CU CS 4995-03: Internet Systems Programming

Sockets and Basic Network Programming

Overview

- socket creation
- `select()`
- name resolution
- multiplexing

Client-server model

- *server* process listens for network requests from *clients*
- server = passive open, client = active open
- examples:
  - file server
  - print server
  - name server
  - web server
  - X window server
  - mail server

Client-server model

- UDP or TCP
- remote procedure call: transport + data *marshalling*
- spawn process (inetd) or thread for each connection request
- find services: DNS MX, DNS SRV, pre-configured, service location protocol, …

Creating a socket

- like filedescriptor (type `int`)
- domain, type and protocol:

domain: `PF_INET`, `PF_UNIX`, ...

	datagram	`SOCK_DGRAM`	UDP
type:	reliable, sequenced stream	`SOCK_STREAM`	TCP
	raw	`SOCK_RAW`	IP

protocol: usually 0 = default protocol for type

Creating a socket

`socket`(*family*, *type*, *protocol*);

```
#include <sys/types.h>
#include <sys/socket.h>
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0)
  perror("socket");
```

Endpoint addresses

Each socket association has five components ("5-tuple"):

protocol: `socket`

local (source) address, port: `bind` (address may be wild-carded)

remote (destination) address, port: `connect`, `sendto`

Incoming packets match most specific tuple first.

Socket addresses

```
struct sockaddr {  /* sys/socket.h */
  sa_family_t sa_family;      /* address family */
  char sa_data[14];           /* addressing information */
}
struct sin_addr {  /* netinet/in.h */
  u_long s_addr;              /* 32-bit host address */
}
struct sockaddr_in {
  short sin_family;           /* AF_INET */
  u_short sin_port;           /* network byte order */
  struct sin_addr sin_addr;   /* network address */
}
```

Address conversion for IPv4

internally: 32-bit integers for IPv4
text: “dotted quad” (decimal bytes), e.g., 128.59.16.1

host name to numeric: `gethostbyname()`, resolver

dotted quad to numeric: `inet_addr()`

numeric to dotted quad: `inet_ntoa()`

Aside: DNS

DNS is more than just "www.yahoo.com $\longrightarrow$ 204.71.200.74": indexed, hierarchical database, with *resource records*

A: address record

CNAME: canonical name mapping; "aliases"

MX: mail exchange

SRV: services

HINFO: hardware and operating system

NAPTR: regular expression translation

SOA: "start of authority"

```
$ host -t hinfo bart.cs.columbia.edu
bart.cs.columbia.edu HINFO "Ultra10" "Unix"
```

DNS examples

```
$ host www.cs.columbia.edu
www.cs.columbia.edu      CNAME   cosmos.cs.columbia.edu
cosmos.cs.columbia.edu   A       128.59.16.149

$ host www.columbia.edu
www.columbia.edu         A       128.59.35.59
www.columbia.edu         A       128.59.35.60
(ten more)

$ host www.columbia.edu
www.columbia.edu         A       128.59.35.56
www.columbia.edu         A       128.59.35.57
```

DNS MX records

- short email address
- independent of particular host
- fall-back to off-side delivery, several servers

```
$ host -t mx cs.columbia.edu
cs.columbia.edu  MX 200 tune.cs.columbia.edu
cs.columbia.edu  MX 300 smtp.columbia.edu
cs.columbia.edu  MX 400 pipeline.com
cs.columbia.edu  MX 100 cs.columbia.edu
```

DNS SRV records

- where is the “foo” service for domain aol.com?
- avoid ftp.columbia.edu, www.cs.columbia.edu, etc.
- load balancing (“weights”), priority
- SRV records (RFC 2052)

```
$ host -t srv sip.udp.cs.columbia.edu
sip.udp.cs.columbia.edu SRV 0 200 5060 erlang.cs.columbia.edu
                            0 500 5060 erlang.cs.columbia.edu
```

