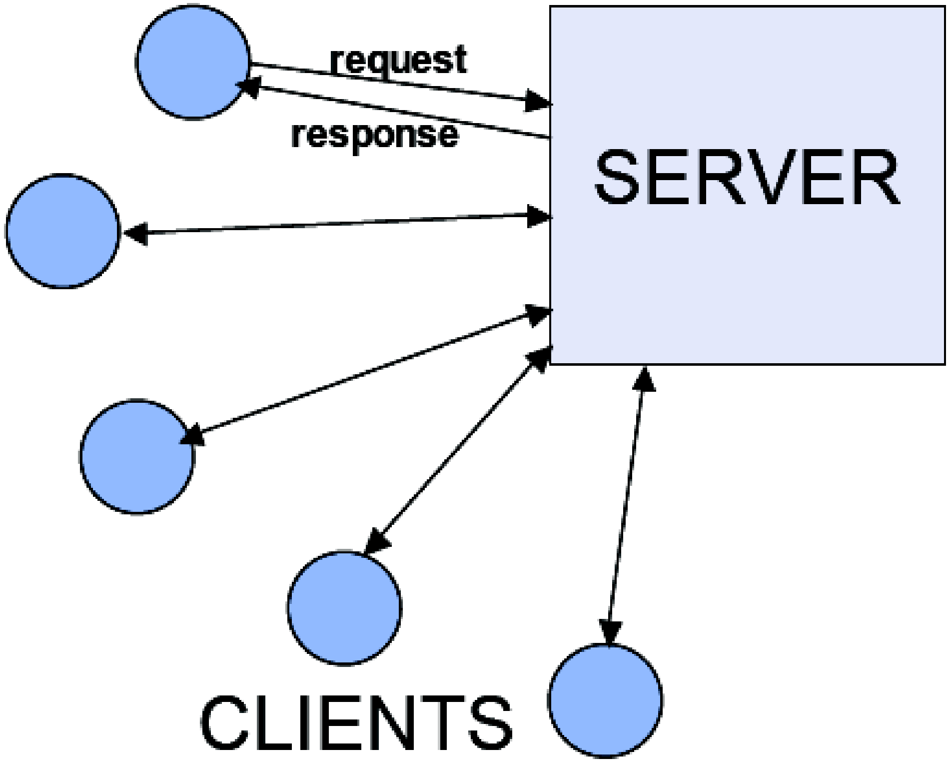
Sockets

Sockets

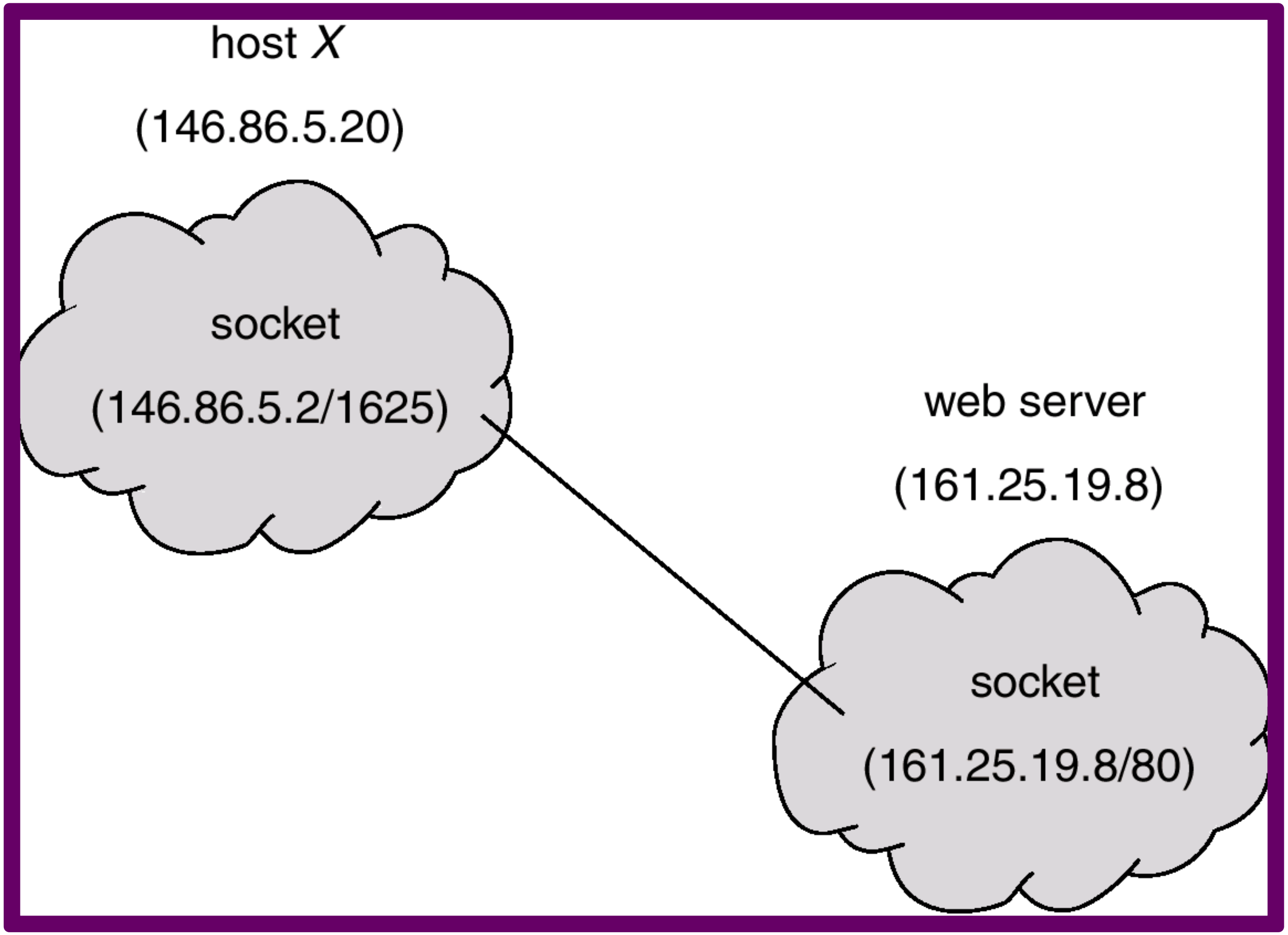
- File I/O:
  - open, close, read, write, seek, fcntl, ...
- Network communication:
  - developers extended set of file descriptors to include network connections.
  - extended read/write to work on these new file descriptors.
  - but other required functionality did not fit into the ‘openread-write-close’ paradigm.
- -> Socket API

Sockets

- Server listens for requests from clients
- Server: passive open
- Client: active open
- Example:
  - file server
  - web server
  - print server
  - mail server
  - name server
  - X window server



Sockets



host *X*

(146.86.5.20)

socket

(146.86.5.2/1625)

web server

(161.25.19.8)

socket

(161.25.19.8/80)

Basics

- The basic building block for communication is the socket.
- A socket is an endpoint of communication to which a name may be bound.
- Each socket in use has a *type* and *one or more associated processes*.

Domains

- Sockets exist within *communication domains*.
- A communication domain is an *abstraction* introduced to *bundle common properties* of processes communicating through sockets, e.g. socket name.
- For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named ‘‘/dev/foo’’.
- Sockets normally exchange data only with sockets in the same domain

Domains

- The 4.4BSD IPC facilities supported *four* separate communication domains
  - the *UNIX domain*, for on-system communication
  - the *Internet domain*, which is used by processes which communicate using the Internet standard communication protocols
  - the NS domain, which is used by processes which communicate using the Xerox standard communication protocols
  - the ISO OSI protocols

Socket Types

- Sockets are typed according to the communication properties visible to a user.
- Processes are presumed to communicate only between sockets of the same type
- Four types of sockets currently are available

Stream Sockets

- A *stream socket* provides for the *bidirectional, reliable, sequenced*, and *unduplicated* flow of data *without record boundaries*.
- Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes

Datagram Sockets

- A *datagram socket* supports *bidirectional* flow of data which is *not promised to be sequenced, reliable, or unduplicated*.
- Messages may be dropped, duplicated, and, possibly, delivered in an order different from the order in which they were was sent.
- An important characteristic of a datagram socket is that *record boundaries in data are preserved*.

