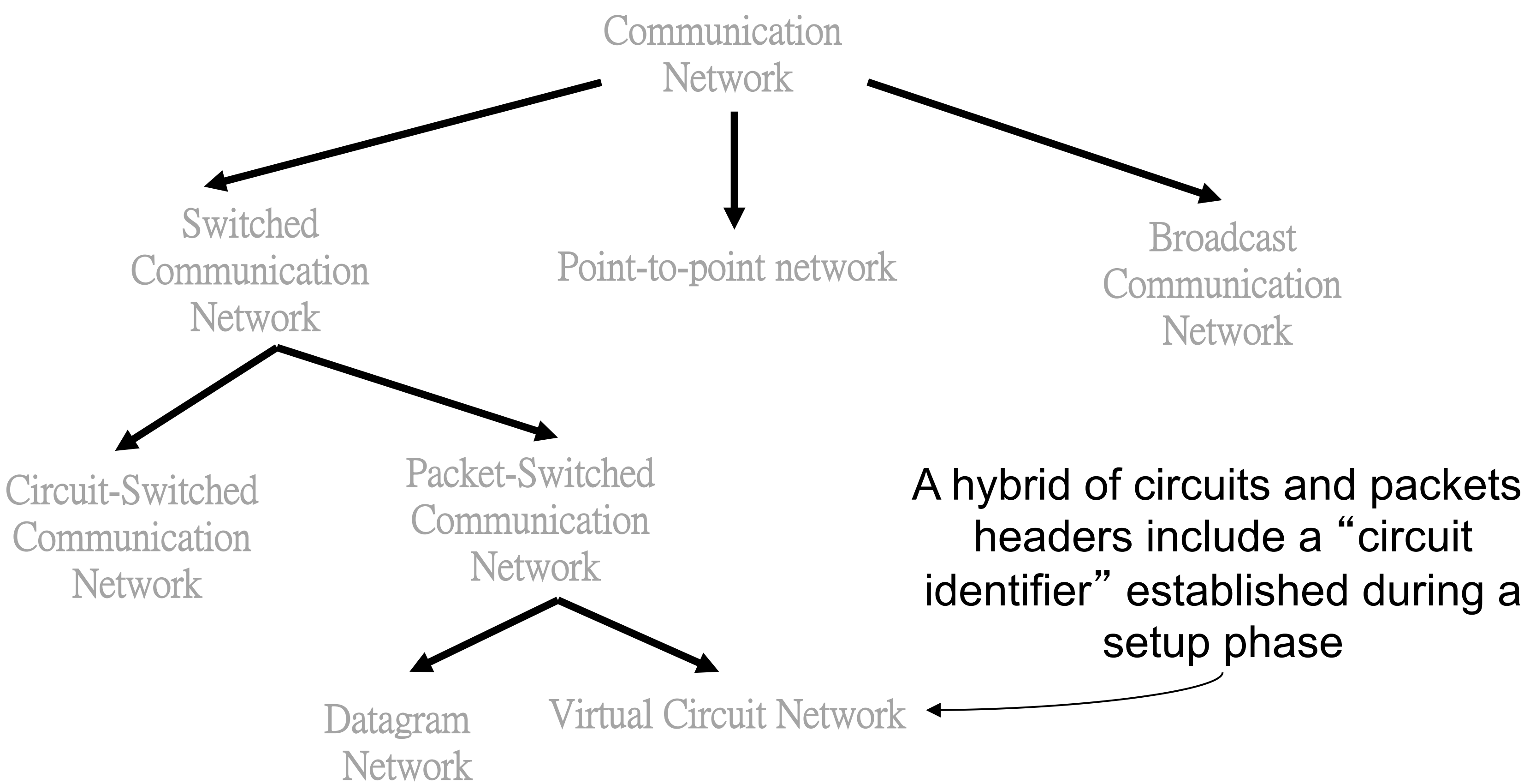
- **Guaranteed allocation**
  - Time division / Frequency division multiplexing
- **Low space overhead**
- **Easy to reason about**

- **Failures: must re-establish connection**
  - For any failures along path
- **Overload: all or nothing**
  - No graceful degradation
- **Waste: allocate for peak, waste for less than peak**
- **Set up time**

