COMPUTER NETWORKS
UNIT – III

Network Programming

Socket Interface: Sockets, Socket Address, Elementary Sockets, Advanced Sockets, Socket Options, Out of Band Data, Daemon process and Internet Super Server, IPv4 and IPv6 interoperability. Remote Procedure Calls: Introduction, Transparency Issues and Sun RPC.

Network Programming

It makes such a program in which the machines connected in network and will send and receive data from other machine in the network by programming. When the computer needs to communicate with another computer, it's required the other computer's address.

Network programming can be done using various other APIs. Network programming is done using sockets directly or using various other layers on top of sockets (e.g. HTTP which is normally implemented with TCP over sockets). TCP/IP and UDP/IP are done primarily via the sockets interface.

The java.net package provides support for the two common network protocols:
TCP: Transfer Control Protocol, which allows for reliable communication between two applications.
UDP: User Datagram Protocol, a connection-less Protocol that allows for pocket of data to be transmitted between applications

SOCKETS

It is a port / an endpoint of communication is used as a low-level interface to network protocols. A socket has a type and one associated process. Sockets are used to implement the client-server model for inter-process communication. The interface to network protocols needs to accommodate multiple communication protocols, such as TCP/IP, Xerox internet protocols (XNS), and UNIX family.

The interface to network protocols needs to accommodate server code that waits for connections and client code that initiates connections. It also needs to operate differently, depending on whether communication is connection-oriented or connectionless.Application programs might want to specify the destination address of the datagrams it delivers instead of binding the address with the open(2) call.

SOCKET ADDRESS

1. Process to process delivery (transport layer communication) needs two identifiers, one is IP address and the other is port number at each end to make a connection.
2. Socket address is the combinations of IP address and port number as shown in the figure.

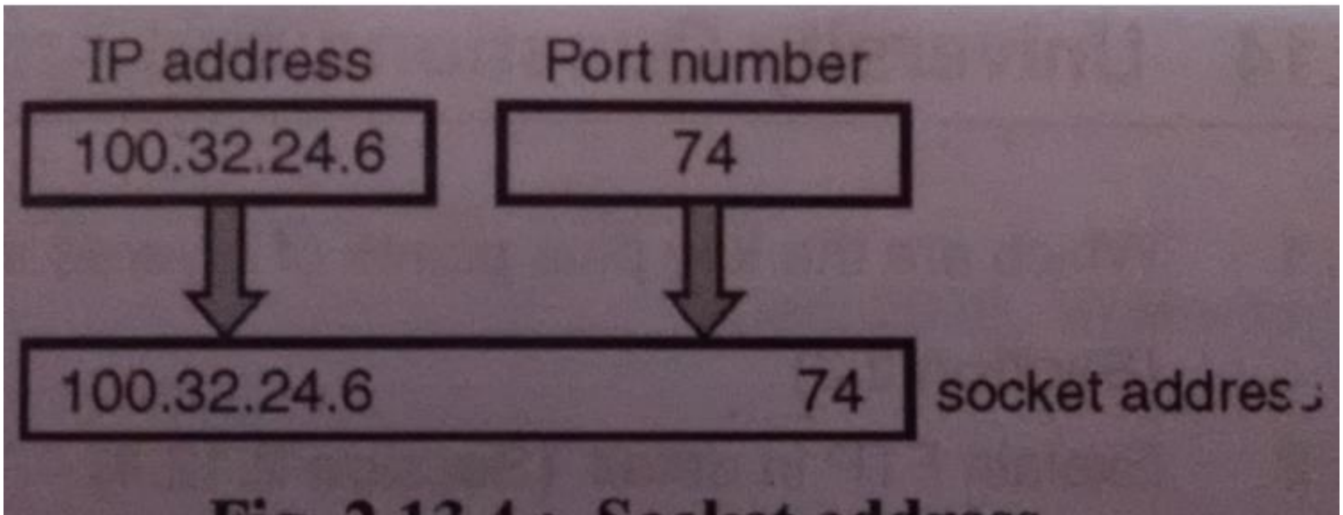
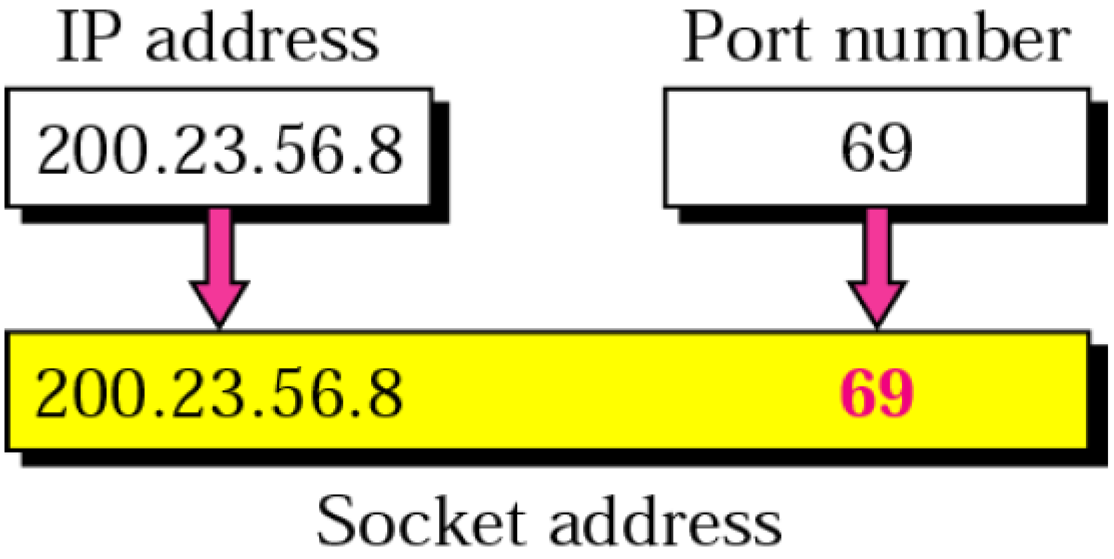