Name resolution

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
...
  struct hostent *hp;
  struct sockaddr_in sin;

  if ((hp = gethostbyname(argv[1])) == NULL) {
    fprintf(stderr, "no such host %s\n", argv[1]);
    exit(1);
  }
```

gethostbyname

```
memset(&sin, sizeof(sin), 0);
sin.sin_family = hp->h_addrtype;
memcpy(&sin.sin_addr, hp->h_addr, hp->h_length);
printf("address %s\n", inet_ntoa(sin.sin_addr));
```

`h_addr_list` for all addresses

resolver: host names $\longrightarrow$ addresses

- `gethostbyname` only works with A records
- `res_search` allows default names (non-FQDN)
- link with `-lresolv`

```
#include <arpa/nameser.h>
#include <resolv.h>

res_init();
res_search(char *dname, int class, int type,
uchar_t *answer, int anslen);
res_query(char *domain, C_IN, T_CNAME,
uchar_t *answer, int anslen);
```

resparse

- `res_search` only returns raw resource record
- use `res_parse` for parsing:

```
char msg[...];
s_rr *rr;
res_search(domain, C_IN, T_CNAME, msg, sizeof (msg));
res = res_parse(msg);
rr = res->answer;
if ((rr[i]->type == T_CNAME) && (rr[i]->class == C_IN)) {
  printf("%s\n", rr[i]->rdata.string");
}
```

Data representation

- integers:

little endian	least significant byte first	DEC, Intel
big endian	most significant byte first	Sun, SGI, HP, …

- *network byte order* = big endian
- `m = ntohl(m)`: network-to-host byte order, 32 bit
- `m = htonl(m)`: host-to-network byte order
- `ntohs`, `htons`: short (16 bit)

Data representation

Other problems:

- characters (ASCII, 8859-x, multi-byte)
- floating point, structured data, unions, …
- ➠ XDR, ASN.1 (canonical representation)

Binding the local address

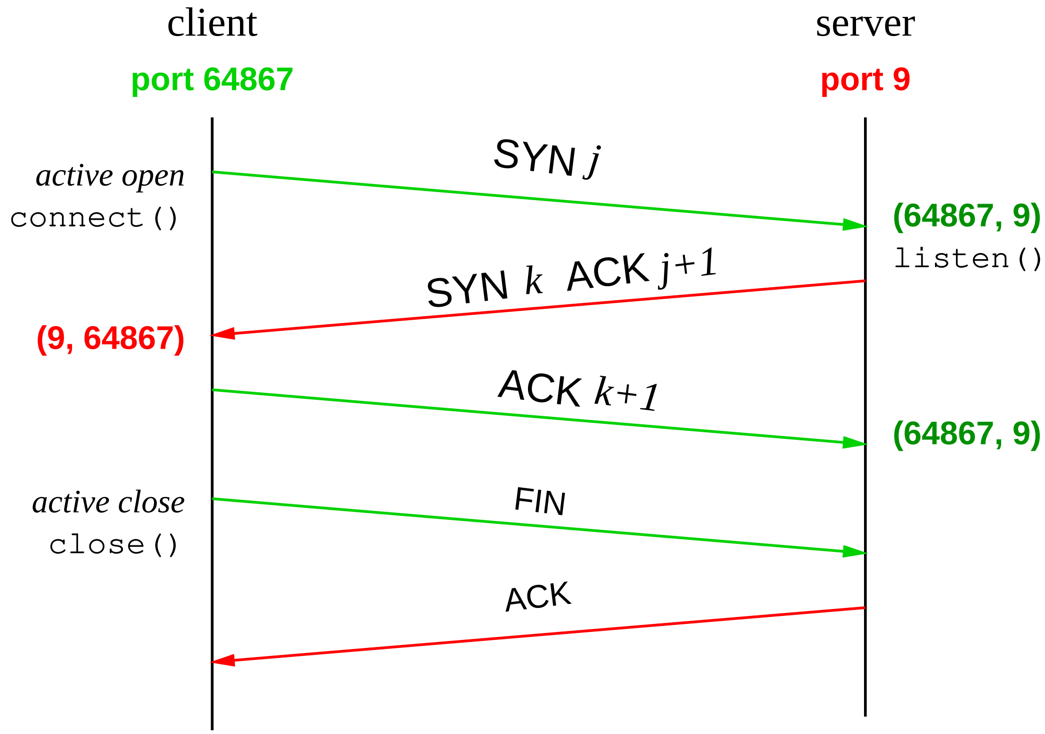
```
int bind(int s, struct sockaddr *addr, int addresslen);
```

- `addr` typically `INADDR_ANY`; exception: multi-homed hosts (virtual servers)
- only one socket per port and protocol (almost)
- optional for clients ➠ system chooses random local port

Binding the local address

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int s;
struct sockaddr_in sin;

if ((s = socket(PF_INET, SOCK_DGRAM, 0)) < 0) { /* error */ }
memset((char *)&sin, 0, sizeof(sin));
sin.sin_family      = AF_INET;
sin.sin_port        = htons(6000); /* 0: let the system choose */
sin.sin_addr.s_addr = INADDR_ANY;  /* allow any interface */
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) { /* */ }
```