Packet Switching

- **Break information in small chunks: *packets***
- **Each packet forwarded independently**
  - Must add metadata to each packet
- **Allows statistical multiplexing**
  - High utilization
  - Very flexible
  - Fairness not automatic
  - Highly variable queueing delays
  - Different paths for each packet

A Taxonomy of networks

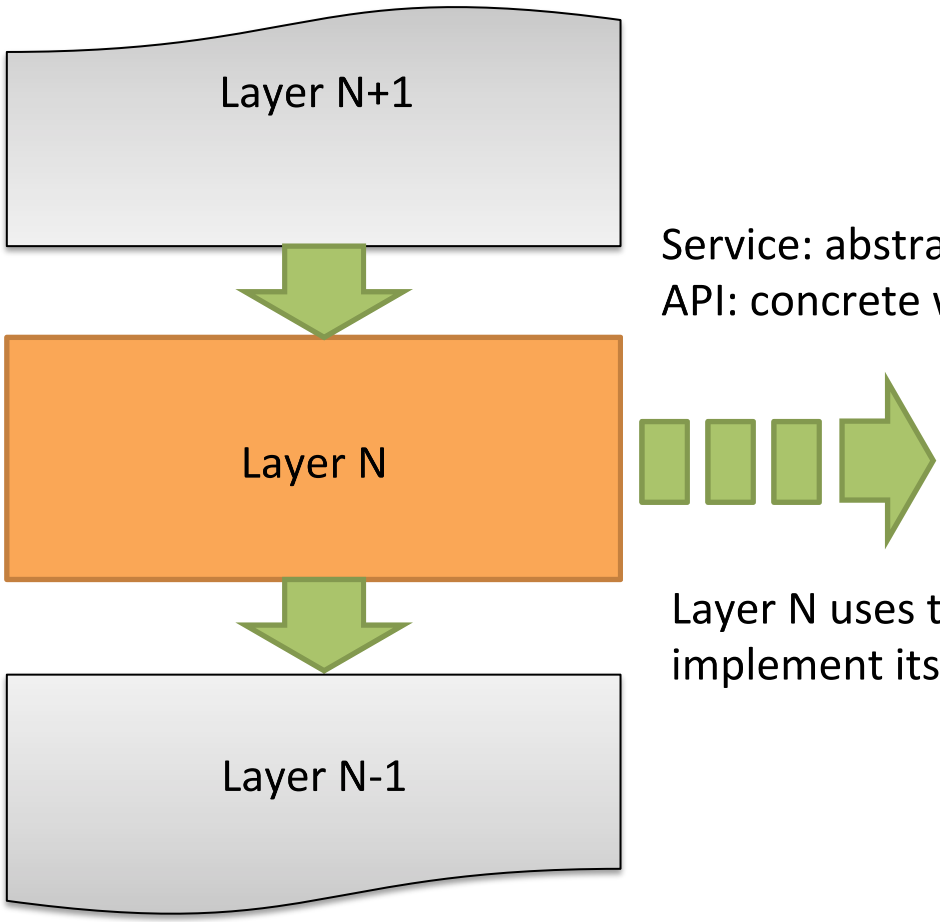


Layers, Services, Protocols

- **Last class: layering, separation of concerns**

Application	Service: user-facing application. Application-defined messages
Transport	Service: multiplexing applications Reliable byte stream to other node (TCP), Unreliable datagram (UDP)
Network	Service: move packets to any other node in the network IP: Unreliable, best-effort service model
Link	Service: move frames to other node across link. May add reliability, medium access control
Physical	Service: move bits to other node across link

Layers, Services, Protocols



Service: abstraction provided to layer above
API: concrete way of using the service