Fig. 2.13.4 : Socket address

1. The client socket address defines the client process uniquely whereas the server socket address defines the server process uniquely.
2. A transport layer protocol requires the client socket address as well as the server socket address. These two addresses contain four pieces.
3. These four pieces go into the IP header and the transport layer protocol header.
4. The IP header contains the IP addresses while the UDP and TCP headers contain the port numbers.

- A transport-layer protocol in the TCP suite needs both the IP address and the port number, at each end, to make a connection. The combination of an IP address and a port number is called a socket address.
- The client socket address defines the client process uniquely just as the server socket address defines the server process uniquely as shown in Figure.



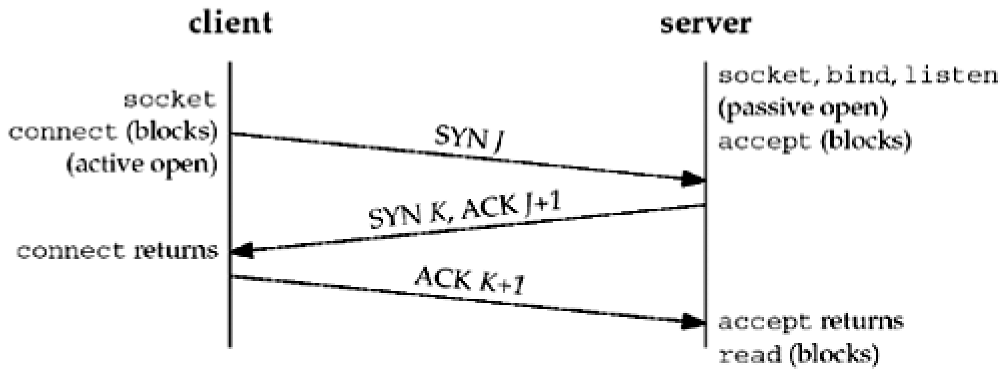
- To use the services of transport layer in the Internet, we need a pair of socket addresses: the **client socket address** and the **server socket address**.
- These four pieces of information are part of the network-layer packet header and the transport-layer packet header. The first header contains the IP addresses; the second header contains the port numbers.

ELEMENTARY TCP SOCKETS

1. socket function	**6. fork and exec functions**
2. connect function	**7. Concurrent servers**
3. bind function	**8. close function**
4. listen function	**9. getsockname and getpeername** functions , or complicated terms.
5. accept function	

TCP Three-Way Handshake

To understand the **connect, accept and close** functions and to debug TCP application using neststat we need to follow the state transition diagram.



