Inter-Process Communication in Unix (Internet Domain)

Dheeraj Sanghi
Department of Computer Science & Engineering
Indian Institute of Technology
Kanpur - 208 016 (UP)
E-mail: dheeraj@iitk.ernet.in

References

- S. J. Leffler, *et. al.*, An Advanced 4.3BSD Interprocess Communication Tutorial. November, 1988.
- W. Richard Stevens, *Unix Network Programming.* Prentice Hall, New Jersey, USA, 1990.
- On-line manual pages of related system calls, and library functions.

Outline

- Communication Domains, Sockets
- System Calls
- Connection Establishment
- Connectionless Sockets
- Network Library Routines
- Socket Options
- Non-blocking Sockets
- I/O Multiplexing, Inetd Server

Communication Domains

AF_UNIX	Unix Domain
AF_INET	Internet Domain
AF_NS	Xerox Protocols Domain

Department of Computer Science and Engineering, IIT Kanpur, INDIA

Socket

Socket is endpoint of communication, and has:

- a type
- a bound name (port number)
- an associated process

Socket Types

SOCK_STREAM	Stream	reliable, no record boundaries
SOCK_DGRAM	Datagram	unreliable, record boundaries
SOCK_RAW	Raw	Access underlying protocol directly
SOCK_SEQPACKET	Sequenced	reliable, record boundaries

Socket Creation

To create a socket,

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

Protocol '0' means use of default protocol.

Socket Address

Socket address structure (defined in `<sys/socket.h>`):

```
struct sockaddr {
    u_short sa_family;
            /* address family; AF_xxx */
    char    sa_data[14];
            /* protocol specific address */
};
```

Socket Address (contd)

For Internet domain, structures defined in `<netinet/in.h>`:

```
struct in_addr {
  u_long s_addr;        /* 32-bit netid/hostid */
};


struct sockaddr_in {
  short sin_family;     /* AF_INET */
  u_short sin_port;     /* 16-bit port number */
  struct in_addr sin_addr;
                        /* 32-bit netid/hostid */
  char sin_zero[8];     /* unused */
};
```

Binding Local Names

To bind a name to a socket,

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

In the Internet domain,

```
...
struct sockaddr_in sin;
...
/* assign port number, address */
...
bind(s,(struct sockaddr *) &sin, sizeof(sin));
```

Client-Server Architecture

The two communicating processes have different roles.

Server:

Does a *passive* open.
Waits for other processes to send a connection request.

Client:

Does an *active* open.
Sends a request for connection to the server.

Connection Establishment

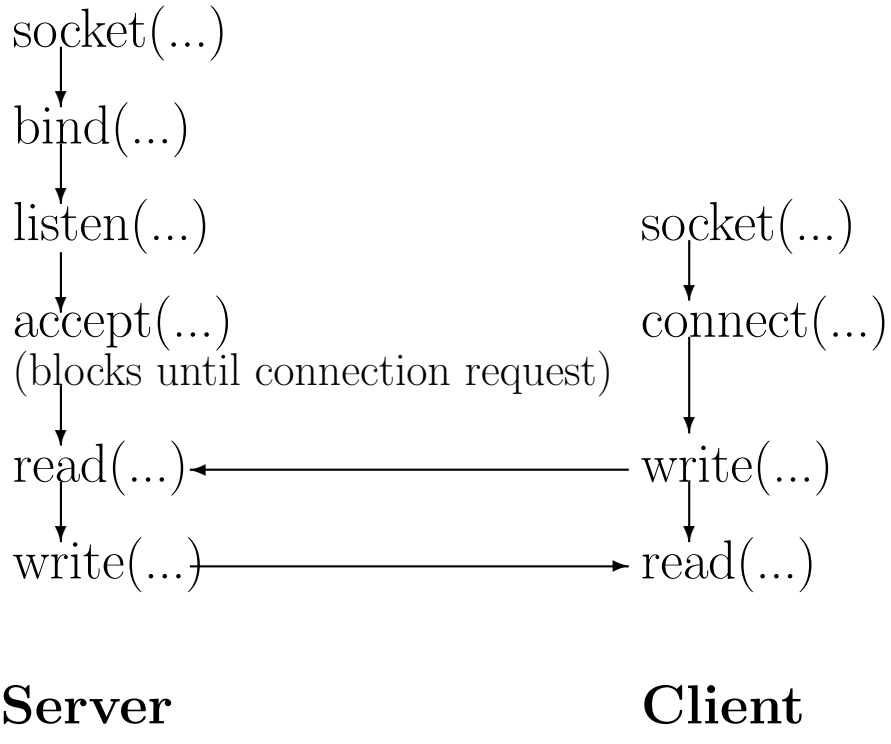
Server:

```
struct sockaddr_in from;
...
fromlen = sizeof (from);
listen (s, n);
news = accept (s, (struct sockaddr *) &from,
                  &fromlen);
```