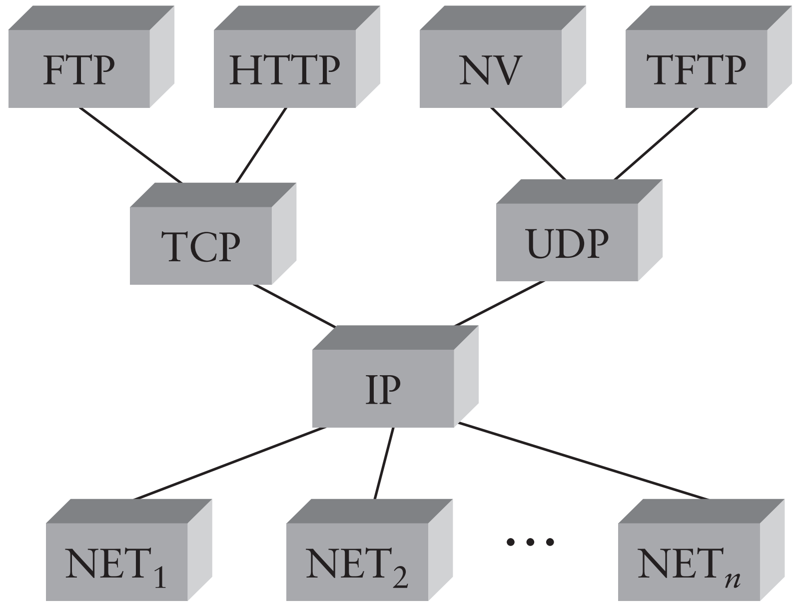
Protocol: rules for communication within same layer

Layer N uses the services provided by N-1 to implement its protocol and provide its own services

Challenge

- **Decide on how to factor the problem**
  - What services at which layer?
  - What to leave out?
  - Balance demands,
- **For example:**
  - IP offers pretty crappy service, even on top of reliable links… why?
  - TCP: offers reliable, in-order, no-duplicates service. Why would you want UDP?

IP as the Narrow Waist

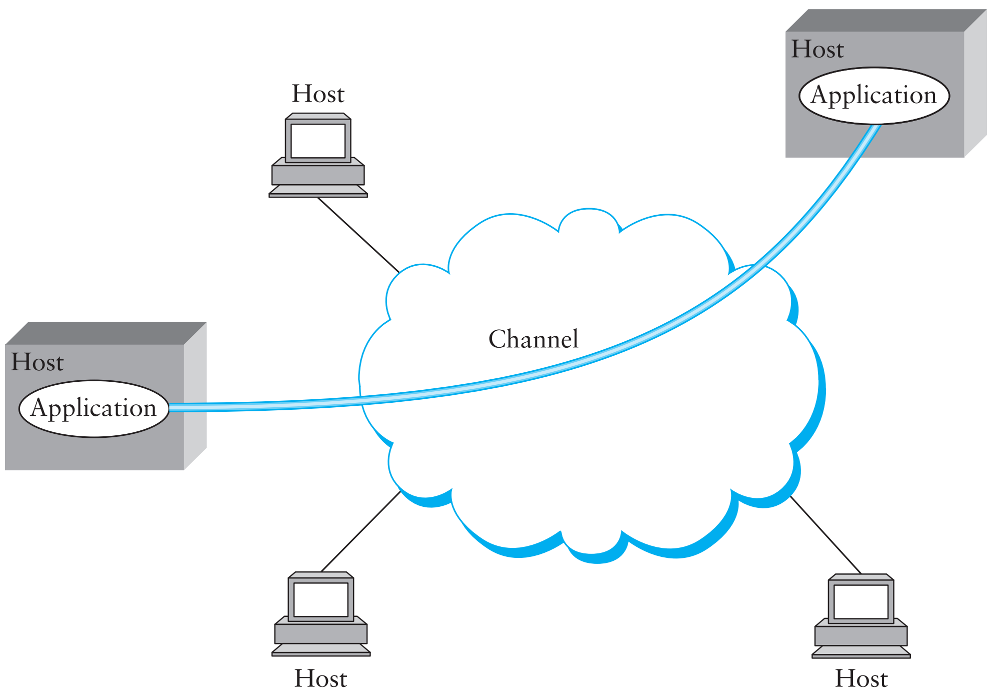


- **Many applications protocols on top of UDP & TCP**
- **IP works over many types of networks**
- **This is the "Hourglass" architecture of the Internet.**
  - If every network supports IP, applications run over many different networks (*e.g.*, cellular network)

Network Layer: Internet Protocol (IP)

- **Used by most computer networks today**
  - Runs *over* a variety of physical networks, can connect Ethernet, wireless, modem lines, etc.
- **Every host has a unique 4-byte IP address (IPv4)**
  - *E.g.*, www.cs.brown.edu →128.148.32.110
  - The *network* knows how to route a packet to any address
- **Need more to build something like the Web**
  - Need naming (DNS)
  - Interface for browser and server software (next lecture)
  - Need demultiplexing within a host: which packets are for web browser, Skype, or the mail program?