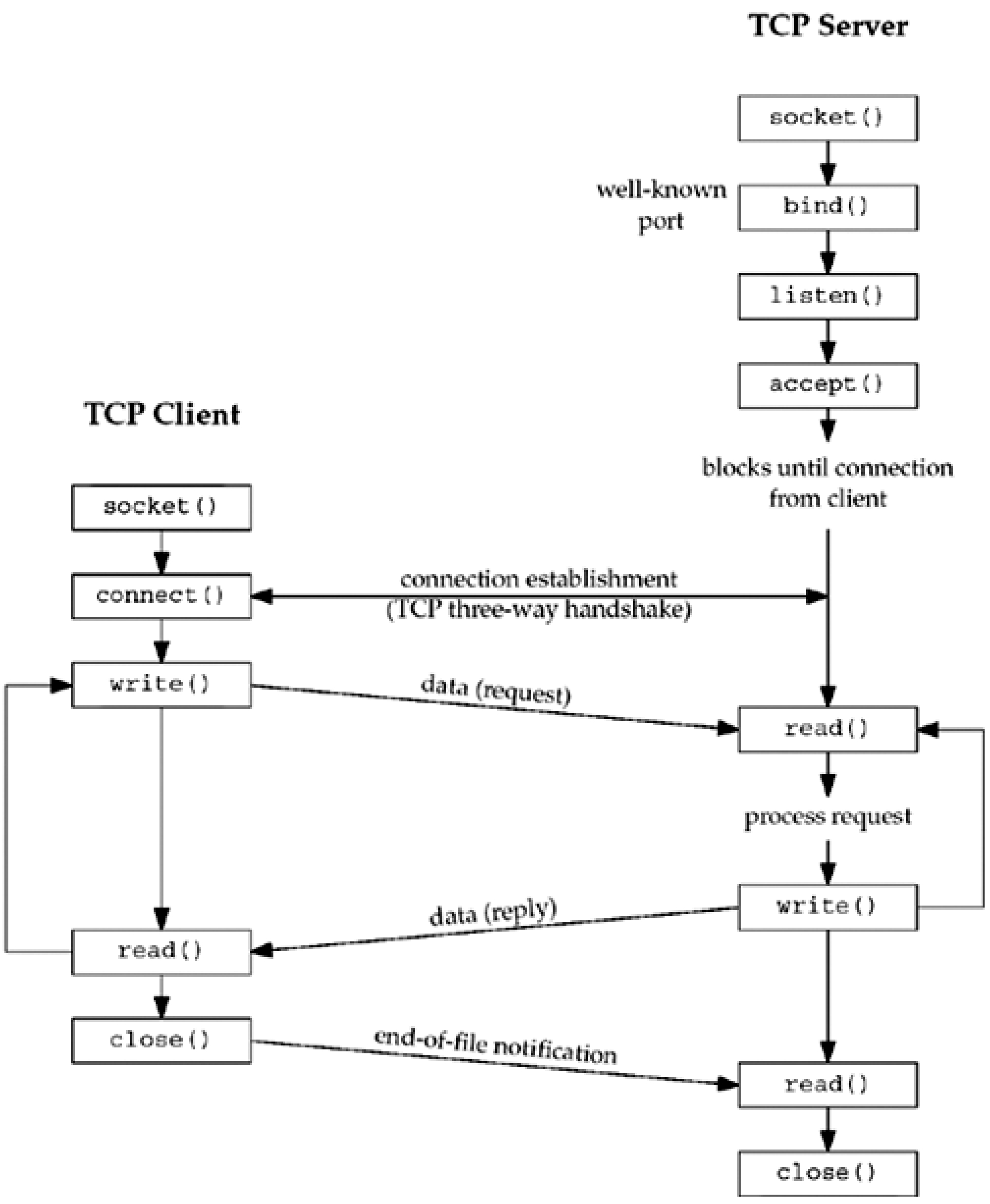
The **three way handshake that** takes place is as follows:

1. The server must be prepared to accept an incoming connection. This is normally done by calling socket, bind and listen functions and is called passive open.
2. The client issues an active open by calling connect. This causes the client TCP to send SYN segment to tell the server that the client‛s initial sequence number for the data that the client will send on that connection. No data is sent with SYN. It contains an IP header, TCP header and possible TCP options.
3. The server acknowledges the client‛s SYN and sends its own SYN and the ACK of the client‛s SYN in a single segment.
4. The client must ACK the server‛s SYN.

System calls used with sockets:

Socket calls are those functions that provide access to the underlying functionality and utility routines that help the programmer. A socket can be used by client or by a server, for a stream transfer (TCP) or datagram (UDP) communication with a specific endpoints address.