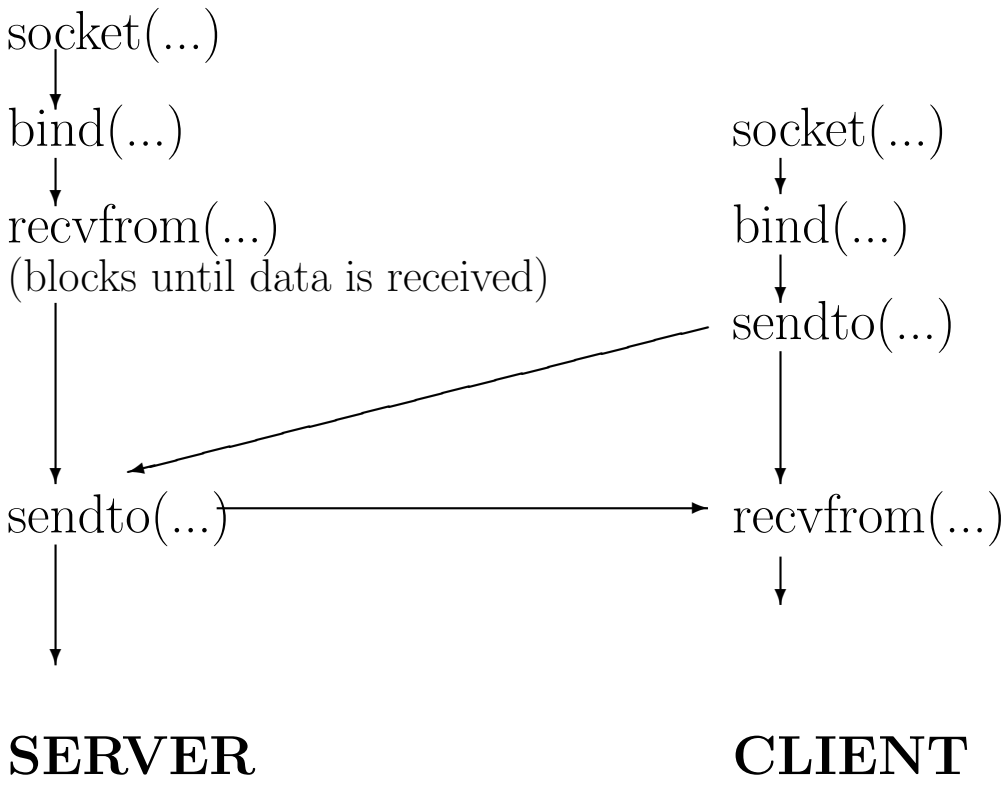
Client:

```
struct sockaddr_in server;
...
/* set remote port, address in 'server' */
...
connect (s, (struct sockaddr *) &server,
                  sizeof(server));
```

System Calls for Connection-oriented Protocols



System Calls for Connectionless Protocols



Data Transfer

Normal read and write system calls are usable.

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

In addition, there are new system calls.

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

flags are defined in `<sys/socket.h>`.

Connectionless Sockets

Each message:

- can be sent to, or received from a different process.
- would have source/destination address.

The system calls used are:

```
recvfrom(s, buf, buflen, flags,
        (struct sockaddr *) &from, &fromlen);
sendto  (s, buf, buflen, flags,
        (struct sockaddr *) &to, tolen);
```

Connectionless sockets can use connect to communicate with a specific process.