Raw Sockets

- A *raw socket* provides users *access to the underlying communication protocols* which support socket abstractions.
- These sockets are *normally datagram oriented*,
- Not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol.

Sequenced Packet Sockets

- A sequenced packet socket is similar to a stream socket, with the exception that record boundaries are preserved.
- This interface is provided only as part of the NS socket abstraction.

Socket Creation

`s = socket(domain, type, protocol);`

- Create a socket in the specified domain and of the specified type.
- A particular protocol may also be requested.
- If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol. from those protocols which comprise the communication domain and which

Socket Creation

```
s = socket(domain, type, protocol);
```

- The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets.

Socket Creation

```
s = socket(domain, type, protocol);
```

- The domain is specified as one of:
  - AF_UNIX (Unix Domain)
  - AF_INET (Internet Domain)
  - AF_NS (NS Domain)
- The socket types are:
  - SOCK_STREAM
  - SOCK_DGRAM
  - SOCK_RAW
  - SOCK_SEQPACKET

Examples

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

- Creates a stream socket in the Internet domain

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

- Creates a datagram socket for on-machine use (Unix Domain)
- The default protocol (last argument to the socket call is **0**) should be correct for almost every situation

Socket Names

- A socket is created without a name.
- Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it.
- Communicating processes are bound by an association. In the Internet (and NS) domains, an association is composed of local and foreign addresses, and local and foreign ports.
- In the UNIX domain, an association is composed of local and foreign path names.

Socket Names

- In the Internet domain there may never be duplicate <*protocol, local address, local port, foreign address, foreign port*> tuples.
- UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate <*protocol, local pathname, foreign pathname*> tuples.

Binding Names

```
bind(s, name, namelen);
```

- The *bind()* system call allows a process to specify half of an association, <local address, local port> (or <local pathname>).
- The *connect()* and *accept()* primitives are used to complete a socket’s association.
- The *bound name* is a *variable length byte string* which is interpreted by the supporting protocol(s).

Binding Names

- In the Internet domain names contain an Internet address and port number.
- In the UNIX domain, names contain a path name and a family, which is always AF_UNIX.

Example

```
#include <sys/un.h>

...

struct sockaddr_un addr;

...

strcpy(addr.sun_path, "/tmp/foo");

addr.sun_family = AF_UNIX;

len = strlen(addr.sun_path) +
       sizeof (addr.sun_family)

bind(s, (struct sockaddr *) &addr, len);
```

Binding Names

- File name referred to in *addr.sun_path* is created as a socket in the system file space.
- The caller must, therefore, have write permission in the directory where *addr.sun_path* is to reside, and this file should be deleted by the caller when it is no longer needed.

Example

```
#include <sys/types.h>

#include <netinet/in.h>

...

struct sockaddr_in sin;

...

bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Connecting Sockets

- Connection establishment is usually asymmetric, with one process a ‘‘client’’ and the other a‘‘server’’.
- The server binds a socket to a well-known address and then passively ‘‘listens’’
- The client requests services from the server by initiating a ‘‘connection’’ to the server’s socket.
- On the client side the connect call is used to initiate a connection.

Connecting Sockets

```
// Unix Domain

struct sockaddr_un server;

...

connect(s, (struct sockaddr *)&server,
  strlen(server.sun_path) + sizeof
  (server.sun_family));

----------------------------------------

// Internet Domain

struct sockaddr_in server;

...

connect(s, (struct sockaddr *)&server, sizeof
  (server));
```

Connecting Sockets

- `server` would contain either the UNIX pathname, Internet address and port number of the server to which the client process wishes to speak.
- If the client process’s socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary

Connecting Sockets

- An error is returned if the connection was unsuccessful (any name automatically bound by the system, however, remains).
- Otherwise, the socket is associated with the server and data transfer may begin.

Server Side

- For the server to receive a client’s connection it must perform two steps after binding its socket.