TCP Review



SYN retransmitted with intervals 3, 6, 12 s
(ndd `tcp_rexmit_interval_initial`)

TCP TIME_WAIT and connection re-use

- server waits for 2 MSL (max. segment lifetime) for old connection
- if not properly terminated: `bind` → `EADDRINUSE`
- `setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)&opt, sizeof(opt));` before `bind()`
- allows re-use of port already in use (with different remote port); e.g., child process for server socket

Other Socket Options

`SO_ERROR`	both	get error status
`SO_KEEPALIVE`	TCP	send periodic keep-alives (2 hrs.)
`SO_LINGER`	TCP	immediate `close()`?
`SO_OOBINLINE`	TCP	get urgent data in line
`SO_RCVLOWAT`	both	`select` threshold for socket
`SO_RCVTIMEO`	?	socket receive time-out (`timeval`) (Solaris?)
`SO_DGRAM_ERIND`	UDP	error indication on unconnected socket

Graceful close

```
struct linger {
  int l_onoff;    /* 0: off; !=: on */
  int l_linger;   /* seconds */
}
```

l_onoff	l_linger	behavior
0	ignored	`close` returns immediately
1	0	abort & flush, send `RST`
1	> 0	sleep until data ack'ed

Even after lingering, server application may crash before it has read the data.

`shutdown`

```
shutdown(int s, int how);
```

- `how = 0` ⟶ no more receives
- `how = 1` ⟶ no more sending
- `how = 2` ⟶ neither = `close();`

tcpdump

```
$ tcpdump -n -N port 9
3.882020 erlang.64863 > bart.9: S
  1546053952:1546053952(0)
  win 33580 <mss 1460> (DF)
3.882046 bart.9 > erlang.64863: S
  3368235852:3368235852(0) ack 1546053953
  win 33580 <mss 1460> (DF)
3.882346 erlang.64863 > bart.9: . ack 1 win 33580 (DF)
3.883755 erlang.64863 > bart.9: P 1:6(5) ack 1 win 33580 (DF)
3.883861 bart.9 > erlang.64863: . ack 6 win 33580 (DF)
3.884200 erlang.64863 > bart.9: FP 6:11(5) ack 1 win 33580 (DF)
3.884263 bart.9 > erlang.64863: . ack 12 win 33580 (DF)
3.888872 bart.9 > erlang.64863: F 1:1(0) ack 12 win 33580 (DF)
3.889149 erlang.64863 > bart.9: . ack 2 win 33580 (DF)
```