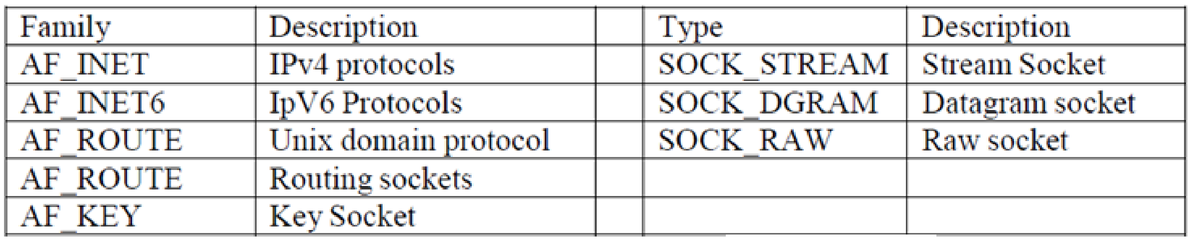
Following figure shows a time line of the typical scenario that takes place between client and server.



First server is started, then sometimes later a client is started that connects to the server. The client sends a request to the server, the server processes the request, and the server sends back reply to the client. This continues until the client closes its end of the connection, which sends an end of file notification to the server. The server then closes its end of the connections and either terminates or waits for a new connection.

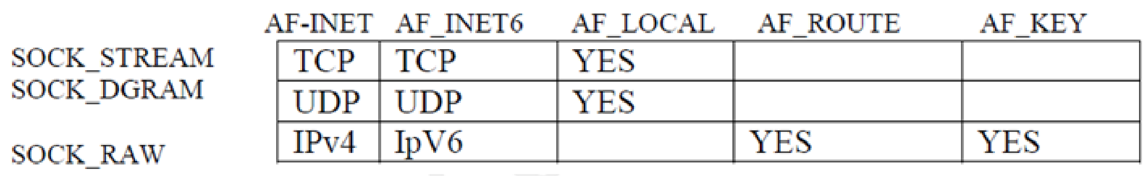
Socket function:
#include socket (int family, int type, int protocol);
returns negative descriptor if OK & –1 on error.
Arguments specify the protocol family and the protocol or type of service it needs (stream or datagram).

The protocol argument is set to 0 except for raw sockets.



Family	Description		Type	Description
AF_INET	IPv4 protocols		SOCK_STREAM	Stream Socket
AF_INET6	IpV6 Protocols		SOCK_DGRAM	Datagram socket
AF_ROUTE	Unix domain protocol		SOCK_RAW	Raw socket
AF_ROUTE	Routing sockets			
AF_KEY	Key Socket			

Not all combinations of socket family and type are valid. Following figure shows the valid combination.



	AF-INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	YES		
SOCK_DGRAM	UDP	UDP	YES		
SOCK_RAW	IPv4	IpV6		YES	YES

Connect Function
The connect function is by a TCP client to establish an active connection with a remote server. The arguments allow the client to specify the remote end points which includes the remote machines IP address and protocol port number.

```
# include <sys/socket.h>
int connect (int sockfd, const struct sockaddr * servaddr, socklen_t addrelen)
returns 0 if ok -1 on error.
```

sockfd is the socket descriptor that was returned by the socket function. The second and third arguments are a pointer to a socket address structure and its size.

In case of TCP socket, the connect() function initiates TCP's three way handshake. The function returns only when the connection is established or an error occurs. Different type of errors are
1. If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned. This is done after the SYN is sent after, 6sec, 24sec and if no response is received after a total period of 75 seconds, the error is returned.
2. In case for SYN request, a RST is returned (hard error), this indicates that no process is waiting for connection on the server. In this case ECONNREFUSED is returned to the client as soon the RST is received. RST is received when (a) a SYN arrives for a port that has no listening server (b) when TCP wants to abort an existing connection, (c) when TCP receives a segment for a connection does not exist.
3. If the SYN elicits an ICMP destination is unreachable from some intermediate router, this is considered a soft error. The client server saves the message but keeps sending SYN for the time period of 75 seconds. If no response is received, ICMP error is returned as EHOSTUNREACH or ENETUNREACH.

bind() Function:
When a socket is created, it does not have any notion of end points addresses An application calls bind to specify the local endpoint address for a socket. That is the bind function assigns a local port and address to a socket.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

Returns: 0 if OK,-1 on error

The second argument is a pointer to a protocol specific address and the third argument is the size of this address structure. Server binds their well-known port when they start. (A TCP client does not bind an IP address to its socket.)