```
listen(s, 5);
```

- Means that the server is willing to listen for incoming connection requests
- The second parameter specifies the maximum number of outstanding connections which may be queued awaiting acceptance

Server Side

- A server may accept a connection:

```
struct sockaddr_in from;

...

fromlen = sizeof (from);

newsock = accept(s, (struct sockaddr *)&from,
  &fromlen);
```

Server Side

- A new descriptor is returned on receipt of a connection (along with a new socket).
- `fromlen`:
  - input; how much space is associated with from
  - output: size of the name
- The second parameter may be a null pointer.

Sockets

- Accept will not return until a connection is available or the system call is interrupted by a signal to the process.
- Further, there is no way for a process to indicate it will accept connections from only a specific source

Data Transfer

- Normal *read* and *write* system calls are usable:

```
write(s, buf, sizeof (buf));

read(s, buf, sizeof (buf));
```

- But also:

```
send(s, buf, sizeof (buf), flags);

recv(s, buf, sizeof (buf), flags);
```

Data Transfer

- *send*/*recv* flags:
  - MSG_OOB send/receive *out of band* data
  - MSG_PEEK look at data without reading
  - MSG_DONTROUTE send data without routing packets
- When MSG_PEEK is specified with a recv call, any data present is returned to the user, but treated as still ‘‘unread’’.
- Next read or recv call applied to the socket will return the data previously previewed.

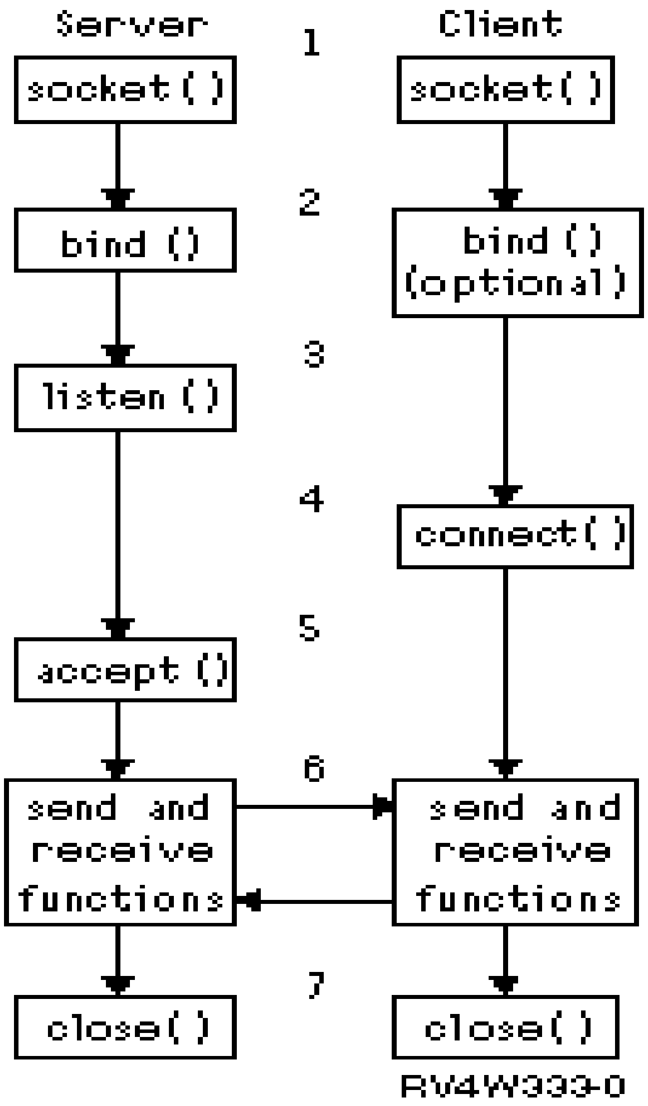
Closing

- Once a socket is no longer of interest, it may be discarded

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

- how:
  - SHUT_RD
  - SHUT_WR
  - SHUT_RDWR

Socket Calls Flow



Datagram Sockets

- `connect()` on datagram sockets returns immediately
  - The system simply records the peer's address
  - On a stream socket a connect request initiates the connection.
- Only one connected address is permitted for each socket at one time
  - a second connect will change the destination
- `accept()` and `listen()` are not used with datagram sockets.

Connectionless Sockets

- Only with datagram sockets (!)
- `sendto()` specifies a destination address