The “real” TCP Packets

tcpdump normally shows relative sequence numbers, unless `-N` flag is specified:

```
$ tcpdump -n -N -S -t -v port 9
.141.64867 > .191.9: S 1604639280:1604639280(0)
   win 33580 <mss 1460> (DF)

.191.9 > .141.64867: S 3426570476:3426570476(0)
   ack 1604639281 win 33580 <mss 1460> (DF)

.141.64867 > .191.9: . ack 3426570477 win 33580 (DF)

.141.64867 > .191.9: P 1604639281:1604639286(5)
   ack 3426570477 win 33580 (DF)

.191.9 > .141.64867: . ack 1604639286 win 33580 (DF)
```

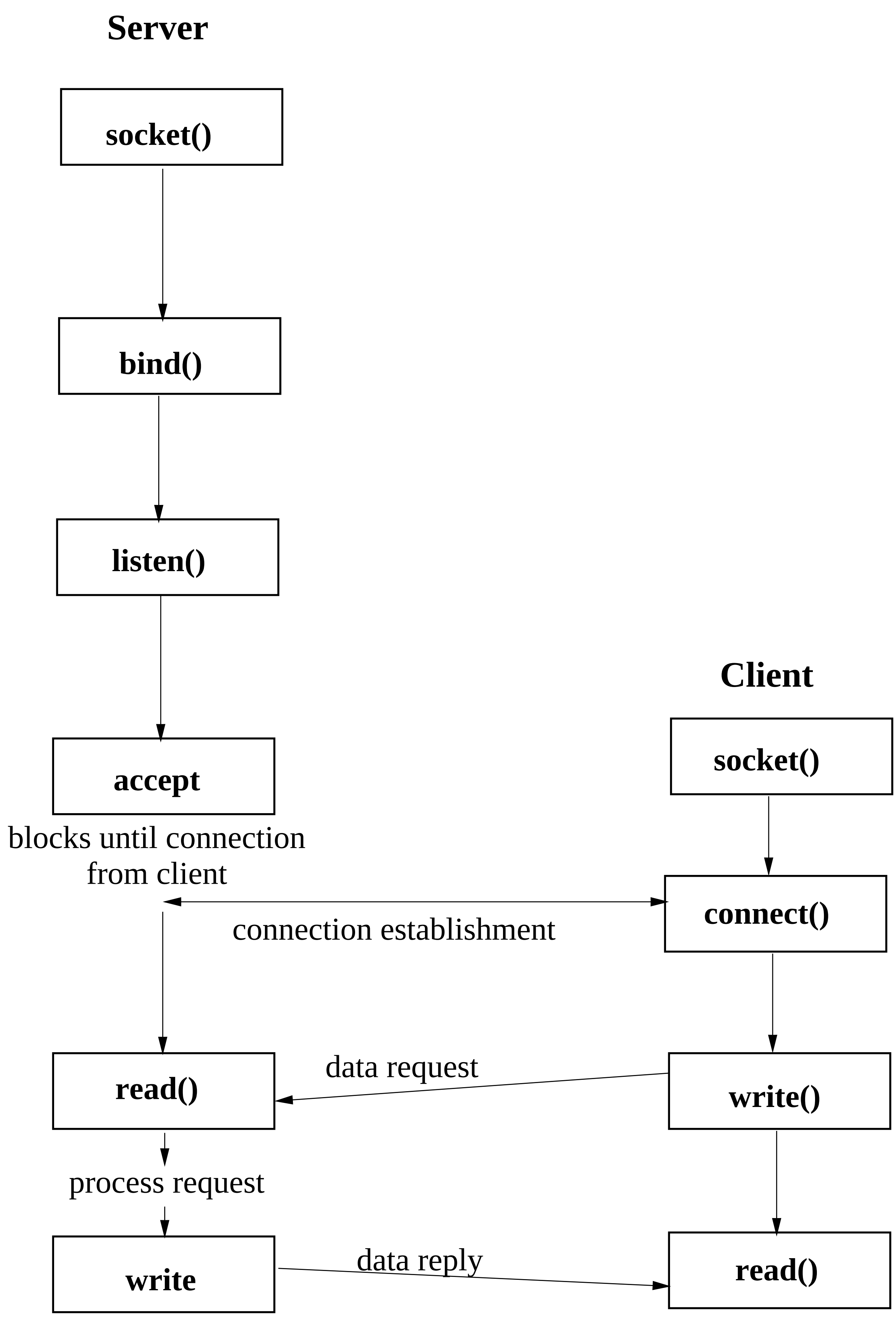
The “real” TCP packets, cont’d.

```
.141.64867 > .191.9: FP 1604639286:1604639291(5)
   ack 3426570477 win 33580 (DF)

.191.9 > .141.64867: . ack 1604639292 win 33580 (DF)

.191.9 > .141.64867: F 3426570477:3426570477(0)
   ack 1604639292 win 33580 (DF)

.141.64867 > .191.9: . ack 3426570478 win 33580 (DF)

.141.64867 > .191.9: FP 1604639286:1604639291(5)
   ack 3426570477 win 33580 (DF)

.191.9 > .141.64867: . ack 1604639292 win 33580 (DF)

.191.9 > .141.64867: F 3426570477:3426570477(0)
   ack 1604639292 win 33580 (DF)
```

```
.141.64867 > .191.9: . ack 3426570478 win 33580 (DF)
```

