

# Client/Server Paradigms

- Basic functions
- Iterative, connectionless servers
- Iterative, connection-oriented servers
- Concurrent, connection-oriented servers
- Single-process, concurrent servers
- Multi-protocol servers
- Multi-service servers
- Concurrency in clients

## connectTCP and connectUDP

```
int connectTCP( host, service )
```

```
char *host; /* name of host to which connection is desired */
```

```
char *service; /* service associated with the desired port */
```

```
{
```

```
    return connectsock( host, service, "tcp");
```

```
}
```

```
int connectUDP( host, service )
```

```
char *host; /* name of host to which connection is desired */
```

```
char *service; /* service associated with the desired port */
```

```
{
```

```
    return connectsock(host, service, "udp");
```

```
}
```

## Connectsock -- (1)

```
int connectsock( host, service, protocol )
char  *host; /* name of host to which connection is desired */
char  *service; /* service associated with the desired port */
char  *protocol; /* name of protocol to use ("tcp" or "udp") */
{
    struct hostent  *phe; /* pointer to host information entry */
    struct servent  *pse; /* pointer to service information entry
                          */
    struct protoent *ppe; /* pointer to protocol information entry*/
    struct sockaddr_in sin; /* an Internet endpoint address */
    int    s, type; /* socket descriptor and socket type */

    bzero((char *)&sin, sizeof(sin));
    sin.sin_family = AF_INET;
```

## Connectsock -- (2)

```
/* Map service name to port number */
if ( pse = getservbyname(service, protocol) )
    sin.sin_port = pse->s_port;
else if ( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
    errexit("can't get \"%s\" service entry\n", service);

/* Map host name to IP address, allowing for dotted decimal */
if ( phe = gethostbyname(host) )
    bcopy(phe->h_addr, (char *) &sin.sin_addr, phe->h_length);
else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
    errexit("can't get \"%s\" host entry\n", host);
```

# Services

## Sample /etc/services file

```
# /etc/services
# This file associates official service names and aliases with the port
# number and protocol the services use. The form for each entry is:
# <official service name> <port number/protocol name> <aliases>
#
echo          7/tcp          # Echo
echo          7/udp          #
discard      9/tcp          sink null    # Discard
discard      9/udp          sink null    #
sysstat      11/tcp         users       # Active Users
qotd         17/tcp         quote       # Quote of the Day
ftp-data     20/tcp         # File Transfer Protocol (Data)
ftp          21/tcp         # File Transfer Protocol (Control)
telnet       23/tcp         # Virtual Terminal Protocol
smtp         25/tcp         # Simple Mail Transfer Protocol
domain       53/tcp         nameserver  # Domain Name Service
domain       53/udp         nameserver  #
bootps       67/udp         # Bootstrap Protocol Service
bootpc       68/udp         # Bootstrap Protocol Client
hostnames    101/tcp        hostname    # NIC Host Name Server
pop          109/tcp        postoffice  # Post Office Protocol - Version 2
portmap      111/tcp        sunrpc      # Sun Remote Procedure Protocol
portmap      111/udp        sunrpc      #
nntp         119/tcp        readnews    # Network News Transfer Protocol
ntp          123/udp        # Network Time Protocol
snmp         161/udp        snmpd       # Simple Network Management Protocol
biff         512/udp        comsat      # mail notification
exec         512/tcp        # remote execution, passwd required
login        513/tcp        # remote login
who          513/udp        whod        # remote who and uptime
syslog       514/udp        # remote system logging
printer      515/tcp        spooler     # remote print spooling
route        520/udp        router routed # routing information protocol
kerberos     750/udp        kdc         # Kerberos (server) udp -kfall
kerberos     750/tcp        kdc         # Kerberos (server) tcp -kfall
krbupdate    760/tcp        kreg        # Kerberos registration -kfall
nfsd         2049/udp       # NFS remote file system
```

# Hosts

## Excerpt from /etc/hosts file

---

```
#/etc/hosts
# The form for each entry is:
# <internet address> <official hostname> <aliases>
#
127.0.0.1      localhost      loopback
232.48.1.107   merlin
232.48.1.215   arthur
232.48.1.200   lancelet
232.48.2.35    guenevere
232.48.1.104   robin
232.48.1.183   sherlock
232.48.1.177   dracula
232.48.2.49    godzilla
```

---

## Connectsock -- (3)

```
/* Map protocol name to protocol number */
if ( (ppe = getprotobyname(protocol)) == 0)
    errexit("can't get \"%s\" protocol entry\n", protocol);

/* Use protocol to choose a socket type */
if (strcmp(protocol, "udp") == 0)
    type = SOCK_DGRAM;
else
    type = SOCK_STREAM;

/* Allocate a socket */
s = socket(PF_INET, type, ppe->p_proto);
if (s < 0)
    errexit("can't create socket: %s\n", sys_errlist[errno]);
```

# Protocol List

## Example /etc/protocols file

---

```
#/etc/protocols
# This file contains information regarding the known protocols
# used in the DARPA Internet
#
# The form for each entry is:
# <official protocol name> <Protocol number> <aliases>
#
# Internet (IP) protocols
#
ip          0      IP          # Internet protocol, pseudo protocol number
icmp        1      ICMP        # internet control message protocol
ggp         3      GGP         # gateway-gateway protocol
tcp         6      TCP         # transmission control protocol
egp         8      EGP         # exterior gateway protocol
pup         12     PUP         # PARC universal packet protocol
udp         17     UDP         # user datagram protocol
hmp         20     HMP         # host monitoring protocol
xns-idp     22     XNS-IDP     # Xerox NS IDP
rdp         27     RDP         # "reliable datagram" protocol
```

---



## Connectsock -- (4)

```
/* Connect the socket */  
if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)  
    errexit("can't connect to %s.%s: %s\n", host, service,  
            sys_errlist[errno]);  
return s;  
}
```

# TCPdaytime.c -- (1)

```
int
main(argc, argv)
int argc;
char    *argv[];
{
    char *host = "localhost";      /* host to use if none supplied */
    char *service = "daytime"; /* default service port */
```

## TCPdaytime.c -- (2)

```
switch (argc) {
case 1:
    host = "localhost";
    break;
case 3:
    service = argv[2];
    /* FALL THROUGH */
case 2:
    host = argv[1];
    break;
default:
    fprintf(stderr, "usage: TCPdaytime [host [port]]\n");
    exit(1);
}
TCPdaytime(host, service);
exit(0);
}
```

