TCP/IP - Socket Programming

jrb@socket.to.me

sockets - overview

- sockets
- simple client - server model
  - look at tcpclient/tcpserver.c
  - look at udpclient/udpserver.c
  - tcp/udp contrasts
- “normal” master/slave setup for TCP
- inetd on UNIX - mother server
- some details - **there are more**...

sockets

- in BSD world since early 80’s, 4.2 BSD
- client/server model
- “like” unix file i/o up to a point, can be redirected to stdin/stdout/stderr (on unix)
- sockets are dominant tcp/ip application API
  - other API is System V TLI (OSI-based)
  - winsock - windows variations on sockets
    - sockets in windows event-driven framework

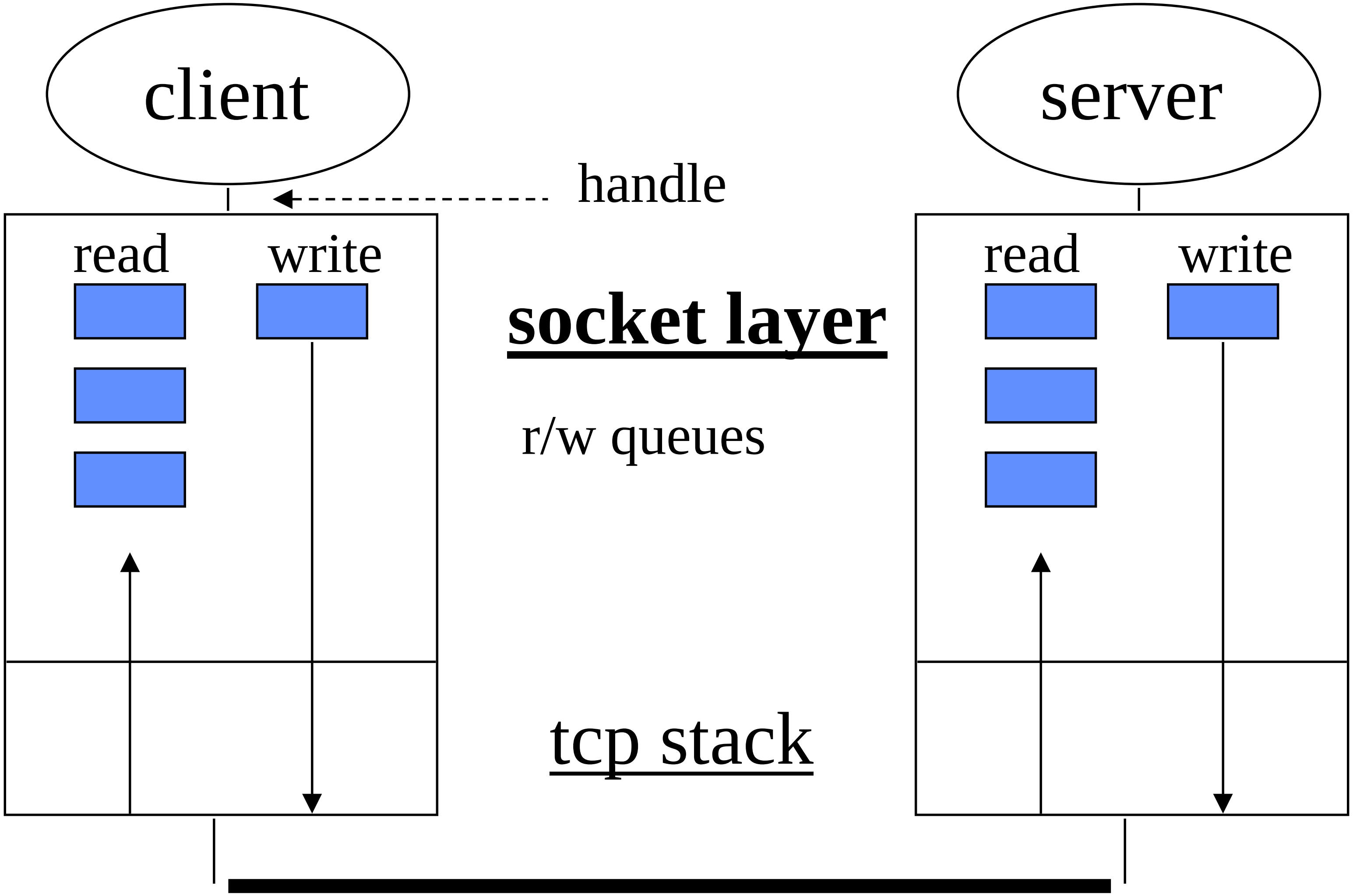
sockets

- basic definition - **“endpoint of communication”**
- allows connected streams (TCP) or discrete messages (UDP) between processes on same machine, cross network
- in o.s., really read/write data queues + TCP has connection Queue (server side)
- talk to “socket” with handle/sock descriptor