Inter-process Communication



- **Talking from host to host is great, but we want abstraction of inter-process communication**
- **Solution: *encapsulate* another protocol within IP**

Transport: UDP and TCP

- **UDP and TCP most popular protocols on IP**
  - Both use 16-bit *port* number & 32-bit IP address
  - Applications *bind* a port & receive traffic on that port
- **UDP – User (unreliable) Datagram Protocol**
  - Exposes packet-switched nature of Internet
  - Sent packets may be dropped, reordered, even duplicated (but there is corruption protection)
- **TCP – Transmission Control Protocol**
  - Provides illusion of reliable 'pipe' or 'stream' between two processes anywhere on the network
  - Handles congestion and flow control

Performance Metrics

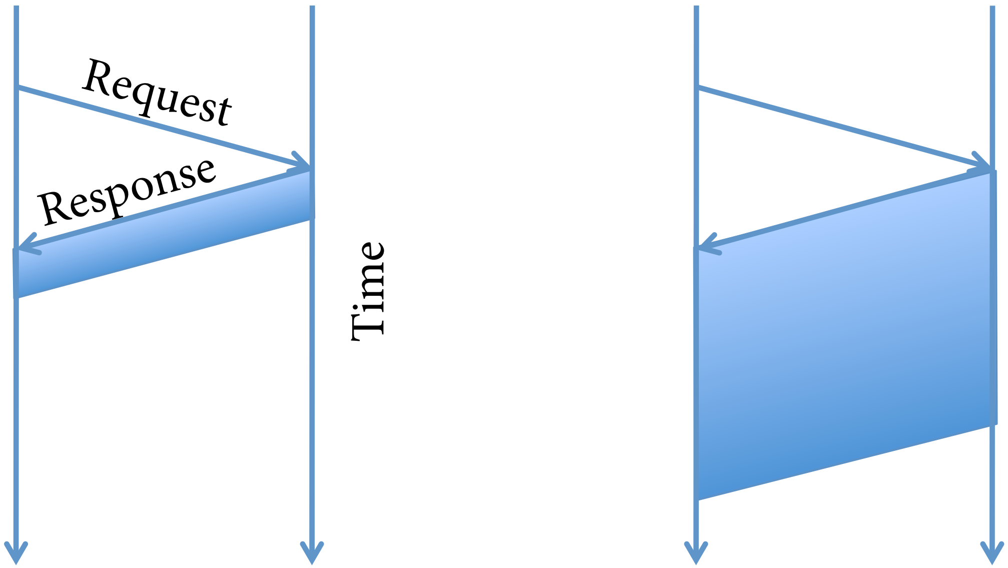
- **Throughput - Number of bits received/unit of time**
  - *e.g.* 10Mbps
- **Goodput -** ***Useful*** **bits received per unit of time**
- **Latency – How long for message to cross network**
  - Process + Queue + Transmit + Propagation
- **Jitter – Variation in latency**

Latency

- **Processing**
  - Per message, small, limits throughput
  - *e.g.* $\frac{100Mb}{s} \times \frac{pkt}{1500B} \times \frac{B}{8b} \approx 8{,}333 pkt/s$ or 120µs/pkt
- **Queue**
  - Highly variable, offered load vs outgoing b/w
- **Transmission**
  - Size/Bandwidth
- **Propagation**
  - Distance/Speed of Light

Bandwidth and Delay

- How much data can we send during one RTT?
- *E.g.*, send request, receive file



- For small transfers, latency more important, for bulk, throughput more important

Maximizing Throughput

- **Can view network as a pipe**
  - For full utilization want bytes in flight ≥ bandwidth × delay
  - But don't want to overload the network (future lectures)
- **What if protocol doesn't involve bulk transfer?**
  - Get throughput through concurrency – service multiple clients simultaneously

Using TCP/IP

- **How can applications use the network?**
- ***Sockets* API.**
  - Originally from BSD, widely implemented (*BSD, Linux, Mac OS X, Windows, …)
  - Important do know and do once
  - Higher-level APIs build on them
- **After basic setup, much like files**

System Calls

- **Problem: how to access resources other then CPU**
  - Disk, network, terminal, other processes
  - CPU prohibits instructions that would access devices
  - Only privileged OS kernel can access devices
- **Kernel supplies well-defined system call interface**
  - Applications request I/O operations through syscalls
  - Set up syscall arguments and trap to kernel
  - Kernel performs operation and returns results
- **Higher-level functions built on syscall interface**
  - `printf`, `scanf`, `gets`, all user-level code