Discarding Sockets

The system calls used are:

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

The value of `how` can be:

0	process will not read any more data
1	process will not send any more data
2	process will neither read nor write

Byte Ordering Routines

`htonl` convert host-to-network, long integer

`htons` convert host-to-network, short integer

`ntohl` convert network-to-host, long integer

`ntohs` convert network-to-host, short integer

Byte Operations

```
char *src, *dest, *ptr1, *ptr2;
...
bcopy (src, dest, nbytes);
bzero (dest, nbytes);
bcmp (ptr1, ptr2, nbytes);
```

`bcopy` copies from one string to another

`bzero` writes null bytes to the string

`bcmp` compares two strings

Host Names and Addresses

Following library routines are available:

```
gethostname (char *name, namelen);
unsigned long gethostid();
struct hostent *gethostbyname(char *hostname);
struct hostent *gethostbyaddr(char *addr,
                              int len, int type);
```

Address Conversion Routine

Conversion routines between dotted-decimal format, and an `in_addr` structure.

```
unsigned long inet_addr(char *ptr);
char *inet_ntoa(struct in_addr inaddr);
```

Host Entry Structure

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

#define h_addr h_addr_list[0]
                       /* first address */
```

Protocol Names

```
struct protoent getprotobyname (char *name);
struct protoent getprotobynumber (int proto);
```

The protocol entry structure looks like:

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

Service Names

```
struct servent getservbyname
            (char *name,char *proto);
struct servent getservbyport
            (int port,char *proto);
```

The service entry structure looks like:

```
struct servent {
  char *s_name;       /* official service */
  char **s_aliases;   /* alias list */
  int s_port;         /* port number */
  char *s_proto;      /* protocol to use */
};
```

Reserved Ports

System reserved ports	1-1023
ports assigned by `rresvport()`	512-1023
user-reserved ports	5001-FFFF
ports assigned by system	1024-5000

Socket Options

```
getsockopt(s, level, optname, optval, optlen);
setsockopt(s, level, optname, optval, optlen);
```

Various levels in Internet domain are:

- IPPROTO_IP
- IPPROTO_TCP
- SOL_SOCKET

Examples of Socket Options

IPPROTO_IP	IP_OPTIONS	options in IP header
IPPROTO_TCP	TCP_NODELAY	don't wait for more data
SOL_SOCKET	SO_DEBUG	turn on debugging info
	SO_RCVBUF	receive buffer size
	SO_SNDBUF	send buffer size
	SO_ERROR	get error status

Non-Blocking Sockets

```
fcntl (s, F_SETFL, FNDELAY);
```

Error EWOULDBLOCK would be returned if any system call would have normally blocked.

Affects the system calls:
`read, write, send, recv, accept, connect`

Interrupt Driven Socket I/O

Same procedure as interrupt-driven I/O on any file.

```
signal(SIGIO, io_handler);
/* set a signal handler */

fcntl (s, F_SETOWN, getpid());
/* set process-id of socket */

fcntl (s, F_SETFL, FASYNC);
/* enables async notification */
```

Input/Output Multiplexing

Different techniques:

- Make sockets non-blocking.
  Do polling.
- Fork a separate process for each I/O channel.
  Send a message to parent when ready for I/O.
- Asynchronous I/O.
- `select` system call.

Select System Call

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

Declare three sets of file descriptors:

- one, on which we are waiting for read.
- second, on which we are waiting to write.
- third, on which some exception conditions may occur.

Select waits until one of the descriptors is ready for I/O,
but not beyond the timeout value.
If timeout is a null pointer, then it is indefinite wait.

Some Useful Macros with Select

```
FD_ZERO(fd_set *fdset);
/* clear all bit in fdset */

FD_SET (int fd, fd_set *fdset);
/* turn bit for fd on in fdset */

FD_CLR (int fd, fd_set *fdset);
/* turn bit for fd off in fdset */

FD_ISSET(int fd, fd_set *fdset);
/* test bit for fd in fdset */
```

Iterative v/s Concurrent Servers

Two approaches to server design:

Iterative server: Accept a request, process it, then accept next request.

Concurrent server: Accept a request, fork a child to process it. Parent immediately ready to accept next request.

Program Example
A Remote Time Utility

- TCP Based.
- Server port number is hard-coded
- Server sends absolute time (seconds elapsed since Jan. 1, 1970)