kinds of sockets

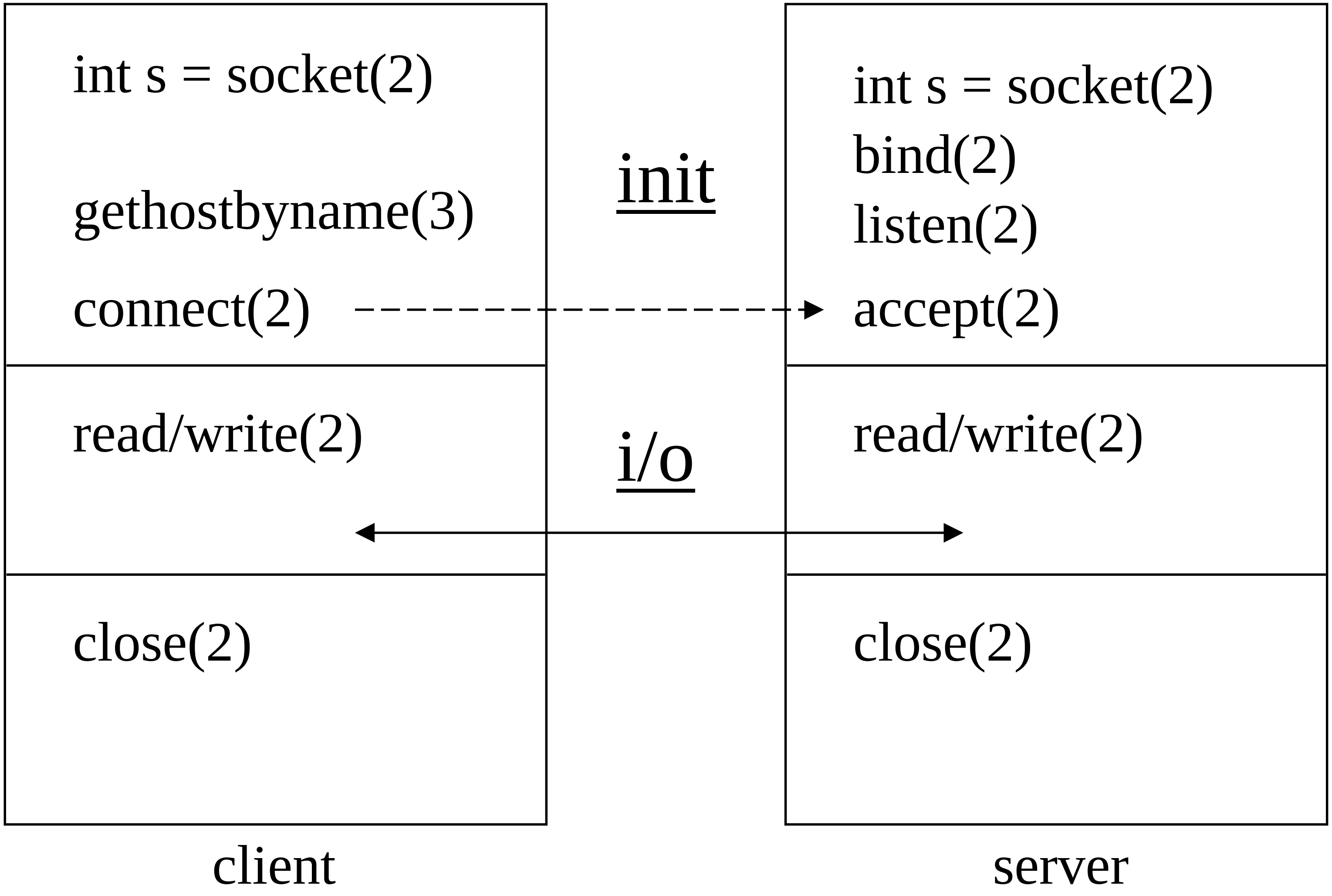
- acc. to address family; i.e. how does addressing work
- IP address family -> **IP addr, tcp/udp port**
- traditional BSD families
  - TCP/IP (AF_INET; i.e., Internet)
    - TCP/UDP/”raw” (talk to IP)
  - UNIX (intra-machine, pipes)
  - XNS, and even
  - APPLETALK, DECNET, IPX ...