## TCPdaytime.c -- (2)

```
TCPdaytime(host, service)
char      *host;
char      *service;
{
    char    buf[LINELEN+1]; /* buffer for one line of text */
    int     s, n;           /* socket, read count */

    s = connectTCP(host, service);

    while( (n = read(s, buf, LINELEN)) > 0) {
        buf[n] = '\0';      /* insure null-terminated */
        (void) fputs( buf, stdout );
    }
}
```

## UDPtime.c -- (1)

```
int main(argc, argv)
int argc;
char    *argv[];
{
    char *host = "localhost";      /* host to use if none supplied */
    char *service = "time"; /* default service name */
    time_t      now;               /* 32-bit integer to hold time */
    int    s, n;                   /* socket descriptor, read count*/
    switch (argc) {
        ....
        See TCPdaytime.c -- (2)
        ....
    }
```

## UDPtime.c -- (2)

```
s = connectUDP(host, service);
```

```
(void) write(s, MSG, strlen(MSG));
```

```
/* Read the time */
```

```
n = read(s, (char *)&now, sizeof(now));
```

```
if (n < 0)
```

```
    errexit("read failed: %s\n", sys_errlist[errno]);
```

```
now = ntohl((u_long)now); /* put in host byte order */
```

```
now -= UNIXEPOCH; /* convert UCT to UNIX epoch */
```

```
printf("%s", ctime(&now));
```

```
exit(0);
```

```
}
```

# passiveTCP and passiveUDP

```
int
passiveTCP( service, qlen )
char  *service; /* service associated with the desired port */
int  qlen; /* maximum server request queue length */
{
    return passivesock(service, "tcp", qlen);
}
```

```
int
passiveUDP( service )
char  *service; /* service associated with the desired port */
{
    return passivesock(service, "udp", 0);
}
```

## Passivesock -- (1)

```
int passivesock( service, protocol, qlen )
char  *service; /* service associated with the desired port */
char  *protocol; /* name of protocol to use ("tcp" or "udp") */
int  qlen; /* maximum length of the server request queue */
{
    struct servent  *pse; /* pointer to service information          entry */
    struct protoent *ppe; /* pointer to protocol information entry */
    struct sockaddr_in sin; /* an Internet endpoint address */
    int  s, type; /* socket descriptor and socket type */

    bzero((char *)&sin, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
```



## Passivesock -- (2)

```
/* Map service name to port number */
if ( pse = getservbyname(service, protocol) )
    sin.sin_port = htons(ntohs((u_short)pse->s_port) + portbase);
else if ( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
    errexit("can't get \"%s\" service entry\n", service);

/* Map protocol name to protocol number */
if ( (ppe = getprotobyname(protocol)) == 0 )
    errexit("can't get \"%s\" protocol entry\n", protocol);

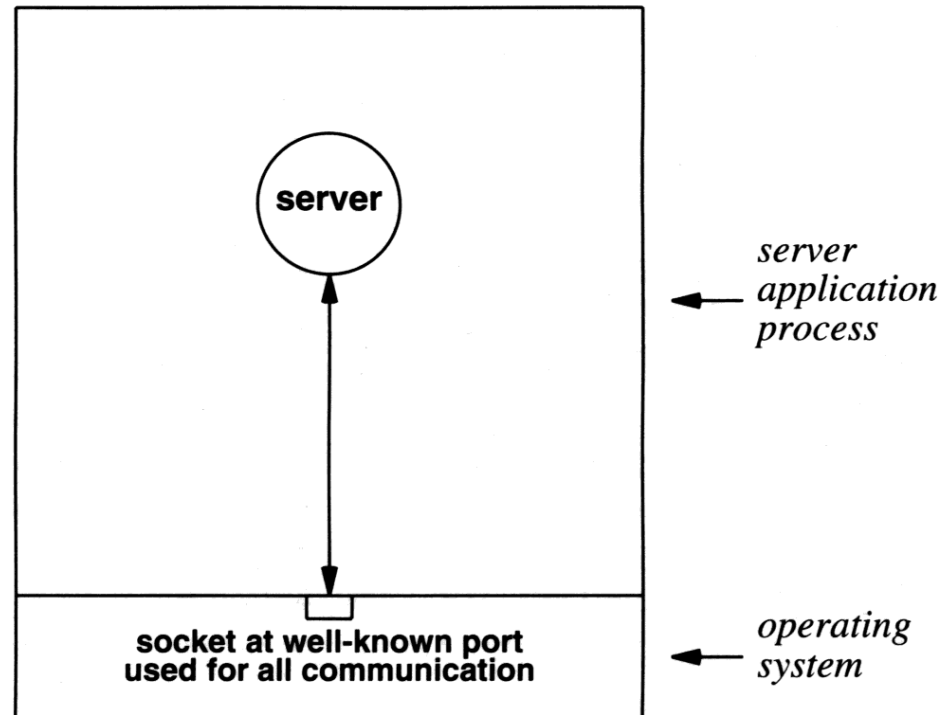
/* Use protocol to choose a socket type */
if (strcmp(protocol, "udp") == 0)
    type = SOCK_DGRAM;
else
    type = SOCK_STREAM;
```

## Passivesock -- (3)

```
/* Allocate a socket */
s = socket(PF_INET, type, ppe->p_proto);
if (s < 0)
    errexit("can't create socket: %s\n", sys_errlist[errno]);

/* Bind the socket */
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    errexit("can't bind to %s port: %s\n", service,
            sys_errlist[errno]);
if (type == SOCK_STREAM && listen(s, qlen) < 0)
    errexit("can't listen on %s port: %s\n", service,
            sys_errlist[errno]);
return s;
}
```

# Iterative, Connectionless Servers



The process structure for an iterative, connectionless server. A single server process communicates with many clients using one socket.

