Administrivia

- **Responses should be formal, paragraph form**
- **Writing practice @ Princeton writing center**
  - Technical writing classes (Y2—)
  - Individual appointments to review writing
  - http://web.princeton.edu/sites/writing/
- **Q: 2 papers on 1 day, or 1 paper over 2 days?**
- **Start creating project teams: 2-3 people / team**

Sockets: Communication between machines

- **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 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.

Client interface

```
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));
```

Server interface

```
struct sockaddr_in sin;
int s = socket (AF_INET, SOCK_STREAM, 0);
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);
}
```

Using UDP

- **Call** `socket` **with** `SOCK_DGRAM`**,** `bind` **as before**
- **New system calls for sending individual packets**
  - ```
    int sendto(int s, const void *msg, int len, int flags,
        const struct sockaddr *to, socklen_t tolen);
    ```
  - ```
    int 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`

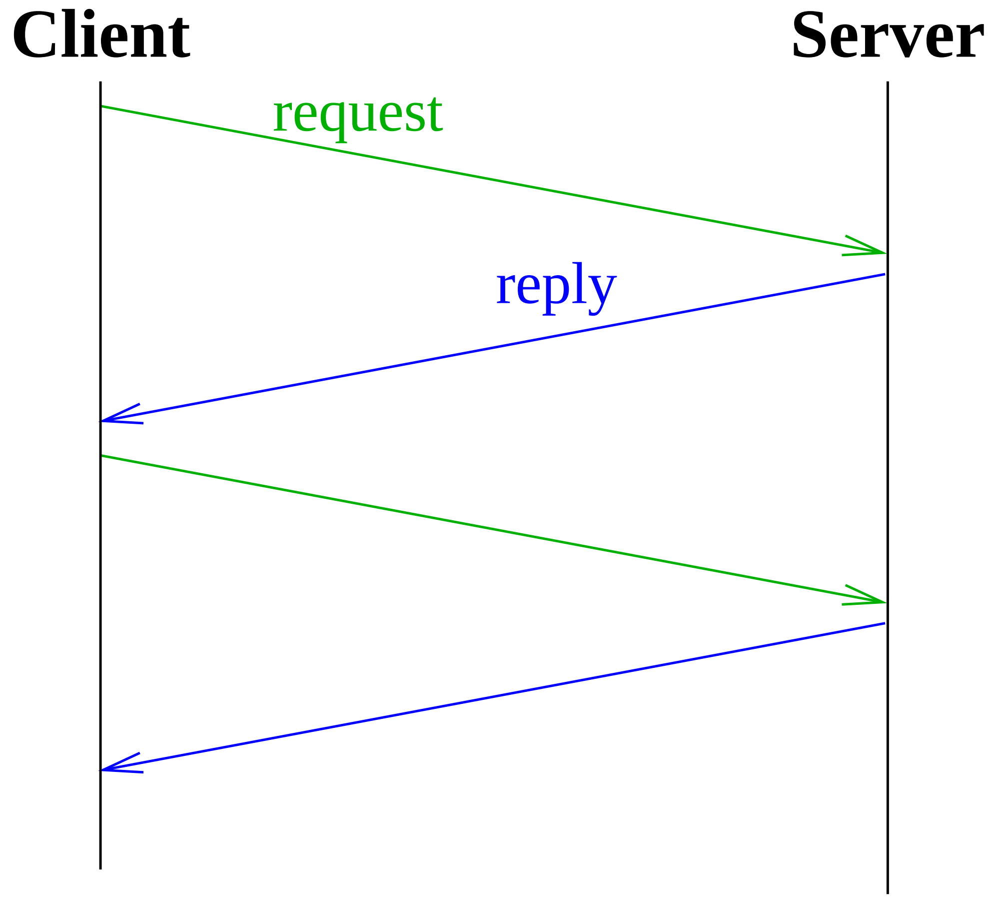
Using UDP

- `bind`: **Kernel demultplexes packets based on port**
  - So can have different processes getting UDP packets from different peers
  - For security, ports < 1024 usually can't be bound
- **Can use UDP in connected mode (Why?)**
  - `connect` assigns remote address
  - `send`/`recv` syscalls, like `sendto`/`recvfrom` w/o last 2 args