File Descriptors

- **Most I/O in Unix done through *file descriptors***
  - Integer *handles* to per-process table in kernel
- `int open(char *path, int flags, ...);`
- **Returns file descriptor, used for all I/O to file**

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`
- **Stream sockets: bi-directional pipes**
  - With IP, gives you TCP
  - Bytes written on one end read on another
  - Reads may not return full amount requested, must re-read

Error Returns

- **What if** open **fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific type of error in global int errno
- #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

Some operations on File Descriptors

- `ssize_t read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- `ssize_t write (int fd, void* buf, int nbytes);`
  - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t offset, int whence);`
  - whence: `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
  - returns new offset, or -1 on error
- `int close (int fd);`
- `int fsync (int fd);`
  - Guarantees that file contents is stably on disk
- **See `type.c`**

```c
/* type.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

void typefile (char *filename) {
  int fd, nread;
  char buf[1024];

  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    perror (filename);
    return;
  }
  while ((nread = read (fd, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  close (fd);
}

int main (int argc, char **argv) {
  int argno;
  for (argno = 1; argno < argc; argno++)
    typefile (argv[argno]);
  exit (0);
}
```

System calls for using TCP

Client | **Server**

Server: `socket` – **make socket**

Server: `bind` – **assign address, port**

Server: `listen` – **listen for clients**

Client: `socket` – **make socket**

Client: `bind*` – **assign address**

Client: `connect` – **connect to listening socket**

Server: `accept` – **accept connection**

- **This call to bind is optional, connect can choose address & port.**

Socket Naming

- **Recall how TCP & UDP name communication endpoints**
  - IP address specifies host (128.148.32.110)
  - 16-bit port number demultiplexes within host
  - Well-known services listen on standard ports (*e.g.* ssh – 22, http – 80, mail – 25, see /etc/services for list)
  - Clients connect from arbitrary ports to well known ports
- **A connection is named by 5 components**
  - Protocol, local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP

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 addr */
};
```

- ***E.g.*** `int connect(int s, struct sockaddr* srv,`
  `                  socklen_t addrlen);`
- **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 addr */
   chars    in_zero[8];
};
```

Dealing with Address Types

- **All values in network byte order (Big Endian)**
  - `htonl()`, `htons()`: host to network, 32 and 16 bits
  - `ntohl()`, `ntohs()`: network to host, 32 and 16 bits
  - Remember to always convert!
- **All address types begin with family**
  - `sa_family` in `sockaddr` tells you actual type
- **Not all addresses are 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
  - New `sockaddr_storage` is big enough

Client Skeleton (IPv4)

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

int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (13);  /* daytime port */
sin.sin_addr.s_addr = htonl (IP_ADDRESS);
connect (s, (sockaddr *) &sin, sizeof (sin));
while ((n = read (s, buf, sizeof (buf))) > 0)
  write (1, buf, n);
```

Server Skeleton (IPv4)

```
int s = socket (AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin;
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (9999);
sin.sin_addr.s_addr = htonl (INADDR_ANY);
bind (s, (struct sockaddr *) &sin, sizeof (sin));
listen (s, 5);

for (;;) {
  socklen_t len = sizeof (sin);
  int cfd = accept (s, (struct sockaddr *) &sin, &len);
  /* cfd is new connection; you never read/write s */
  do_something_with (cfd);
  close (cfd);
}
```

Looking up a socket address with 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.brown.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! */
```

getaddrinfo() [RFC3493]

- **Protocol-independent node name to address translation**
  - Can specify port as a service name or number
  - May return multiple addresses
  - You must free the structure with freeaddrinfo
- **Other useful functions to know about**
  - getnameinfo – Lookup hostname based on address
  - inet_ntop – Convert IPv4 or 6 address to printable
  - Inet_pton – Convert string to IPv4 or 6 address

A Fetch-Store Server

- **Client sends command, gets response over TCP**
- **Fetch command (“fetch\n”):**
  - Response has contents of last stored file
- **Store command (“store\n”):**
  - Server stores what it reads in file
  - Returns OK or ERROR
- **What if server or network goes down during store?**
  - Don’t say “OK” until data is safely on disk
- **See `fetch_store.c`**