- Example: `UDPtimed.c` (next slides)

# UDPtimed.c -- (1)

```
int main(argc, argv)
int  argc;
char*argv[];
{
    struct sockaddr_in fsin; /* the from address of a client */
    char  *service = "time"; /* service name or port number */
    char  buf[1];    /* "input" buffer; any size > 0 */
    int    sock;      /* server socket */
    time_t now;       /* current time */
    int    alen;      /* from-address length */

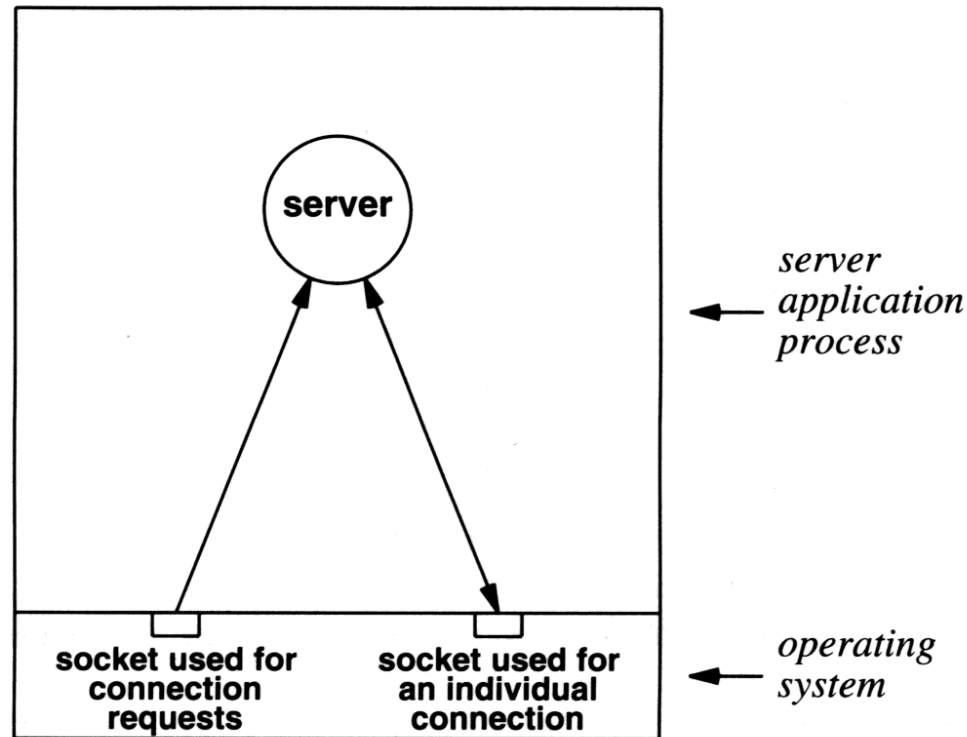
    switch (argc) {
    case  1:
        break;
    case  2:
        service = argv[1];
        break;
    default:
        errexit("usage: UDPtimed [port]\n");
    }
```

## UDPTimed.c -- (2)

```
sock = passiveUDP(service);
```

```
while (1) {  
    alen = sizeof(fsin);  
    if (recvfrom(sock, buf, sizeof(buf), 0,  
        (struct sockaddr*)&fsin, &alen) < 0)  
        errexit("recvfrom: %s\n", sys_errlist[errno]);  
    (void) time(&now);  
    now = htonl((u_long)(now + UNIXEPOCH));  
    (void) sendto(sock, (char *)&now,  
        sizeof(now), 0, (struct sockaddr *)  
        &fsin, sizeof(fsin));  
}  
}
```

# Iterative, Connection-Oriented Servers



The process structure of an iterative, connection-oriented server. The server waits at the well-known port for a connection, and then communicates with the client over that connection.

- Example: TCPtimed.c (next slides)

# TCPdaytimed.c -- (1)

```
int main(argc, argv)
int  argc;
char*argv[];
{
    struct  sockaddr_in fsin;      /* the from address of a client    */
    char    *service = "daytime"; /* service name or port number    */
    int     msock, ssock;         /* master & slave sockets        */
    int     alen;                 /* from-address length            */

    switch (argc) {
    case  1:
        break;
    case  2:
        service = argv[1];
        break;
    default:
        errexit("usage: TCPdaytimed [port]\n");
    }
}
```

## TCPdaytimed.c -- (2)

```
msock = passiveTCP(service, QLEN);

while (1) {
    ssock = accept(msock, (struct sockaddr *)
                  &fsin, &alen);

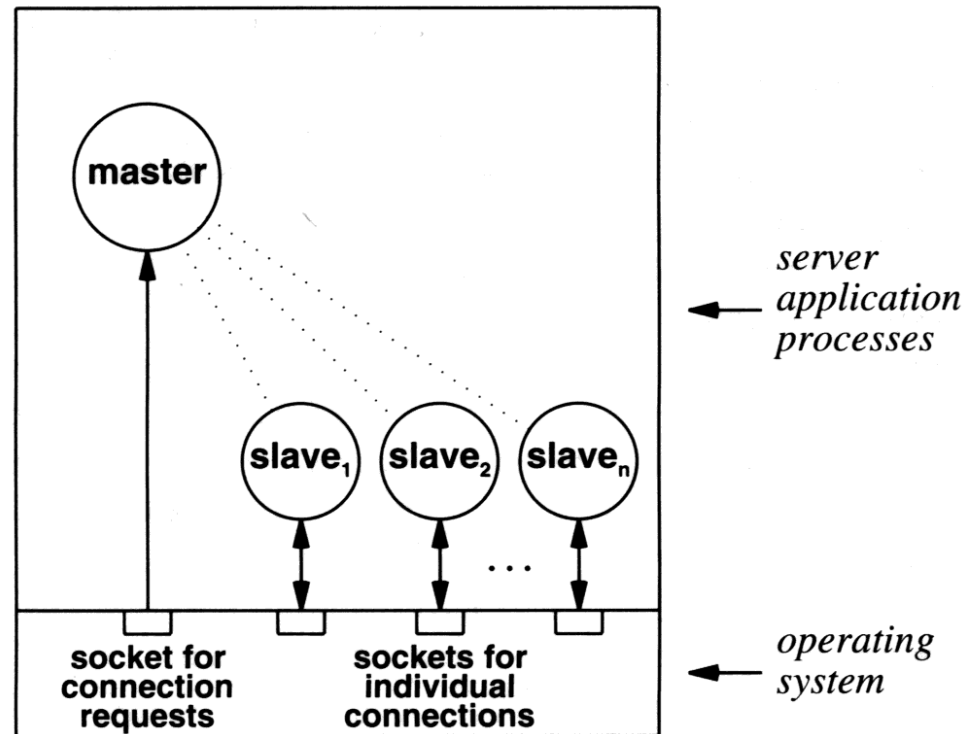
    if (ssock < 0)
        errexit("acceptfailed:%s\n",sys_errlist[errno]);
    (void) TCPdaytimed(ssock);
    (void) close(ssock);
}
}
```



## TCPdaytimed.c -- (3)

```
/*-----  
 * TCPdaytimed - do TCP DAYTIME protocol  
 *-----  
 */  
int  
TCPdaytimed(fd)  
int fd;  
{  
    char    *pts;                /* pointer to time string */  
    time_t  now;                /* current time */  
    char    *ctime();  
  
    (void) time(&now);  
    pts = ctime(&now);  
    (void) write(fd, pts, strlen(pts));  
    return 0;  
}
```

# Concurrent, Connection-Oriented Servers



The process structure of a concurrent, connection-oriented server. A master server process accepts each incoming connection, and creates a slave process to handle it.