sockets



client
server
handle
read
write
read
write
socket layer
r/w queues
tcp stack

syscalls - TCP client/simple test server



socket(2) syscall

- *int s = socket(family, socktype, protocol);*
  - family = AF_INET, AF_APPLETALK, etc.
  - socktype = SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
  - protocol = 0, TCP_PROTO, IP_PROTO
- example - TCP socket:
  s = socket(AF_INET, SOCK_STREAM, 0);
- **used by both client/server**

gethostbyname(3) - client

- *struct hostent h* = gethostbyname("sirius.cs.pdx.edu");*
- kernel(2) calls take ip addresses via struct sockaddr_in pointers, not DNS names
- maps DNS name to ip address SOMEHOW
  - /etc/hosts
  - DNS servers
  - yellow pages (NIS)
  - SOMEHOW - OS specific

connect(2) - client

- *rc = connect(sd, struct sockaddr *sa, len);*
- **client connects to server that can accept**
- normally TCP, but UDP rarely might use
- client must fill in server port, ip address
- TCP will attempt to connect to remote machine
- client side TCP has client TCP port - **implicit bind**

TCP/UDP ports

- distinct 64k port spaces
- client has port, server has port
- o.s. may typically allocate client port dynamically
- server SETS port, as “well-known” number; i.e., client sends packets to that port
- server port == which service (telnet/ftp/web)

bind(2) - set tcp/udp port (server)

- *int rc = bind(sock, struct sockaddr *sa, len);*
  - sock - valid sd
  - sa - struct sockaddr_in (next slide)
    port value goes in here
  - len - sizeof struct sockaddr_in data storage
- **server sets well-known TCP/UDP port**
- client rarely sets client port with bind
- if port == 0, kernel chooses for you

sockaddr structure

- sockaddr is generic structure,
  - struct sockaddr_in is instance of it for INET
  - struct sockaddr_un for UNIX sockets
- used in bind, connect, accept, sendto, recvfrom calls when ip/port # needs to be passed to/from kernel
- ip addr/port # are in NETWORK byte order

sockaddr_in - address structure

```
struct sockaddr_in {
        short sin_family;  /* AF_INET *;
        u_short sin_port;
        struct in_addr sin_addr;  /* ip addr */
        char sin_zero[8];   /* pad */
}
```

```
struct in_addr {
        u_long s_addr;
}
```

listen(2) - server

- *int rc = listen(sd, 5);*
- TCP server only, NOT UDP
- has two functions:
  - 1. enables TCP state machine, can now get connection
  - 2. sets TCP socket connection queue to 5 at a time - enables concurrent connections
- accept(2) takes connection from conn. Q

accept(2)

- int csd = accept(lsd, struct sockaddr *sa, *len);
- accepts connection - paired with connect(2)
- blocks without select(2) call until connection arrives, **returns connected sd**
- now in connected state, can make read/write calls, use connected sd (not listen sd)
- returns client ip/port in sockaddr_in
- **NOTE: len is *call by value-result***