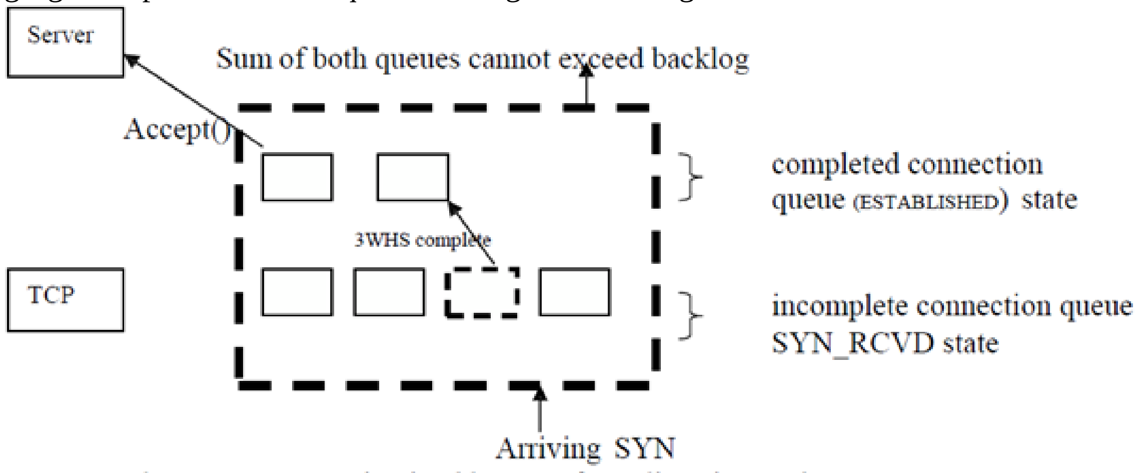
listen Function:
The listen function is called only by TCP server and it performs following functions. The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. In terms of TCP transmission diagram the call to listen moves the socket from the CLOSED state to the LISTEN state.

```
#include <sys/socket.h>
#int listen (int sockfd, int backlog);
Returns: 0 if OK, -1 on error
```

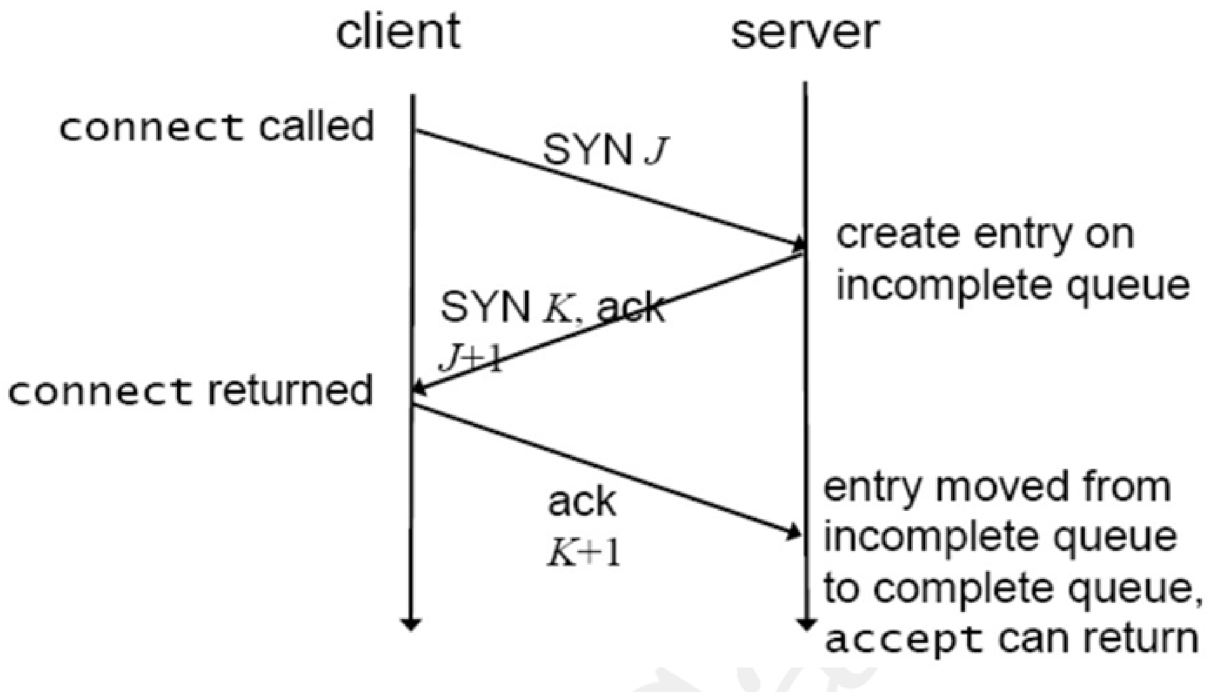
•The second argument to this function specifies the maximum number of connections that the kernel should queue for this socket.
•This function is normally called after both the socket and bind functions and must be called before calling the accept function.