- Example: TCPEchod.c (next slides)

# TCPechod.c -- (1)

```
int main(argc, argv)
int  argc;
char  *argv[];
{
    char  *service = "echo"; /* service name or port number */
    struct sockaddr_in fsin; /* the address of a client */
    int    alen; /* length of client's address */
    int    msock; /* master server socket */
    int    ssock; /* slave server socket */
    switch (argc) {
    case 1:
        break;
    case 2:
        service = argv[1];
        break;
    default:
        errexit("usage: TCPechod [port]\n");
    }
}
```

## TCPeched.c -- (2)

```
msock = passiveTCP(service, QLEN);
```

```
(void) signal(SIGCHLD, reaper);
```

```
while (1) {  
    alen = sizeof(fsin);  
    ssock = accept(msock, (struct sockaddr *)  
                  &fsin, &alen);  
    if (ssock < 0) {  
        if (errno == EINTR)  
            continue;  
        errexit("accept: %s\n", sys_errlist[errno]);  
    }  
}
```

## TCPechod.c -- (3)

```
switch (fork()) {  
  case 0:          /* child */  
    (void) close(msock);  
    exit(TCPechod(ssock));  
  default: /* parent */  
    (void) close(ssock);  
    break;  
  case -1:  errexit("fork: %s\n", sys_errlist[errno]);  
  }  
}
```

## TCPechod.c -- (4)

```
int TCPechod(fd)
int fd;
{
    char  buf[BUFSIZ];
    int   cc;

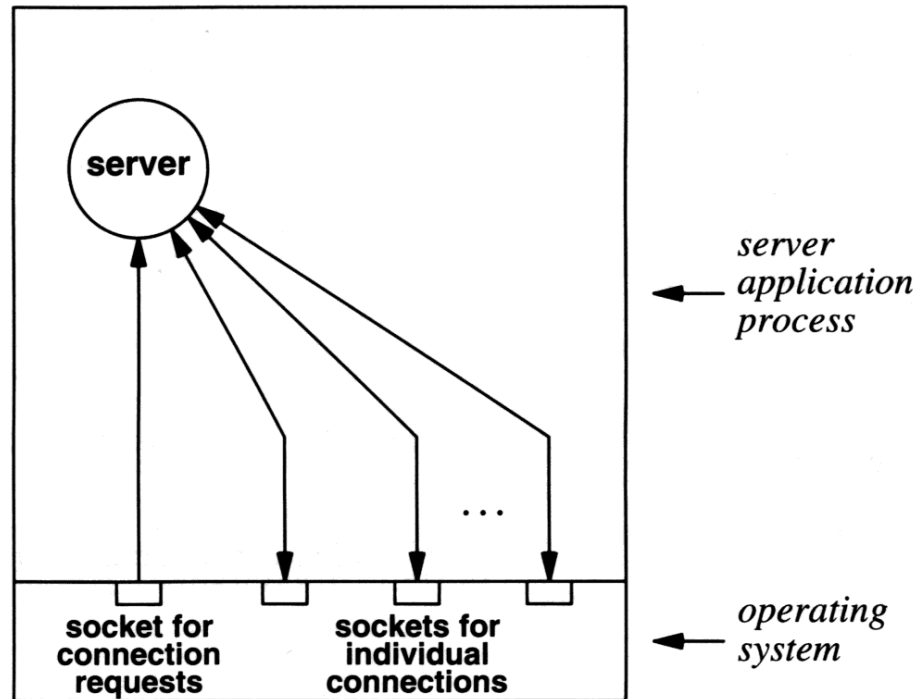
    while (cc = read(fd, buf, sizeof(buf))) {
        if (cc < 0)
            errexit("echo read: %s\n", sys_errlist[errno]);
        if (write(fd, buf, cc) < 0)
            errexit("echo write: %s\n", sys_errlist[errno]);
    }
    return 0;
}
```

## TCPechod.c -- (5)

```
int reaper()
{
    union wait    status;

    while (wait3(&status, WNOHANG,
        (struct rusage *)0) >= 0)
        /* empty */;
}
```

# Single-Process, Concurrent Servers (TCP)



The process structure of a connection-oriented server that achieves concurrency with a single process. The process manages multiple sockets.

- Example: TCPmechod.c (next slides)



# TCPmechod.c (1)

```
/* TCPmechod.c - main, echo */

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

#include <unistd.h>
#include <string.h>
#include <stdio.h>

#define QLEN 5 /* maximum connection queue length */
#define BUFSIZE 4096

extern int errno;
int errexit(const char *format, ...);
int passiveTCP(const char *service, int qlen);
int echo(int fd);
```

## TCPmechod.c (2)

```
/*-----  
 * main - Concurrent TCP server for ECHO service  
 *-----*/  
  
int main(int argc, char *argv[])  
{  char  *service = "echo"; /* service name or port number */  
   struct sockaddr_in fsin; /* the from address of a client*/  
   int    msock;           /* master server socket */  
   fd_set      rfds;           /* read file descriptor set */  
   fd_set      afds;           /* active file descriptor set */  
   int    alen;             /* from-address length */  
   int    fd, nfds;
```

## TCPmechod.c (2)

```
switch (argc) {  
case 1:  
    break;  
case 2:  
    service = argv[1];  
    break;  
default:  
    errexit("usage: TCPmechod [port]\n");  
}
```

## TCPmechod.c (3)

```
msock = passiveTCP(service, QLEN);
```

```
nfds = getdtablesize();
```

```
FD_ZERO(&afds);
```

```
FD_SET(msock, &afds);
```

```
while (1) {
```

```
    memcpy(&rfd, &afds, sizeof(rfd));
```

```
    if (select(nfds, &rfd, (fd_set *)0, (fd_set *)0,  
        (struct timeval *)0) < 0)
```

```
        errexit("select: %s\n", strerror(errno));
```