TCP "push"

- `PSH` flag in TCP header
- "please deliver immediately"
- no consequence for Unix – always does this
- cannot be set explicitly, but is always set after each buffer


Server
socket()
bind()
listen()
accept
blocks until connection
from client
connection establishment
read()
process request
write
data request
data reply
Client
socket()
connect()
write()
read()

Connecting to a server: `connect()`

```
int connect(int s, struct sockaddr *name, int namelen);
```

- client issues `connect()` to
  - establish remote address, port (stream, datagram sockets)
  - establish connection (stream socket)
- fails: host not listening to port, timeout

Establish connection queue: `listen()`

```
int listen(int s, int backlog)
```

- only used for stream sockets
- socket being listened to can’t be used for client
- `listen()` does **not** wait for connections, but needed to receive connection attempts
- completes three-way handshake (3WH)

`listen`

- allows *backlog* pending connection requests (SYN, completed 3WH) while waiting for `accept()`
- if queue full, SYN requests are silently dropped
- "SYN-flood": send fake TCP SYN requests to fill queue ⟶ cache all, drop oldest
- interpretation of *backlog* is OS-dependent; limit can be set with `ndd`:

```
$ ndd /dev/tcp tcp_conn_req_max_q
128
```

- aside: get summary of TCP status via `ndd /dev/tcp tcp_status`

Accepting connections: `accept()`

```
int accept(int s, struct sockaddr *addr, int *addrlen);
```

- by connection-oriented server, after `listen()`
- can't "preview" connection ➠ accept and close
- returns a **new** socket for data exchange with client
- optionally, returns address of client

Sending and receiving data

call	device	buffering	opt	peer?
read(), write()	any	single	N	N
readv(), writev()	any	s/g	N	N
recv(), send()	socket	single	Y	N
recvfrom(), sendto()	socket	single	Y	Y
recvmsg(), sendmsg()	socket	s/g	Y	Y

- block size read (may) $\neq$ block size written
- read from stream: may read fewer bytes than requested
- scatter/gather: read/write from/to several buffers at once

Socket programming: TCP client

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
extern struct in_addr host2ip(char *host);
int main(int argc, char *argv[])
{
  int s, n;
  struct sockaddr_in sin;
  char msg[80] = "Hello World!";

  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket"); return -1;
  }
```

Socket programming: TCP client, cont'd.