Performance definitions

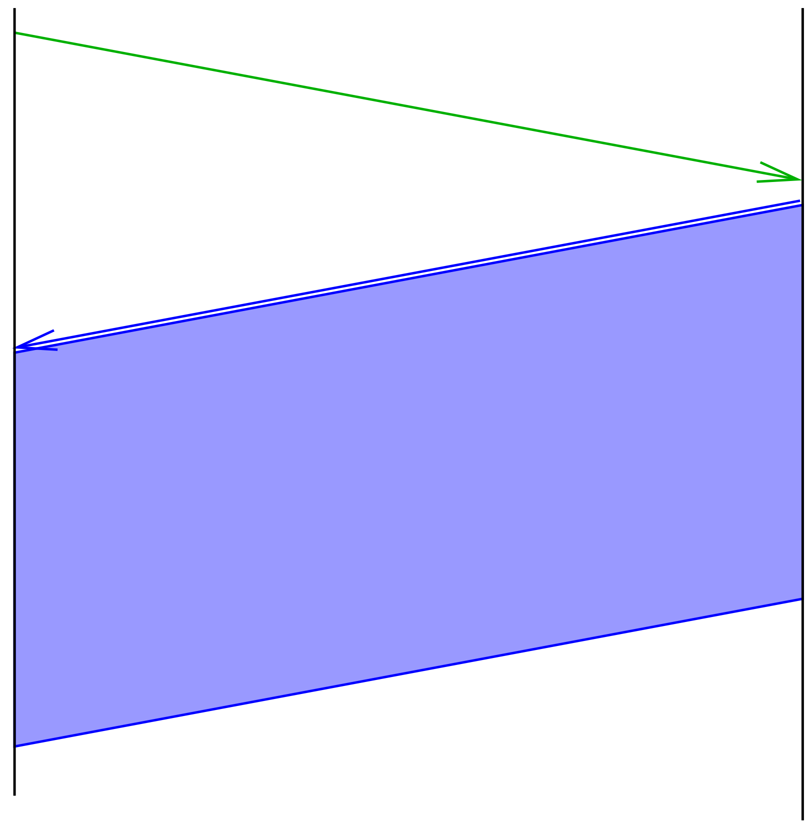
- **Bandwidth – Number of bits/time you can transmit**
  - Improves with technology
- **Latency – How long for message to cross network**
  - Propagation + Transmit + Queue
  - We are stuck with speed of light…
    10s of milliseconds to cross country
- **Throughput – TransferSize/Latency**
- **Jitter – Variation in latency**
- **What matters most for your application?**

Small request/reply protocol



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

Large reply protocol



- For bulk tranfer, throughput is most important

Bandwidth-delay

- **Can view network as a pipe**
  - For full utilization want bytes in flight $\geq \textsf{bandwidth} \times \textsf{delay}$
  - But shouldn't overload the network (congestion control)
- **What if protocol doesn't involve bulk transfer?**
  - Get throughput through concurrency—service multiple clients simultaneously

Traditional fork-based servers

- **When is a server not transmitting data**
  - Read or write of a socket connected to slow client can block
  - Server may be busy with CPU (e.g., computing response)
  - Server might be blocked waiting for disk I/O
- **Concurrency through multiple processes (MP)**
  - Accept, fork, close in parent; child services request
- **Advantages of one process per client**
  - Don't block on slow clients
  - May scale to multiprocessors if CPU intensive
  - For disk-heavy servers, keeps disk queues full
    (similarly get better scheduling & utilization of disk)