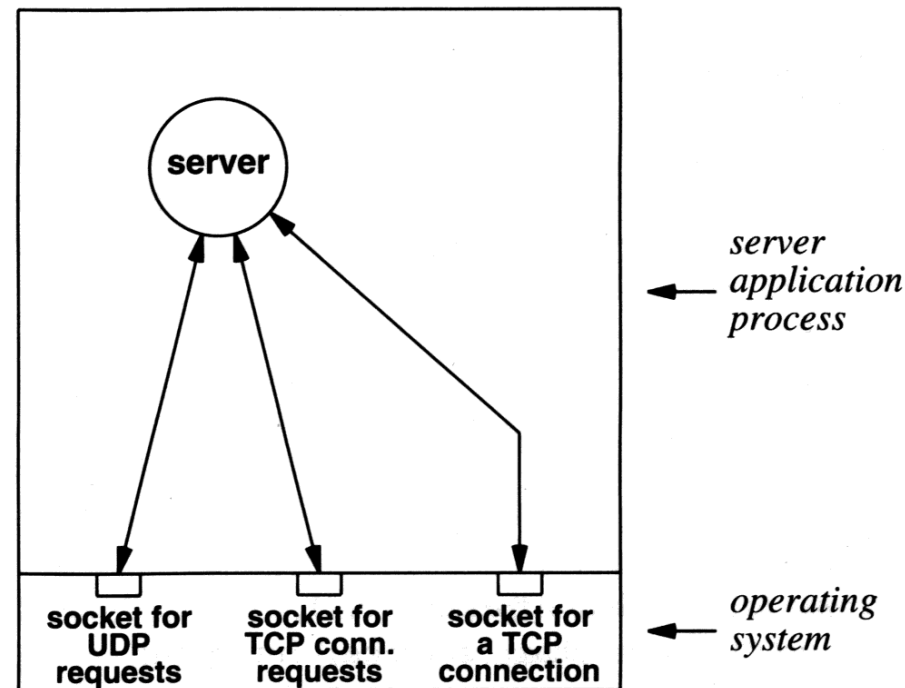
# TCPmechod.c (4)

```
if (FD_ISSET(msock, &rfd)) {
    int      ssock;
    alen = sizeof(fsin);
    ssock = accept(msock, (struct sockaddr *)&fsin,
                  &alen);
    if (ssock < 0)
        errexit("accept: %s\n", strerror(errno));
    FD_SET(ssock, &afds);
}
for (fd=0; fd<nfd; ++fd)
    if (fd != msock && FD_ISSET(fd, &rfd))
        if (echo(fd) == 0) {
            (void) close(fd);
            FD_CLR(fd, &afds);
        }
}
```

# TCPmechod.c (5)

```
/*-----  
 * echo - echo one buffer of data, returning byte count  
 *-----  
 */  
int  
echo(int fd)  
{  
    char    buf[BUFSIZE];  
    int     cc;  
  
    cc = read(fd, buf, sizeof buf);  
    if (cc < 0)  
        errexit("echo read: %s\n", strerror(errno));  
    if (cc && write(fd, buf, cc) < 0)  
        errexit("echo write: %s\n", strerror(errno));  
    return cc;  
}
```

# Multiprotocol Servers



The process structure of an iterative, multiprotocol server. At any time, the server has at most three sockets open: one for UDP requests, one for TCP connection requests, and a temporary one for an individual TCP connection.

● Example: `daytimed.c` (next slides)

## daytimed.c -- (1)

```
int
main(argc, argv)
int argc;
char    *argv[];
{
    char *service = "daytime"; /* service name or port number */
    char buf[LINELEN+1]; /* buffer for one line of text */
    struct sockaddr_in fsin; /* the request from address */
    int  alen; /* from-address length */
    int  tsock; /* TCP master socket */
    int  usock; /* UDP socket */
    int  nfd;
    fd_set rfd; /* readable file descriptors */
```



## daytimed.c -- (2)

```
switch (argc) {  
case 1:  
    break;  
case 2:  
    service = argv[1];  
    break;  
default:  
    errexit("usage: daytimed [port]\n");  
}
```

## daytimed.c -- (3)

```
tsock = passiveTCP(service, QLEN);
usock = passiveUDP(service);
nfd = MAX(tsock, usock) + 1; /* bit number of max fd */

FD_ZERO(&rfd);

while (1) {
    FD_SET(tsock, &rfd);
    FD_SET(usock, &rfd);

    if (select(nfd, &rfd, (fd_set *)0,
               (fd_set *)0, (struct timeval *)0) < 0)
        perror("select error");
}
```

## daytimed.c -- (4)

```
if (FD_ISSET(tsock, &rfdset)) {  
    int      ssock; /* TCP slave socket */  
  
    alen = sizeof(fsin);  
    ssock = accept(tsock, (struct sockaddr *) &fsin, &alen);  
    if (ssock < 0)  
        errexit("accept failed: %s\n", sys_errlist[errno]);  
    daytime(buf);  
  
    write(ssock, buf, strlen(buf));  
    close(ssock);  
}
```

## daytimed.c -- (5)

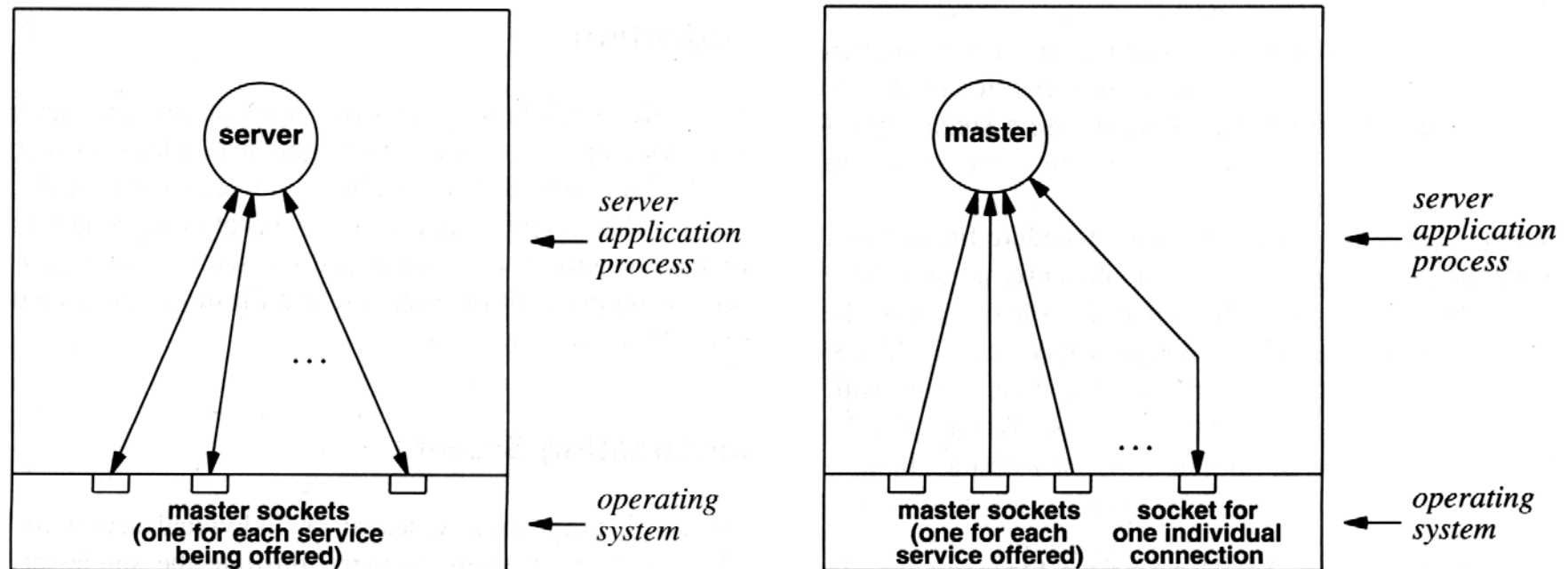
```
if (FD_ISSET(usock, &rfdsets)) {  
    alen = sizeof(fsin);  
    if (recvfrom(usock, buf,  
                sizeof(buf), 0, (struct  
                sockaddr *)&fsin, &alen) < 0)  
        errexit("recvfrom: %s\n",  
                sys_errlist[errno]);  
    daytime(buf);  
    sendto(usock, buf, strlen(buf), 0,  
           (struct sockaddr *)&fsin,  
           sizeof(fsin));  
}  
}
```