The kernel maintains two queues and the backlog is the sum of these two queues
•An Incomplete Connection Queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three way handshakes. These sockets are in the SYN_RECD state.
•A Completed Connection Queue which contains an entry for each client with whom three handshakes has completed. These sockets are in the ESTABLISHED state.

Following figure depicts these two queues for a given listening socket.


Server
Sum of both queues cannot exceed backlog
Accept()
completed connection queue (ESTABLISHED) state
3WHS complete
TCP
incomplete connection queue SYN_RCVD state
Arriving SYN
The two queues maintained by TCP for a listening socket.

When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three way handshake. The server‘s SYN with an ACK of the clients SYN. This entry will remain on the incomplete queue until the third segment of the three way handshake arrives (the client‘s ACK of the server‘s SYN) or the entry times out. If the three way hand shake completes normally, the entry moves from the incomplete queue to the completed queue. When the process calls accept, the first entry on the completed queue is returned to the process or, if the queue is empty, the process is put to sleep until an entry is placed onto the completed queue. If the queue are full when a client arrives, TCP ignores the arriving SYN, it does not send an RST. This is because the condition is considered temporary and the client TCP will retransmit its SYN with the hope of finding room in the queue.


client
server
connect called
SYN J
create entry on incomplete queue
SYN K, ack J+1
connect returned
ack K+1
entry moved from incomplete queue to complete queue, accept can return

accept Function
accept is called by a TCP server to return the next completed connection from the from of the completed connection queue. If the completed queue is empty, the process is put to sleep.

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
Returns: non-negative descriptor if OK, -1 on error
```

 The cliaddr and addrlen arguments are used to return the protocol address of the connected peer process (the client).
 addrlen is a value-result argument before the call, we set the integer value pointed to by *addrlen to the size of the socket address structure pointed to by cliaddr and on return this integer value contains the actual number of bytes stored by the kernel in the socket address structure. If accept is successful, its return value is a brand new descriptor that was automatically created by the kernel. This new descriptor refers to the TCP connection with the client. When discussing accept we call the first argument to accept the listening and we call the return value from a accept the connected socket.

fork function
fork is the function that enables the Unix to create a new process

```
#include <unistd.h>
pid_t fork(void);
Returns: 0 in child, process ID of child in parent, -1 on error
```

Typical uses of fork function:
1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is normal way of working in a network servers.
2. A process wants to execute another program. Since the only way to create a new process is by calling fork, the process first calls fork to make a copy of itself, and then one of the copies(typically the child process) calls exec function to replace itself with a the new program. This is typical for program such as shells.
3. fork function although called once, it returns twice. It returns once in the calling process (called the parent) with a return value that is process ID of the newly created process (the child). It also returns once in the child, with a return value of 0. Hence the return value tells the process whether it is the parent or the child.
4. The reason fork returns 0 in the child, instead of parent‘s process ID is because a child has only one

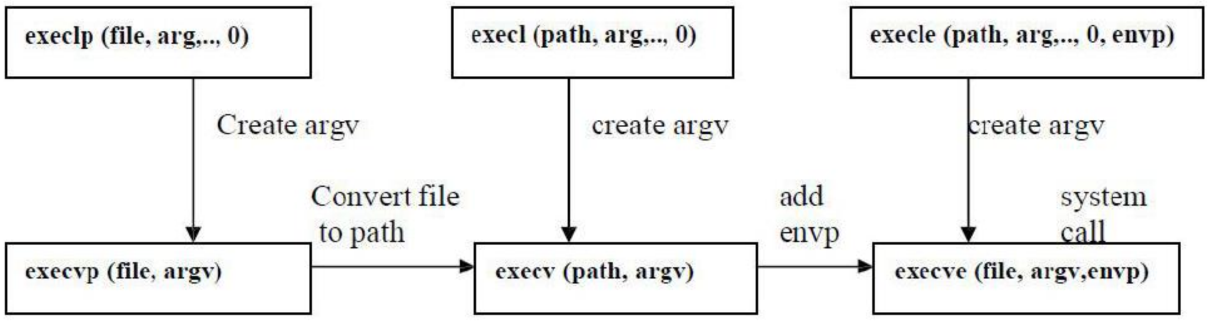
parent and it can always obtain the parent‘s process ID by calling getppid A parent, on the other hand, can have any number of children, and there is no way to obtain the process Ids of its children. If the parent wants to keep track of the process Ids of all its children, it must record the return values form fork

exec function

The only way in which an executable program file on disk is executed by Unix is for an existing process to call one of the six exec functions. exec replaces the current process image with the new program file and this new program normally starts at the main function. The process ID does not change. The process that calls the exec is the calling process and the newly executed program as the new program.