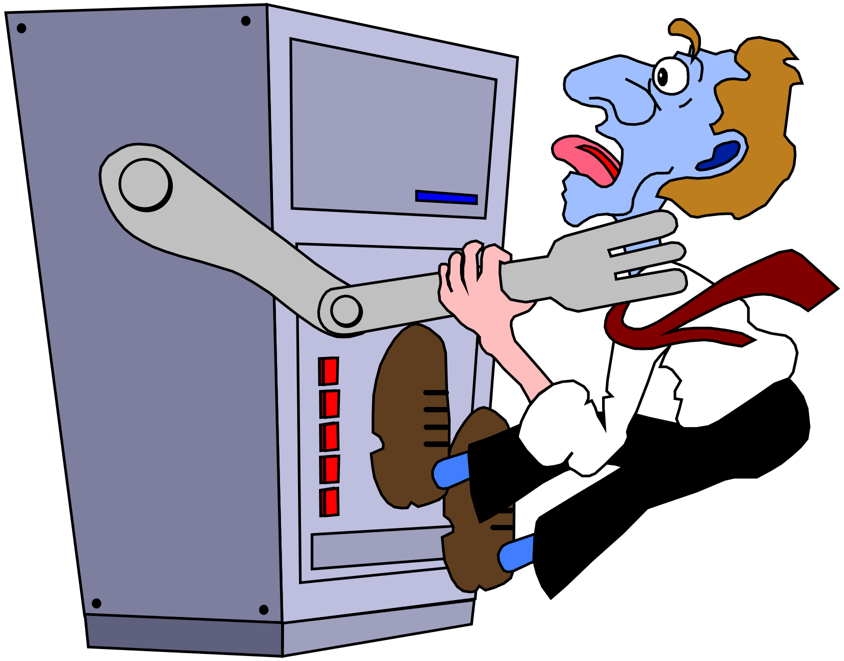
accept(2) no substitute

- what is the problem here?

```
int sock;
struct sockaddr_in recvSock;
int len = sizeof(struct sockaddr_in);
int rc = accept(sock, &recvSock, len);
```

Can you say BOOM!!!!!!...

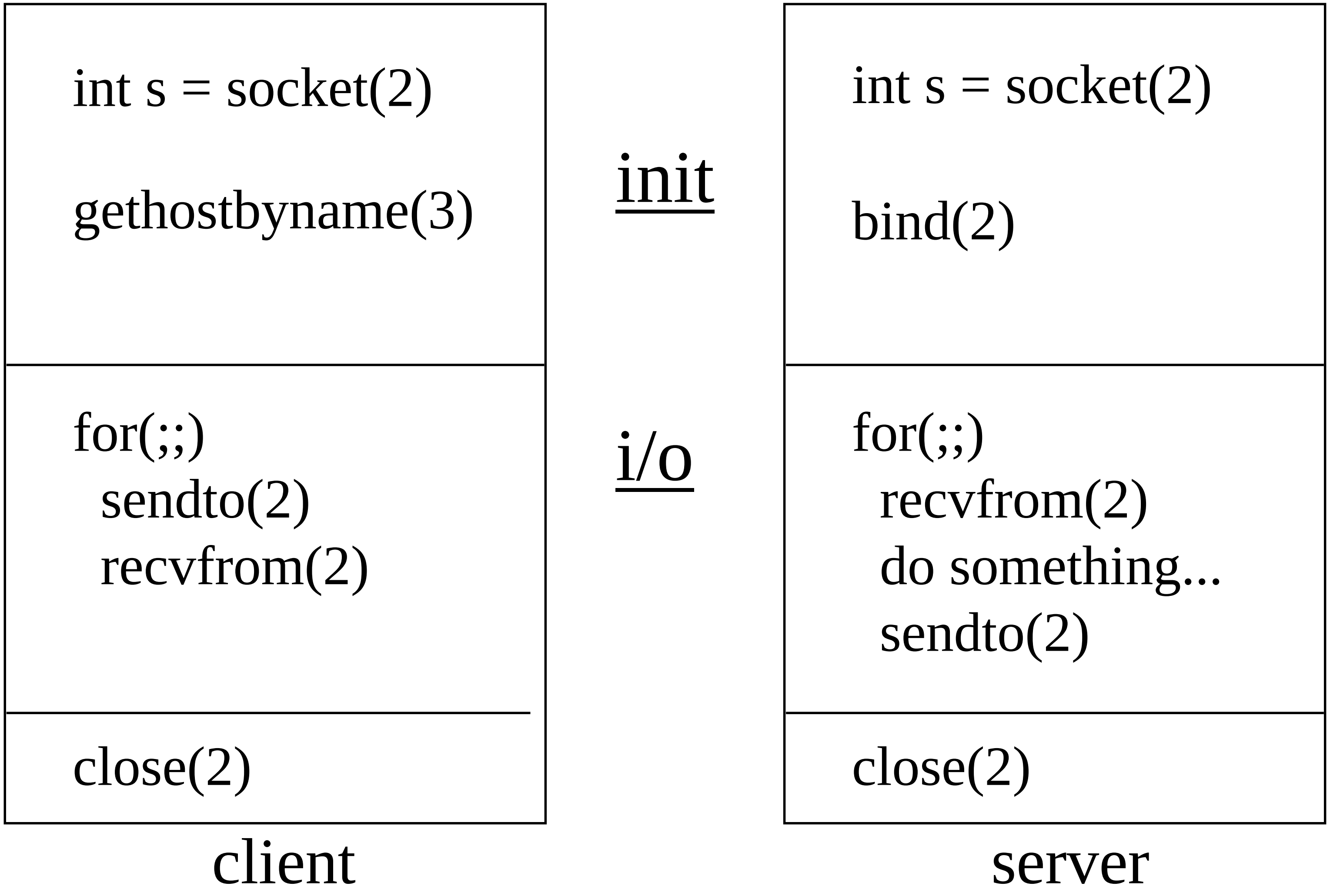
computer gets programmer’s attention



read(2) / write(2)