## daytimed.c -- (6)

```
int daytime(buf)
char    buf[];
{
    time, now;
    (void) time(&now);
    sprintf(buf, "%s", ctime(&now));
}
```

# Multiservice Servers

- Connectionless, Multiservice Server
- Connection-oriented, Multiservice Server



# INETD

- `rc`: The initial process to invoke all servers.
- `/etc/inetd.conf`: table of all servers initially (invoked by `rc`) (see next slide)

**Sample /etc/inet.conf file**

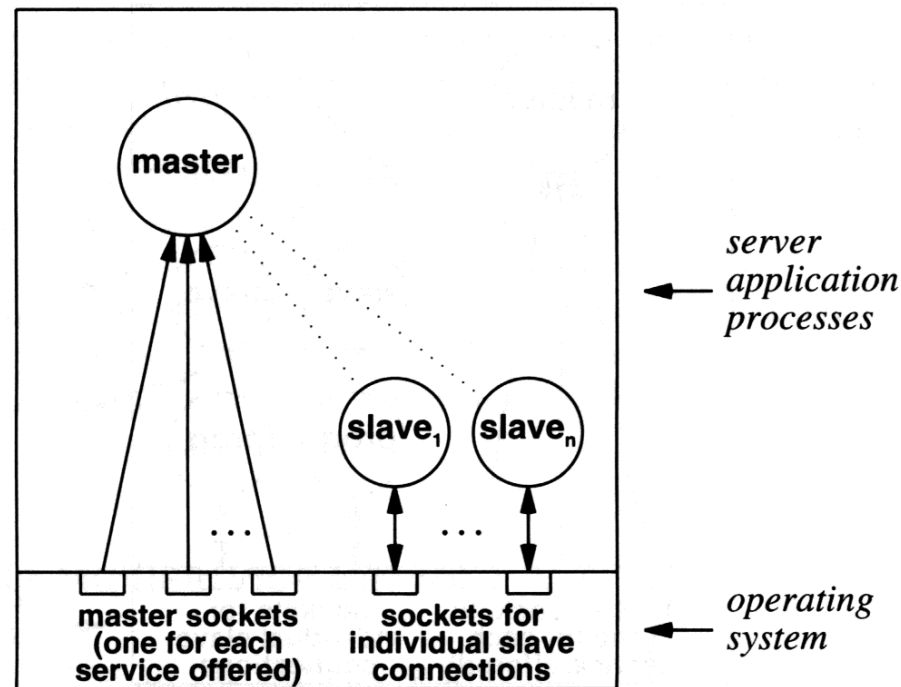
```

#/etc/inetd.conf
#Inetd reads its configuration information from this file upon execution
#and at some later time if it is reconfigured.
#A line in the configuration file has the following fields:
#
#  service name      as in /etc/services
#  socket type       either 'stream' or 'dgram'
#  protocol          as in /etc/protocols
#  wait/nowait       only applies to datagram sockets, stream
#                   sockets should specify nowait
#  user              name of user as whom the server should run
#  server program    absolute pathname for the server inetd will
#                   execute
#  server program args.  server program arguments where argv[0] is
#                   the name of the server
#
ftp      stream  tcp  nowait  root    /etc/ftpd  ftpd -l
telnet   stream  tcp  nowait  root    /etc/telnetd telnetd
tftp     dgram   udp  wait   root    /etc/tftpd  tftpd
bootps   dgram   udp  wait   root    /etc/bootpd  bootpd
finger   stream  tcp  nowait  bin     /etc/fingerd fingerd
login    stream  tcp  nowait  root    /etc/rlogind rlogind
shell    stream  tcp  nowait  root    /etc/remshd  remshd
exec     stream  tcp  nowait  root    /etc/rexecd  rexecd
printer  stream  tcp  nowait  root    /usr/lib/rlpdaemon rlpdaemon -l
daytime  stream  tcp  nowait  root    internal
daytime  dgram   udp  nowait  root    internal
time     stream  tcp  nowait  root    internal
time     dgram   udp  nowait  root    internal
echo     stream  tcp  nowait  root    internal
echo     dgram   udp  nowait  root    internal
discard  stream  tcp  nowait  root    internal
discard  dgram   udp  nowait  root    internal
chargen  stream  tcp  nowait  root    internal
chargen  dgram   udp  nowait  root    internal
#
#  rpc services, registered by inetd with portmap
#
rpc stream  tcp  nowait  root    /usr/etc/rpc.rexd  100017 1  rpc.rexd
rpc dgram   udp  wait    root    /usr/etc/rpc.rstatd 100001 1-3  rpc.rstatd

```



# Concurrent, Connection-oriented, Multiservice Server



The process structure for a concurrent, connection-oriented, multiservice server. The master process handles incoming connection requests, while a slave process handles each connection.