```
sts=sendto(s, buf, buflen, flags,
           (struct sockaddr *)&to, tolen);
```

  - `to` and `tolen` indicate the *address* of recipient

- `recvfrom()` receives messages on an *unconnected* datagram socket

```
sts=recvfrom(s, buf, buflen, flags,
         (struct sockaddr *)&from, &fromlen)
```

netdb

- Routines for
  - mapping host names to network addresses, network names to network numbers
  - protocol names to protocol numbers
  - service names to port numbers and the appropriate protocol
- The file `<netdb.h>` must be included

Host Names

- Internet host name to address mapping

```
struct hostent {
  char *h_name;          /* official name of host */
  char **h_aliases;      /* alias list */
  int h_addrtype;        /* host address type
                            (e.g., AF_INET) */
  int h_length;          /* length of address */
  char **h_addr_list;    /* list of addresses,
                            null terminated */

};

#define h_addr h_addr_list[0] /* first address,
                            network byte order */
```

Host Names

`gethostbyname(const char *name)`

- takes an host name and returns a hostent structure

```
gethostbyaddr(const char *addr,
    int len, int type)
```

- maps Internet host addresses (AF_INET, AF_INET6) into a hostent structure

Network Names

```
struct netent {
  char *n_name;         /* official name of net */
  char **n_aliases;     /* alias list */
  int n_addrtype;       /* net address type */
  int n_net;            /* network number,
                         host byte order */

};
```

```
getnetbyname(const char *name);

getnetbynumber(long net,
                int type);
```

Protocol Names

```
struct protoent {
  char *p_name;     /* official protocol name */
  char **p_aliases; /* alias list */
  int p_proto;      /* protocol number */

};
```

- `getprotobyname(const char *name)`
- `getprotobynumber(int proto);`

Service Names

```
struct servent {
  char *s_name;     /* official service name */
  char **s_aliases;/* alias list */
  int s_port;       /* port number,
                        network byte order */
  char *s_proto;    /* protocol to use */

};
```

- `getservbyname(const char *name,`
  `               const char *proto);`
- `getservbyport(int port, const char *proto);`

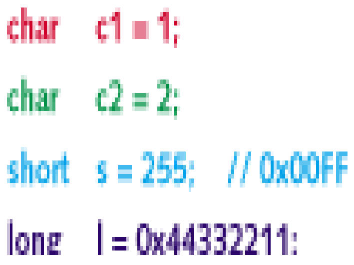
```
sp = getservbyname("telnet", (char *) 0));

sp = getservbyname("telnet", "tcp");
```

Endiannes

- Big Endian
  - the most significant byte of any multibyte data field is stored at the lowest memory address
- Little Endian
  - the least significant byte of any multibyte data field is stored at the lowest memory address

Host Independent Formats

- Intel CPUs are *Little Endian*, while the network byte order is *Big Endian*
- Macros to convert “host” order to “network byte order



```
char   c1 = 1;
char   c2 = 2;
short  s = 255;   // 0x00FF
long   l = 0x44332211;
```

```
Offset  :      Memory dump
0x0000 :      01  02  FF  00
0x0004 :      11  22  33  44
```

Network Byte Order

- `htonl(val)`
  - convert 32-bit quantity from host to network byte order
- `htons(val)`
  - convert 16-bit quantity from host to network byte order

Network Byte Order

- `ntohl(val)`
  - convert 32-bit quantity from network to host byte order
- `ntohs(val)`
  - convert 16-bit quantity from network to host byte order

Example

```
#include <stdio.h>

#include <netdb.h>

#include <stdlib.h>

unsigned long ResolveName(char name[])