```
  sin.sin_family = AF_INET;
  sin.sin_port   = htons(atoi(argv[2]));
  sin.sin_addr   = host2ip(argv[1]);
  if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("connect"); return -1;
  }
  if (write(s, msg, strlen(msg)+1) < 0) {
    perror("write"); return -1;
  }
  if ((n = read(s, msg, sizeof(msg))) < 0) {
    perror("read"); return -1;
  }
  printf("%d bytes: %s\n", n, msg);
  if (close(s) < 0) { perror("close"); return -1; }
  return 0;
}
```

Socket programming: single-threaded

```
int server;

server = initialize();
while (1) {
  wait for request(server)
  read request
  send answer
}
```

Socket programming: TCP server

```
int main(int argc, char *argv[])
{
  int s, t;
  struct sockaddr_in sin;
  char *r;
  int sinlen;

  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket"); return -1;
  }
  sin.sin_family      = AF_INET;
  sin.sin_port        = htons(13333);
  sin.sin_addr.s_addr = INADDR_ANY;
  if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("bind"); return -1;
  }
  if (listen(s, 5) < 0) { perror("listen"); return -1; }
```

Socket programming: TCP server

```
  for (;;) {
    sinlen = sizeof(sin);
    if ((t = accept(s, (struct sockaddr *)&sin, &sinlen)) < 0) {
      perror("accept"); return -1;
    }
    r = gettime();
    if (write(t, r, strlen(r)) < 0) {perror("write"); return -1;}
    if (close(t) < 0) { perror("close"); return -1; }
  }
  if (close(s) < 0) { perror("close"); return -1; }
}
```

Socket programming: forking TCP server

```
for (;;) {
  sinlen = sizeof(sin);
  if ((t = accept(s, (struct sockaddr *)&sin, &sinlen)) < 0) {
    perror("accept"); return -1;
  }
  if (fork() == 0) { /* code for child */
    FILE *f;
    char buffer[1024];
    close(s);
    f = fdopen(t, "w");
    fprintf(f, "pid %d", getpid());
    fclose(f);
    exit(0);
  }
  close(t);
}
```

Socket programming: UDP client

```
int s, c, fromlen;
struct sockaddr_in sin, from;
char buf[1024];

if ((s = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
  perror("socket");
}

from.sin_family      = AF_INET;
from.sin_port        = 0;
from.sin_addr.s_addr = INADDR_ANY;
if (bind(s, (struct sockaddr *)&from, sizeof(from)) < 0) {
  perror("bind");
}
```

Socket programming: UDP client

```
sin.sin_family      = AF_INET;
sin.sin_port        = atoi(argv[2]);
sin.sin_addr.s_addr = inet_addr(argv[1]);
if (sendto(s, argv[3], strlen(argv[3]), 0,
  (struct sockaddr *)&sin, sizeof(sin)) < 0) {
perror("sendto");
}
while (1) {
 if ((c = recvfrom(s, buf, sizeof(buf), 0,
   (struct sockaddr *)&from, &fromlen)) < 0) {
 perror("recvfrom");
}
buf[c] = '\0'; puts(buf);
```

Event-based programs

- `read()` is blocking ➠ server only works with single socket
- need I/O multiplexing ➠ event-based programming (or threads)
- also need to handle time-outs, connection requests
- all events (mouse clicks, windows, etc.) handled by event loop

Event-based programs

do
    wait for event(s)
    handle event (hopefully short)
forever

- harder to maintain state, recursion
- alternative 1: "interrupts" (signals) ➠ called at any time
- alternative 2: threads (separate scheduling, same address space)

Multiplexing with `select()`

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

- block until >= 1 file descriptors have something to be read, written, or an exception, or timeout
- set bit mask for descriptors to watch using `FD_SET`
- returns with bits for ready descriptors set ➠ check with `FD_ISSET`
- cannot specify amount of data ready

`select()` server example

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/time.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int s = -1, t = -1, n, sinlen;
  struct sockaddr_in sin;
  char msg[80];
  fd_set rfds;
  struct timeval tv;
```