```
/* rtime.c - client program */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <netdb.h>
#include <time.h>
#include "time.h"
main(int argc, char *argv[])
{
  struct sockaddr_in cli_addr, serv_addr;
  struct hostent *server;
  time_t t, recd_data;
  int s;
  if (argc != 2){
    fprintf (stderr, "Usage:%s <host>\n",
             argv[0]);
    exit(1);
  }
```

```
if ((s= socket(AF_INET,SOCK_STREAM,0))== -1){
      perror("socket");
      exit(1);
}

bzero(&cli_addr, sizeof(cli_addr));
cli_addr.sin_family = AF_INET;
cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
cli_addr.sin_port = htons(0);

if (bind(s, (struct sockaddr *) &cli_addr,
        sizeof(cli_addr)) == -1) {
      perror("bind");
      exit(1);
}

if ((server=gethostbyname(argv[1]))==NULL){
      perror("gethostbyname");
      exit(1);
}
```

```
bzero (&serv_addr, sizeof (serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = *((ulong *)
            server->h_addr);
serv_addr.sin_port = htons (SERVER_PORT);
if (connect(s,(struct sockaddr *)&serv_addr,
            sizeof (serv_addr)) == -1) {
   perror("connect");
   exit(1);
}
if (read (s, &recd_data, sizeof(recd_data))
            != sizeof (recd_data)) {
   fprintf (stderr,
            "Error in communication\n");
   exit(1);
}
t = ntohl(recd_data);
printf ("Current time at %s is %s\n",
            argv[1], ctime(&t));
close(s);
}
```

```
/* timed.c - server program */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <netdb.h>
#include <sys/time.h>
#include <unistd.h>
#include "time.h"
main (int argc, char *argv[])
{
   struct sockaddr_in cli_addr, serv_addr;
   struct hostent *server;
   struct timeval t;
   int cli_addr_len, s, news;
   long data;
```

```
if ((s=socket(AF_INET,SOCK_STREAM,0))== -1){
   perror("socket");
   exit(1);
}

bzero(&serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr= htonl(INADDR_ANY);
serv_addr.sin_port = htons(SERVER_PORT);

if (bind(s, (struct sockaddr *)&serv_addr,
          sizeof(serv_addr)) == -1) {
   perror("bind");
   exit(1);
}
listen(s, 5);
```

```
  while (1) {
    if ((news=accept(s,(struct sockaddr *)
          &cli_addr,&cli_addr_len))== -1){
      perror("accept");
      exit(1);
    }
    if (fork() == 0) { /* child */
      close(s);
      gettimeofday(&t, NULL);
      data = htonl(t.tv_sec);
      if (write(news, &data, sizeof(long))
            != sizeof(long))
          perror("write");
      close(news);
      exit(1);
    }
    /* parent */
    close(news);
  }
}
```

Inetd - Internet Super-server

- Invoked at boot time
- File /etc/inetd.conf has the list of servers
- Inetd creates one socket for each service
- Bind the appropriate port number to each socket
- Perform select on all sockets for read
- When connection arrives, do accept on that socket
- Fork the appropriate server

Inetd (contd)

Advantages:

- Single process waits to service multiple requests. Reduces the total number of processes in the system.
- Simplifies the writing of daemon processes, by taking care of all IPC details.

Disadvantages:

- Inetd has to execute both a `fork` and an `exec` to start the right server. A regular server only executes a `fork`.

Overheads are minimal compared to advantages.

Finding Remote Address

Servers through Inetd do not execute `accept` system call.
Need other mechanisms to find out about remote processes.

```
getpeername (s,(struct sockaddr *)&name,&namelen);
```

Similarly, to find local address,

```
getsockname (s,(struct sockaddr *)&name,&namelen);
```