● Example: `superd.c` (next slides)

## superd.c -- (1)

```
struct service {
    char  *sv_name;
    char  sv_useTCP;
    int    sv_sock;
    int    (*sv_func)();
} svent[] = {
    { "echo", TCP_SERV, NOSOCK, TCPEchod },
    { "chargen", TCP_SERV, NOSOCK, TCPchargend },
    { "daytime", TCP_SERV, NOSOCK, TCPdaytimed },
    { "time", TCP_SERV, NOSOCK, TCPTimed },
    { 0, 0, 0, 0 },
};
```

## superd.c -- (2)

```
int main(argc, argv)
int  argc;
char*argv[];
{
    struct service      *psv,          /* service table pointer */
        *fd2sv[NOFILE];               /* map fd to service pointer */
    int    fd, nfd;
    fd_set afds, rfds;                 /* readable file descriptors */

    switch (argc) {
    case 1:
        break;
    case 2:
        portbase = (u_short) atoi(argv[1]);
        break;
    default:
        errexit("usage: superd [portbase]\n");
    }
}
```

## superd.c -- (3)

```
nfds = 0;
FD_ZERO(&afds);
for (psv = &svent[0]; psv->sv_name; ++psv) {
    if (psv->sv_useTCP)
        psv->sv_sock = passiveTCP(psv->sv_name, QLEN);
    else
        psv->sv_sock = passiveUDP(psv->sv_name);
    fd2sv[psv->sv_sock] = psv;
    nfds = MAX(psv->sv_sock+1, nfds);
    FD_SET(psv->sv_sock, &afds);
}
(void) signal(SIGCHLD, reaper);
```

## superd.c -- (4)

```
while (1) {
    bcopy((char *)&afds, (char *)&rfd, sizeof(rfd));
    if (select(nfds, &rfd, (fd_set *)0,
              (fd_set *)0, (struct timeval *)0)
        < 0) {
        if (errno == EINTR)
            continue;
        errexit("select error: %s\n",
                sys_errlist[errno]);
    }
    for (fd=0; fd<nfd; ++fd)
        if (FD_ISSET(fd, &rfd)) {
            psv = fd2sv[fd];
            if (psv->sv_useTCP)
                doTCP(psv);
            else
                psv->sv_func(psv->sv_sock);
        }
}
```

## superd.c -- (5)

```
int doTCP(psv)
struct service      *psv;
{
    struct sockaddr_in fsin; /* the request from address */
    int    alen      ;      /* from-address length */
    int    fd, ssock;
    alen = sizeof(fsin);
    ssock = accept(psv->sv_sock, (struct sockaddr *)
            &fsin, &alen);
    if (ssock < 0)
        errexit("accept: %s\n", sys_errlist[errno]);
}
```

## superd.c -- (6)

```
switch (fork()) {  
case 0:      break;  
case -1:     errexit("fork: %s\n", sys_errlist[errno]);  
default:     (void) close(ssock);  
             return;          /* parent */  
}  
/* child */  
for (fd = NOFILE; fd >= 0; --fd)  
    if (fd != ssock)  
        (void) close(fd);  
exit(psv->sv_func(ssock));  
}
```

## sv\_funcs.c -- (1)

```
int TCPEchod(fd)
int fd;
{
    char  buf[BUFSIZ];
    int   cc;

    while (cc = read(fd, buf, sizeof buf)) {
        if (cc < 0) errexit("echo read: %s\n", buf);

        if (write(fd, buf, cc) < 0)
            errexit("echo write: %s\n",
                sys_errlist[errno]);
    }
    return 0;
}
```



## sv\_funcs.c -- (2)

```
int TCPcharend(fd)
{
    char    c, buf[LINELEN+2]; /* print LINELEN chars + \r\n */
    c = ' ';
    buf[LINELEN] = '\r';
    buf[LINELEN+1] = '\n';
    while (1) {
        int    i;
        for (i=0; i<LINELEN; ++i) {
            buf[i] = c++;
            if (c > '~') c = ' ';
        }
        if (write(fd, buf, LINELEN+2) < 0)
            break;
    }
    return 0;
}
```

## sv\_funcs.c -- (3)

```
int TCPdaytimed(fd)
Int fd;
{
    char buf[LINELEN], *ctime();
    time_t      time(), now;

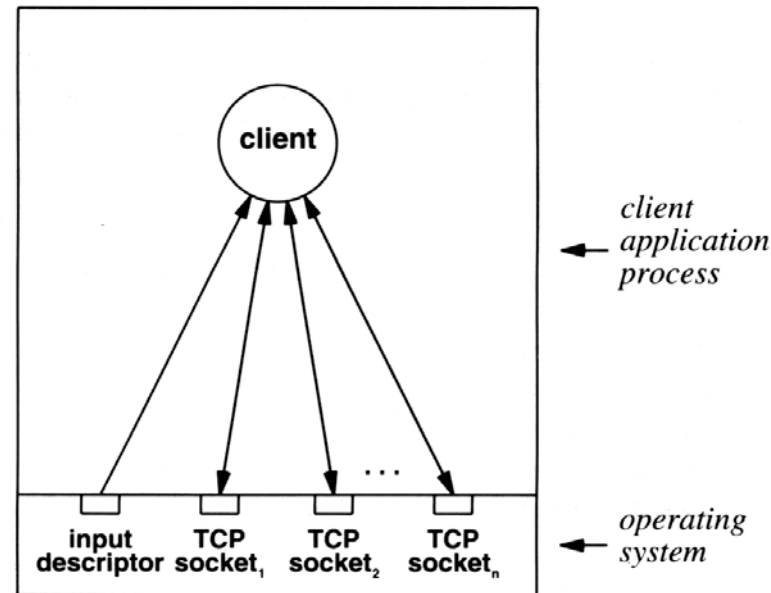
    (void) time(&now);
    sprintf(buf, "%s", ctime(&now));
    (void) write(fd, buf, strlen(buf));
    return 0;
}
```

## sv\_funcs.c -- (4)

```
int TCPTimed(fd)
int fd;
{
    time_t      now, time();
    u_long      htonl();

    (void) time(&now);
    now = htonl((u_long)(now + UNIXEPOCH));
    (void) write(fd, (char *)&now, sizeof(now));
    return 0;
}
```

# Concurrency in Clients



The process structure most often used with UNIX to provide apparent concurrency in a single-process, connection-oriented client. The client uses *select* to handle multiple connections concurrently.