```
#include <unistd.h>
int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ );
int execv (const char *pathname, char *const argv[]);
int execle (const char *pathname, const char *arg0, ...
/* (char *) 0, char *const envp[] */ );
int execve (const char *pathname, char *const argv[], char *const envp[]);
int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */ );
int execvp (const char *filename, char *const argv[]);
```

All six return: -1 on error, no return on success



1. The three functions in the top row specify each argument string as a separate argument to the exec function, with a null pointer terminating the variable number of arguments. The three functions in the second row have an argv array, containing pointers to the argument strings. This argv array must contain a null pointer to specify its end, since a count is not specified.
2. The two functions in the left column specify a filename argument. This is converted into a pathname using the current PATH environment variable. If the filename argument to execlp or execvp contains a slash (/) anywhere in the string, the PATH variable is not used. The four functions in the right two columns specify a fully qualified pathname argument.
3. The four functions in the left two columns do not specify an explicit environment pointer. Instead, the current value of the external variable environ is used for building an environment list that is passed to the new program. The two functions in the right column specify an explicit environment list. The envp array of pointers must be terminated by a null pointer.

Close function

The normal Unix close function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>
int close (int sockfd);
```

Returns: 0 if OK, -1 on error

The default action of close with a TCP socket is to mark the socket as closed and return to the process immediately. The socket descriptor is no longer usable by the process. That is , it cannot be used as an argument to read or write.

getsockname () and getpeername():
These two functions return either the local protocol address associated with a socket or the foreign address associated with a socket.