`select()` server example

```
if (argc < 2) { printf("%s port\n", argv[0]); return -1; }

if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
  perror("socket"); return -1;
}
sin.sin_family      = AF_INET;
sin.sin_port        = htons(atoi(argv[1]));
sin.sin_addr.s_addr = INADDR_ANY;
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
  perror("bind"); return -1;
}
if (listen(s, 5) < 0) { perror("listen"); return -1; }
tv.tv_sec = 10;
tv.tv_usec = 0;
```

`select()` server example

```
while (1) {
  FD_ZERO(&rfds);
  if (s > 0) FD_SET(s, &rfds);
  if (t > 0) FD_SET(t, &rfds);
  n = select(FD_SETSIZE, &rfds, NULL, NULL, &tv);
  if (n == 0) { printf("Timeout!\n"); }
  else if (n > 0) {
    if (s > 0 && FD_ISSET(s, &rfds)) {
      sinlen = sizeof(sin);
      if ((t = accept(s, (struct sockaddr *)&sin, &sinlen)) < 0) {
        perror("accept"); return -1;
      }
    }
```

`select()` server example

```
    if (t > 0 && FD_ISSET(t, &rfds)) {
      if ((n = read(t,msg,sizeof(msg))) < 0) {
        perror("read"); return -1;
      }
      if (n == 0) { close(t); t = -1; }
      else {
        if (write(t,msg,n)<0) {perror("write");return -1;}
}}}}
  if (close(t) < 0) { perror("close"); return -1; }
  if (close(s) < 0) { perror("close"); return -1; }
  return 0;
}
```

inetd

- server processes ("daemons"): `routed`, `snmpd`, `in.rlogind`, `ftpd`, `mountd`, `telnetd`, . . .
- each service, one process ➠ lots of idle processes, listening sockets
- `inetd`: Internet services daemon
- opens sockets, forks processes when request arrives
- stdin/stdout connected to network
- but: process creation overhead

OS performance issues

- high interrupt rate: 100 Mb/s, 500 byte packets ➠ 25,000 p/s
- mostly copying: packet assembly, add/remove headers, demux
- scatter-gather I/O
- cache not very helpful: see each byte once
- one copy architecture, possibly memory map

Scatter/gather I/O

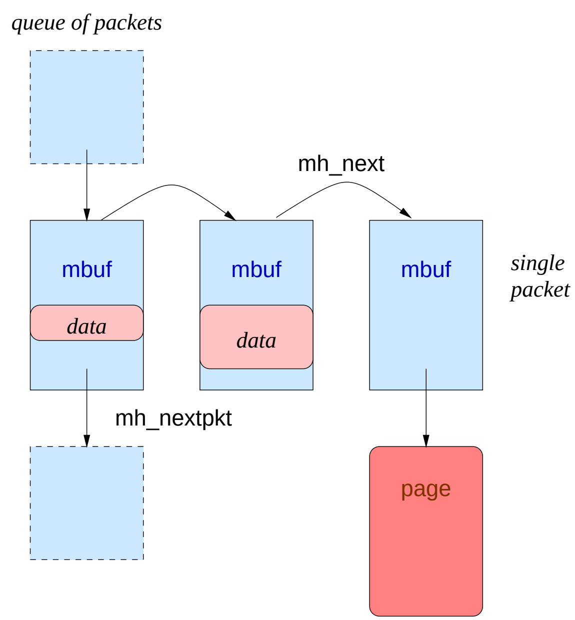
Allows to write/read to/from several buffers in one call ➠ write header and data, etc.

```
struct iovec {
  caddr_t iov_base; /* base address of buffer */
  int     iov_len;  /* length of buffer */
}
int writev(int s, struct iovec iov[], inv iovcnt);
int readv(int s, struct iovec iov[], int iovcnt);
```