{
  struct hostent *host;
  if ((host = gethostbyname(name)) == NULL){
     fprintf(stderr, "gethostbyname() failed");
     exit(1);
  }
  return *((unsigned long *)host->h_addr_list[0]);

}
```

Example

```
unsigned short ResolveService(char service[],
   char protocol[])
{
  struct servent *serv;
  unsigned short port;

  if ((port = atoi(service)) == 0) {
     if ((serv = getservbyname(service, protocol)) ==
  NULL){
        fprintf(stderr, "getservbyname() failed");
        exit(1);
     }
     else port = serv->s_port;
  }
  else port = htons(port);
  return port;
}
```

Multiplexing

- `select()` allows multiplexing i/o requests among multiple sockets and/or file*s*

select

```
#include <sys/time.h>

#include <sys/types.h>

...

fd_set readmask, writemask, exceptmask;

struct timeval timeout;

...

select(nfds, &readmask, &writemask, &exceptmask,
  &timeout);
```

select

- `select()` takes pointers to three sets
  - one for the <u>set of file descriptors</u> for which the caller wishesto be able to read data on
  - one for those descriptors to which data is to be written
  - one for which exceptional conditions are pending (out-of-band data is the only exceptional condition currently implemented bythe socket
- If the user is not interested in certain conditions the corresponding argument should be NULL.

select

- Each set is actually a structure containing an array of long integer bit masks
- The size of the array is set by the definition `FD_SETSIZE`
- The macros
  - `FD_SET(fd, &mask)`
  - `FD_CLR(fd, &mask)`

  allow adding and removing file descriptor `fd` in the set mask.

select

- The set should be zeroed before use
  - `FD_ZERO(&mask)`
- `nfds` specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) to be examined

select

- A `timeout` value may be specified
- If `timeout` (`struct timeval`) is set to `0`, select returns immediately
- If the last parameter is a `NULL` pointer, the selection will block indefinitely
  - returns only when a descriptor is selectable or when a signal

select

- `select()`returns:
  - the number of file descriptors selected
  - `0` if the select call returns due to the timeout expiring
  - `-1` if terminated because of an error or interruption

select

- The status of a file descriptor may be tested
  - `FD_ISSET(fd, &mask)`
  - returns a non-zero value if fd is a member of the set mask, and 0 if it is not

Example

```
#include <sys/time.h>
#include <sys/types.h>
...
fd_set read_template;
struct timeval wait;
...
```

Example

```
for (;;) {
   wait.tv_sec = 1; /* one second */
   wait.tv_usec = 0;
   FD_ZERO(&read_template);
   FD_SET(s1, &read_template);
   FD_SET(s2, &read_template);
   nb = select(FD_SETSIZE, &read_template,
     (fd_set *) 0, (fd_set *) 0, &wait);
```

Example

```
   if (nb <= 0) {
      if (nb<0) perror(“select”)
      else printf(“Timeout.\n);
      continue;
   }
   if (FD_ISSET(s1, &read_template)) {
      sts=ReadDataFromSocket(s1)
   }
   if (FD_ISSET(s2, &read_template)) {
      sts=ReadDataFromSocket(s2)
   }
}
```

select

- `select()` provides a synchronous multiplexing scheme.
- Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of signals (SIGIO and SIGURG)

Closing Sockets

```
s=connect(...);
if( fork() ) {   /*  Child  */
     while( gets(buffer) >0)  write(s,buf,strlen(buffer));
     close(s);
     exit(0);
}
else {    /*  Parent  */
     while( (l=read(s,buffer,sizeof(buffer)) do_something(l,buffer);
     wait(0);
     exit(0);
}
```

Socket Shutdown

```
s=connect(...);
if( fork() ) {   /*  Child  */
      while( gets(buffer) >0)  write(s,buf,strlen(buffer));
      close(s);
      shutdown(s,SHUT_WR);
      exit(0);
}
else {    /*  Parent  */
      while( (l=read(s,buffer,sizeof(buffer)) do_something(l,buffer);
      wait(0);
      exit(0);
}
```

setsockopt/getsockopt