Other methods for concurrency

- **One process per client has disadvantages:**
  - High overhead – fork+exit $\sim$ 100 μsec
  - Hard to share state across clients
  - Maximum number of processes limited
- **Concurrency through threads (MT)**
  - Data races and deadlock make programming tricky
  - Must allocate one stack per request
  - Many thread implementations block on some I/O or have heavy thread-switch overhead
- **Non-blocking read/write calls (SPED)**
  - Unusual programming model

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 real error like `ECONNREFUSED`)
  - `accept` may fail with `EAGAIN` if no pending connections

How do you know when to read/write?

```
int select (int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
```

```
FD_ZERO (&fdset);       // initialize fdset
FD_SET  (fd, &fdset);   // add fd to fd list to watch
FD_ISSET(fd, &fdset);   // if set, read(fd) won't block
FD_CLR  (fd, &fdset);   // remove fd from fd list
```

Using async I/O in libasync

- **Event harness controls `select`, not programmer**
- **Programmer registers events with harness**
- **Callbacks (function pointers) triggered when event fires, e.g.,**
  - File descriptor is ready for reading/writing: `fdcb`
  - Timer completes: `delaycb`
  - Process receives signal: `sigcb`

Example: File-descriptor callbacks

- `void fdcb (int socket, char op, callback cb);`
  - `op`: `selread` or `selwrite`
- **If select on read, callback `cb` triggered when:**
  - Data is available on socket to be read
  - EOF received (read returns 0)
  - Non-transient error on socket (i.e., not `EAGAIN`)

Creating callbacks

- **Need to “save” state for event triggering**
- **Create heap-allocated object**
  - Function pointer to be triggered
  - Existing state saved in heap before creating callback
  - Return values to be added by triggering function

Function currying with `wrap`

```
R func (A, B) { ... }



callback<R, A, B> cb = wrap (func);
(*cb) (A, B);

callback<R, B> cb    = wrap (func, A);
(*cb) (B);

callback<R> cb       = wrap (func, A, B);
(*cb) ();
```

Code before "stack ripping"

```
int query_and_resp (sockaddr_in &sin) {
   int nread;
   int fd = socket (AF_INET, SOCK_STREAM, 0);

   if (connect (fd, (sockaddr *) &sin, sizeof (sin)) == 0)
      if (write (fd, req, sizeof (req)) >= 0)
         while ((nread = read (fd, resp, sizeof (resp))) > 0)
            // handle input of length nread
            if (nread == 0)
               return 0;
   return -1;
}
```

Code after “stack ripping”

```
void query_and_resp (sockaddr_in &sin) {
   int fd = socket (AF_INET, SOCK_STREAM, 0);
   callback<bool> cb = wrap (query_and_resp_2, fd);
   connect_ev (fd, (sockaddr *) &sin, sizeof (sin), cb);
}

void query_and_resp_2 (int fd, bool result) {
   if (result)
      fdcb (fd, selwrite, wrap (query_and_resp_3, fd));
}
```

Code after “stack ripping” (2)

```
void query_and_resp_3 (int fd) {
   fdcb (fd, selwrite, NULL);
   if (write (fd, req, sizeof (req)) >= 0)
      fdcb (fd, selread, wrap (query_and_resp_4, fd));
}

void query_and_resp_4 (int fd) {
   int nread = read (s, resp, sizeof (resp));
   if (nread > 0)
     // handle input of length nread
  else
     fdcb (fd, selread, NULL);
}
```

Return result in stack ripping

```
query_and_resp (wrap (query_and_resp_resp));
query_and_resp_resp (int result);

void query_and_resp   (sockaddr_in &sin, callback<int> cb);
void query_and_resp_2 (int fd, callback<int> cb, bool result);
void query_and_resp_3 (int fd, callback<int> cb);
void query_and_resp_4 (int fd, callback<int> cb) {
   int nread = read (s, resp, sizeof (resp));
   if (nread > 0)
      // handle input of length nread
   else {
      fdcb (fd, selread, NULL);
      (*cb) (((nread < 0) ? -1 : 0));
   }
}
```