- writes *iovcnt* buffers from array *iov*
- returns total number of bytes read/written
- on reading, may not fill all buffers

Unix mbuf

- used for all BSD network I/O, useful for applications
- `mbufs` are 128 bytes long
- 112 bytes data storage within mbuf *or* reference page from page pool



Unix mbuf

```
struct m_hdr {
  struct  mbuf *mh_next;     /* next buffer in chain */
  struct  mbuf *mh_nextpkt; /* next chain in queue */
  caddr_t mh_data;           /* location of data */
  int     mh_len;            /* amount of data */
  short   mh_type;           /* type of data */
  short   mh_flags;          /* flags */
};
```

IP Multicast

- groups indicated by class-D address (224.0.0.0 – 239.255.255.255): “radio frequency”
- mapped to *service*`.mcast.net` for well-known services, e.g., `sap.mcast.net`
- anybody can send data to multicast address
- senders don’t have to be members

Multicast programming

UDP, not TCP (obviously…)

```
struct sockaddr_in name;
struct ip_mreq imr;

sock = socket(AF_INET, SOCK_DGRAM, 0);
imr.imr_multiaddr.s_addr = htonl(groupaddr);
imr.imr_interface.s_addr = htonl(INADDR_ANY);
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
  &imr, sizeof(struct ip_mreq));
name.sin_addr.s_addr = htonl(groupaddr);
name.sin_port = htons(groupport);
bind(sock, &name, sizeof(name));
recv(sock, (char *)buf, sizeof(buf), 0);
```

Multicast options

IP_ADD_MEMBERSHIP	struct ip_mreq	join
IP_DROP_MEMBERSHIP	struct ip_mreq	leave
IP_MULTICAST_IF	struct in_addr	outgoing if
IP_MULTICAST_TTL	u_char	TTL [1]
IP_MULTICAST_LOOP	u_char	get own? [1]

```
unsigned char loop = 0;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP,
  &loop, sizeof(unsigned char));
```

Multicast programming

Set time-to-live:

```
unsigned char ttl = 1;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,
  sizeof(char));
```

0 for local, 1 for LAN, 255 for world.

Also: *scoped* multicast addresses for administrative scopes.

Six common socket mistakes

1. TCP stream has no record boundaries
2. don't forget address family
3. `accept` needs in/out `addrlen`
4. TCP does not check for liveness
5. `sin_addr` vs. `sin_addr.s_addr`
6. `sin.sin_port = htons(port)`

syslog

- central logging facility
- select for console, file, user, network via `/etc/syslog.conf`

```
*.err           /dev/sysmsg
daemon.debug    /var/log/daemon
*.notice        @logger.cs.columbia.edu
*.debug         root,operator
*.crit          *
```

- connect to `syslogd` server

syslog

```
#include <syslog.h>
openlog("sipd", LOG_PID, LOG_DAEMON);
syslog(LOG_ERR, "%s: %d", msg, value);
closelog();
```

LOG_PID	log PID
LOG_CONS	to console

priority: LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR, LOG_WARNING, LOG_NOTICE, LOG_NOTICE, LOG_INFO, LOG_DEBUG

`syslog` **Facilities**

`LOG_KERN`	kernel
`LOG_USER`	default facility
`LOG_MAIL`	mail system
`LOG_DAEMON`	system daemons
`LOG_AUTH`	authorization
`LOG_LPR`	printers
`LOG_LOCAL`n	local use

```
Nov  3 06:12:15 bart xntpd[229]: time reset
```

getopt

Standard Unix command line: “-t” or “-f arg”

```
#include <stdlib.h>
while ((c = getopt(argc, argv, "tf:")) != -1) {
  switch (c) {
    case ’t’:
      printf("flag -t\n");
      break;
    case ’f’:
      printf("argument -f = %s\n", optarg);
      break;
    case ’?’:
      printf("syntax error on %c\n", optopt);
      break;
  }
}
```