```
int setsockopt(int s, int level, int optname,
  const void *optval, int optlen);
```

```
int getsockopt(int s, int level, int optname,
  void *optval, socklen_t *optlen)M
```

- Manipulate the options associated with a socket.
- Options may exist at multiple protocol levels; they are always present at the uppermost socket level (SOL_SOCKET)

setsockopt/getsockopt

- A server waits 2 MSL (maximum segment lifetime) for old connection.
- If not properly terminated, a further `bind()` will return EADDRINUSE.

setsockopt/getsockopt

- Before `bind()`:

```
int opt=1;

setsockopt(s, SOL_SOCKET,SO_REUSEADDR,
   (char *)&opt, sizeof(opt));
```

- Other options:
    - `SO_ERROR` get error status
    - `SO_KEEPALIVE` send periodic keep-alives
    - `SO_LINGER` *close() on non-empty buffer*
    - `SO_SNDBUF` send buffer size
    - `SO_RCVBUF` receive buffer size

Non Blocking I/O

- Once a socket has been created it may be marked as non-blocking

```
#include <fcntl.h>
  ...
int s;
  ...
s = socket(AF_INET, SOCK_STREAM, 0);
  ...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
  perror("fcntl F_SETFL, FNDELAY");
exit(1)
```

NonBlocking I/O

- NB: must check for `errno==EWOULDBLOCK`
- If an operation, such as a send, cannot be done in its entirety the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent

SIGIO

- Allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read
- Three steps:
  - set up a SIGIO signal handler
  - set the process id (or process group id) which is to receive notification of pending input to itself
  - enable asynchronous notification of pending I/O (another `fcntl()` call)

Example

```
#include <fcntl.h>
  ...
int io_handler();
  ...
signal(SIGIO, io_handler);
if (fcntl(s, F_SETOWN, getpid()) < 0) {
  perror("fcntl F_SETOWN");
  exit(1);
}
if (fcntl(s, F_SETFL, FASYNC) < 0) {
  perror("fcntl F_SETFL, FASYNC");
  exit(1);
}
```

Sockets & Signals

- When a signal is sent to a process while performing a sockets function, several things may occur depending on whether the socket function is defined as a slow function.
- A *slow* function is a function that can block **indefinitely:**
  - *write(), recv(), send(), recvfrom(), recvmsg(), sendmsg()*, *accept()*.
  - All other sockets functions are *fast*

Sockets & Signals

- Fast functions are not interrupted by a signal
- The signal is raised when these socket functions exit.

Sockets & Signals

- Slow functions are interrupted by a signal if they are blocked waiting for IO (if they are processing IO, they are not interrupted).
  - They are interrupted in the middle of processing by the raising of a signal.
  - They stop what processing they are doing and return the error EINTR.
  - They do not complete the IO that was initiated.
  - The user program must re-initiate any desired IO explicitly.

Sockets & Signals

- There are three signals that can be generated by actions on a socket:
  - SIGPIPE
  - SIGURG
  - SIGIO

Sockets & Signals

- A SIGPIPE is generated when a *send()*/*write()* operation is attempted on a broken socket.
  - E.g. a socket which has been *shutdown()*.
- The default action is to terminate the process.
- The target of the signal is the process attempting the *send()/write()*.

Sockets & Signals

- SIGIO is somewhat more complex to set up :
  - fcntl(..,F_SETFL,FASYNC) to enable Async. I/O
  - fnctl(..,F_SETOWN, pid) to set target process (group) id.
- A SIGIO signal is generated whenever new I/O can complete on a socket

Sockets & Signals

- SIGIO signal is generated when
  - new data arrives at the socket
  - data can again be sent on the socket
  - the socket is either partially or completely shutdown or when
  - a listen socket has a connection request posted on it
  - ...

Sockets & Signals

- A SIGURG indicates that an urgent condition is present on a socket.
- Either the arrival of *out of band* data or the presence of control status information
  - fnctl(..,F_SETOWN, pid) to set target process (group) id.