- with normal TCP, read may return less data than you expect. Call it in a loop.
- example: you ask for 1024, you get two 512 byte packets.
- *write* will block until your data is written - don’t need loop (unless you use non-blocking i/o)
- note: *read(fd, char *buf, int cc)*
  - **TCP addresses setup at connect time!**

syscalls - UDP client/simple test server



init

i/o

client:

```
int s = socket(2)

gethostbyname(3)

for(;;)
  sendto(2)
  recvfrom(2)

close(2)
```

server:

```
int s = socket(2)

bind(2)

for(;;)
  recvfrom(2)
  do something...
  sendto(2)

close(2)
```

udp - send/recv packets

- *int sd = socket(AF_INET, SOCK_DGRAM, 0);*
- *bind* used to set server port
- *sendto/recvfrom* have sockaddr_in parameters that must specify “peer” ip address/port #.
- *recvfrom* will tell you from whom you got the packet (ip/port), you use *sendto* to send it back
- one server may get packets from N clients
- **no idea of connection**

UDP server

- socket/bind call
- loop

  ```
  recvfrom(sd, ...&fromaddr ...);
   sendto(sd, ...&fromaddr ...);
  ```

- one server can serve packets from many clients
- TCP needs to have one server per client and must use threads/fork a process/task per connection

tcp/udp contrasts

- tcp is stream
- tcp is reliable
- tcp is point to point and “connected”
- connect/accept specify addresses at setup time, read/write don’t need addresses
- data is checksummed

- udp discrete packets
- udp is unreliable
- udp can broadcast, 1 to N or
- server can receive from many clients
- each read/write specifies address
- data MAY be csum’ed

master/slave tcp server

init

```
socket/bind/listen
signal(SIGCHLD, reapem);
```

fork slave
on accept

```
for (;;)

        nsd = accept(lsd, ...)
        if (fork() == 0) {
                read/write(nsd, ...);
                close(nsd);
        }
        close(nsd);
```

slave does
i/o

cleanup
Zombies

```
reapem() - signal handler
```

master/slave - master signal handler

```
init:   int reapem();
        signal(SIGCHLD, reapem)

signal handler:
        reapem() {
                for(;;) {
                        rc = waitpid(,WNOHANG,);
                        if ( rc <= 0)
                                return;
                }
        }
```

inetd - unix mother daemon

- per well-known port protocol servers ate up too many o.s. resources
- combined into one TITANIC mother daemon - only one thread at rest
- “listens” at tcp/udp ports - spawns stub server to do work
- see /etc/inetd.conf for setup
- uses *select(2)* mechanism

BSD/UNIX select(2) call

- *nohits = select(nofds, readmask, writemask, exceptionmask, timeout);*
- ***select functions**:*
  - allows callers to detect i/o to be read on > 1 socket or char device descriptor at a time
  - allows callers to detect TCP connection (so you can call accept) - inetd does this
  - handles TCP “out of band data”
  - can do timed poll or block if time == 0

some socket details:

- inet_addr(3) routines - manipulate ip addrs

  example: convert string to ip addr

  *struct in_addr * = inet_addr("1.2.3.4");*

  *char *inet_ntoa(struct in_addr inaddr);*

- BSD "database" routines:
  – /etc/hosts - *gethostbyname(3), gethostbyaddr(3)*
  – /etc/services - *getservbyname(3)*
  – /etc/protocols - *getprotobyname(3)*

BSD oft-used TCP/IP files

- /etc/hosts - host/ip pairs, they don’t all fit
- /etc/services - TCP/UDP well known ports
  - 9 - discard port
- /etc/resolv.conf - DNS servers
- /etc/protocols - proto name to number mapping (protocols above IP)
- /etc/inetd.conf - servers inetd can run

details, the end:

- byte-order routines: BIG-ENDIAN rules
  - sparc/68k not Intel Architecture
  - long word: *htonl(3), ntohl(3)*
  - short: *htons(3), ntohs(3)*
  - bytes - no problem
- misc. socket ops
  - *setsockopt(2), getsockopt(2)*
    - turn on UDP broadcast, multicast
    - see Stevens for details