EOF in more detail

- **What happens at end of store?**
  - Server receives EOF, renames file, responds OK
  - Client reads OK, *after* sending EOF: didn't close fd
- `int shutdown(int fd, int how);`
  - Shuts down a socket w/o closing file descriptor
  - how: 0 = read, 1 = write, 2 = both
  - Note: applies to *socket*, not descriptor, so copies of descriptor (through fork or dup affected)
  - Note 2: with TCP, can't detect if other side shuts for reading

Using UDP

- **Call socket with SOCK_DGRAM, bind as before**
- **New calls for sending/receiving individual packets**
  - `sendto(int s, const void *msg, int len, int flags, const struct sockaddr *to, socklen t tolen);`
  - `recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, socklen t *fromlen);`
  - Must send/get peer address with each packet
- **Example:** `udpecho.c`
- **Can use UDP in connected mode (Why?)**
  - connect assigns remote address
  - `send`/`recv` syscalls, like `sendto`/`recvfrom` w/o last two arguments

Uses of UDP Connected Sockets

- **Kernel demultiplexes packets based on port**
  - Can have different processes getting UDP packets from different peers
- **Feedback based on ICMP messages (future lecture)**
  - Say no process has bound UDP port you sent packet to
  - Server sends port unreachable message, but you will only receive it when using connected socket

Creating/Monitoring Processes

- `pid_t fork(void);`
  - Create new process that is exact copy of current one
  - Returns twice!
  - In parent: process ID of new process
  - In child: 0
- `pid_t waitpid(pid_t pid, int *stat, int opt);`
  - pid – process to wait for, or -1 if any
  - stat – will contain status of child
  - opt – usually 0 or `WNOHANG`

Fork example

```
switch (pid = fork ()) {
   case -1:
     perror ("fork");
     break;
   case 0:
     doexec ();
     break;
   default:
     waitpid (pid, NULL, 0);
     break;
```

Deleting Processes

- `void exit(int status);`
  - Current process ceases to exist
  - Status shows up on waitpid (shifted)
  - By convention, status of 0 is success, non-zero error
- `int kill (int pid, int sig);`
  - Sends signal sig to process pid
  - SIGTERM most common sig, kills process by default (but application can catch it for "cleanup")
  - SIGKILL stronger, always kills

Serving Multiple Clients

- **A server may block when talking to a client**
  - Read or write of a socket connected to a slow client can block
  - Server may be busy with CPU
  - Server might be blocked waiting for disk I/O
- **Concurrency through multiple processes**
  - Accept, fork, close in parent; child services request
- **Advantages of one process per client**
  - Don't block on slow clients
  - May use multiple cores
  - Can keep disk queues full for disk-heavy workloads

Threads

- **One process per client has disadvantages:**
  - High overhead – fork + exit ~100µsec
  - Hard to share state across clients
  - Maximum number of processes limited
- **Can use threads for concurrency**
  - Data races and deadlocks make programming tricky
  - Must allocate one stack per request
  - Many thread implementations block on some I/O or have heavy thread-switch overhead

Rough equivalents to `fork()`, `waitpid()`, `exit()`, `kill()`, plus locking primitives.

Non-blocking I/O

- **fcntl sets O_NONBLOCK flag on descriptor**

```
int n;
if ((n = fcntl(s, F_GETFL)) >= 0)
  fcntl(s, F_SETFL, n|O_NONBLOCK);
```

- **Non-blocking semantics of system calls:**
  - read immediately returns -1 with errno EAGAIN if no data
  - write may not write all data, or may return EAGAIN
  - connect may fail with EINPROGRESS (or may succeed, or may fail with a real error like ECONNREFUSED)
  - accept may fail with EAGAIN or EWOULDBLOCK if no connections present to be accepted

How do you know when to read/write?

```
struct timeval {
  long    tv_sec;          /* seconds */
  long    tv_usec;         /* and microseconds */
};

int select (int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);
FD_ZERO(&fdset);
```

- **Entire program runs in an *event loop***

Event-driven servers

- **Quite different from processes/threads**
  - Race conditions, deadlocks rare
  - Often more efficient
- **But…**
  - Unusual programming model
  - Sometimes difficult to avoid blocking
  - Scaling to more CPUs is more complex

Coming Up

- **Next class: Physical Layer**
- **Fri 03: Snowcast milestones**