● Example: TCPtecho.c (next slides)

# TCPtecho.c -- (1)

```
/* TCPtecho.c - main, TCPtecho, reader, writer, mstime */
#include <sys/types.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <stdio.h>

extern int  errno;
extern char *sys_errlist[];

#define      BUFSIZE          4096      /* write buffer size */
#define      CCOUNT          64*1024  /* default character count */
#define      USAGE            "usage: TCPtecho [ -c count ] host1 host2...\n"

char*hname[NOFILE]; /* fd to host name mapping */
int  rc[NOFILE], wc[NOFILE]; /* read/write character counts */
charbuf[BUFSIZE];    /* read/write data buffer */
longmstime();
```

## TCPtecho.c -- (2)

```
int
main(argc, argv)
int  argc;
char*argv[];
{
    int    ccount = CCOUNT;
    int    i, hcount, maxfd, fd;
    int    one = 1;
    fd_set afds;

    hcount = 0;
    maxfd = -1;
    for (i=1; i<argc; ++i) {
        if (strcmp(argv[i], "-c") == 0) {
            if (++i < argc && (ccount = atoi(argv[i])))
                continue;
            errexit(USAGE);
        }
    }
```

## TCPtecho.c -- (3)

```
/* else, a host */

fd = connectTCP(argv[i], "echo");
if (ioctl(fd, FIONBIO, (char *)&one))
    errexit("can't mark socket nonblocking: %s\n", sys_errlist[errno]);
if (fd > maxfd)
    maxfd = fd;
hname[fd] = argv[i];
++hcount;
FD_SET(fd, &afds);
}
TCPtecho(&afds, maxfd+1, ccount, hcount);
exit(0);
}
```

## TCPtecho.c -- (4)

```
int
TCPtecho(pafds, nfd, ccount, hcount)
fd_set *pafds;
int nfd, ccount, hcount;
{
    fd_set rfd, wfd;          /* read/write fd sets */
    fd_set rcfd, wcfd;        /* read/write fd sets (copy) */
    int fd, i;

    for (i=0; i<BUFSIZE; ++i) /* echo data */
        buf[i] = 'D';
    bcopy((char *)pafds, (char *)&rcfd, sizeof(rcfd));
    bcopy((char *)pafds, (char *)&wcfd, sizeof(wcfd));
    for (fd=0; fd<nfd; ++fd)
        rc[fd] = wc[fd] = ccount;

    (void) mstime((long *)0); /* set the epoch */
}
```



## TCPtecho.c -- (5)

```
while (hcount) {
    bcopy((char *)&rcfds, (char *)&rfd, sizeof(rfd));
    bcopy((char *)&wcfds, (char *)&wfd, sizeof(wfd));

    if (select(nfds, &rfd, &wfd, (fd_set *)0, (struct timeval *)0) < 0)
        errexit("select failed: %s\n", sys_errlist[errno]);
    for (fd=0; fd<nfds; ++fd) {
        if (FD_ISSET(fd, &rfd))
            if (reader(fd, &rcfd) == 0)
                hcount--;
        if (FD_ISSET(fd, &wfd))
            writer(fd, &wcfd);
    }
}
```

## TCPtecho.c -- (6)

```
int reader(fd, pfdset)
int fd;
fd_set    *pfdset;
{
    long  now;
    int    cc;
    cc = read(fd, buf, sizeof(buf));
    if (cc < 0)
        errexit("read: %s\n", sys_errlist[errno]);
    if (cc == 0)
        errexit("read: premature end of file\n");
    rc[fd] -= cc;
    if (rc[fd])
        return 1;
```

## TCPtecho.c -- (6)

```
(void) mstime(&now);  
printf("%s: %d ms\n", hname[fd], now);  
(void) close(fd);  
FD_CLR(fd, pfdset);  
return 0;  
}
```

## TCPtecho.c -- (7)

```
Int writer(fd, pfdset)
int fd;
fd_set    *pfdset;
{
    int    cc;

    cc = write(fd, buf, MIN(sizeof(buf), wc[fd]));
    if (cc < 0)
        errexit("read: %s\n", sys_errlist[errno]);
    wc[fd] -= cc;
    if (wc[fd] == 0) {
        (void) shutdown(fd, 1);
        FD_CLR(fd, pfdset);
    }
}
```

## TCPtecho.c -- (8)

```
Long mstime(pms)
long      *pms;
{
    static struct timeval      epoch;
    struct timeval             now;

    if (gettimeofday(&now, (struct timezone *)0))
        errexit("gettimeofday: %s\n", sys_errlist[errno]);
    if (!pms) {
        epoch = now;
        return 0;
    }
    *pms = (now.tv_sec - epoch.tv_sec) * 1000;
    *pms += (now.tv_usec - epoch.tv_usec + 500)/ 1000;
    return *pms;
}
```

**Sample excerpt from sockets.h**

```
/* Copyright (c) 1982, 1985, 1986 Regents of the University of California.
All rights reserved.
Redistribution and use in source and binary forms are permitted provided
that the above copyright notice and this paragraph are duplicated in all
such forms and that any documentation, advertising materials, and other
materials related to such distribution and use acknowledge that the
software was developed by the University of California, Berkeley. The name
of the University may not be used to endorse or promote products derived
from this software without specific prior written permission.
THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
*/
* Types of sockets
*/
#define SOCK_STREAM      1      /* stream socket */
#define SOCK_DGRAM      2      /* datagram socket */
#define SOCK_RAW        3      /* raw-protocol interface */
#define SOCK_RDM        4      /* reliably-delivered message */
#define SOCK_SEQPACKET  5      /* sequenced packet stream */
/*
* Option flags per-socket
*/
#define SO_DEBUG          0x0001 /* turn on debugging info recording */
#define SO_ACCEPTCONN    0x0002 /* socket has had listen() */
#define SO_REUSEADDR     0x0004 /* allow local address reuse */
#define SO_KEEPAIVE      0x0008 /* keep connections alive */
#define SO_DONTROUTE     0x0010 /* just use interface addresses */
#define SO_BROADCAST     0x0020 /* permit sending of broadcast msgs */
#define SO_USELOOPBACK   0x0040 /* bypass hardware when possible */
#define SO_LINGER         0x0080 /* linger on close if data present */
#define SO_OOBINLINE     0x0100 /* leave received OOB data in line */
/*
* Additional options, not kept in so_options.
*/
#define SO_SNDBUF         0x1001 /* send buffer size */
#define SO_RCVBUF         0x1002 /* receive buffer size */
#define SO_SNDLOWAT       0x1003 /* send low-water mark */
#define SO_RCVLOWAT       0x1004 /* receive low-water mark */
#define SO_SNDTIMEO       0x1005 /* send timeout */
#define SO_RCVTIMEO       0x1006 /* receive timeout */
#define SO_ERROR          0x1007 /* get error status and clear */
#define SO_TYPE           0x1008 /* get socket type */
```