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
Both return: 0 if OK, -1 on error
```

These functions are required for the following reasons.
1. After connect successfully returns a TCP client that does not call bind(), getsocketname() returns the local IP address and local port number assigned to the connection by the kernel
2. After calling bind with a port number of 0, getsockname() returns the local port number that was assigned
3. When the server is exceed by the process that calls accept(), the only way the server can obtain the identity of the client is to call getpeername().

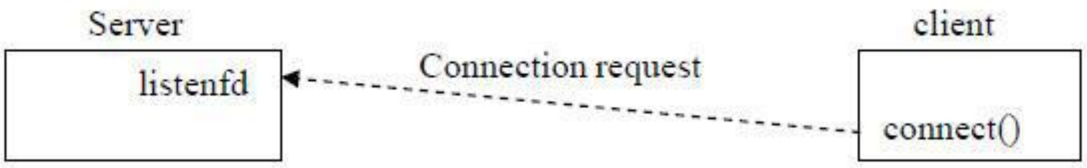
Concurrent Servers
A server that handles a simple program such as daytime server is a iterative server. But when the client request can take longer to service, the server should not be tied upto a single client. The server must be capable of handling multiple clients at the same time. The simplest way to write a concurrent server under Unix is to fork a child process to handle each client.

Outline for typical concurrent server.
```
pid_t pid;
int listenfd, connfd;
listenfd = Socket( ... );
/* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);
for ( ; ; ) {
connfd = Accept (listenfd, ... ); /* probably blocks */
if( (pid = Fork()) == 0) {
Close(listenfd); /* child closes listening socket */
doit(connfd); /* process the request */
Close(connfd); /* done with this client */
exit(0); /* child terminates */
}
Close(connfd); /* parent closes connected socket */
}
```

When a connection is established , accept returns, the server calls fork, and then the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket). The parent closes the connected socket since the child handles this new client.

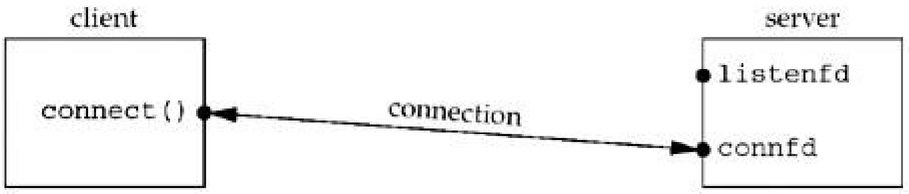
In the above program, the function doit does whatever is required to service the client. When this functions returns, we explicitly close the connected socket in the child. This is not required since the next statement calls exit, and part of process termination is closing all open descriptors by the kernal. Whether to include this explicit call to close or not is a matter of personal programming taste.
Below fig. shows the status of the client and server while the server is blocked in the call to accept and the connection request arrives from the client.

Status of client/server before call to accept returns.

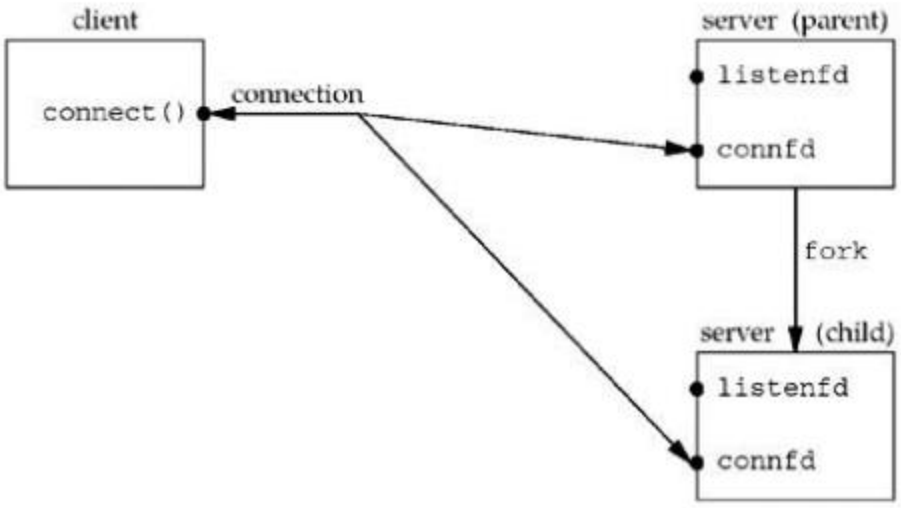


Status of client –server before call to accept().

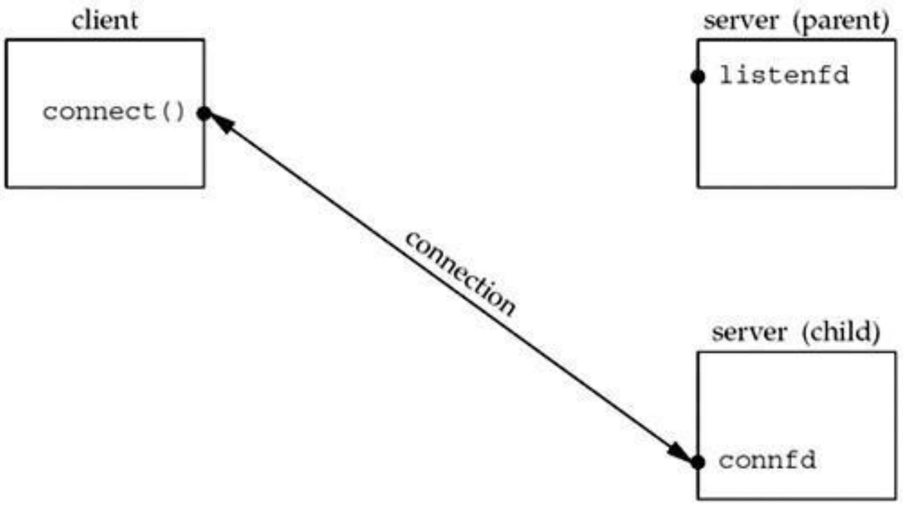
Immediately after accept returns, we have the scenario shown in Figure B. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.



The next step in the concurrent server is to call fork. Fig c Show the status after fork returns.



Notice that both descriptors, listenfd and connfd, are shared (duplicated) between the parent and child.



This is the desired final state of the sockets. The child is handling the connection with the client and the parent can call accept again on the listening socket, to handle the next client connection.

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.

Types of sockets are currently available:
1. Stream socket
Provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectional nature of the dataflow, a pair of connected stream sockets provides an interface nearly identical to that of pipes.
2. Datagram socket
Supports bidirectional flow of data that isn't guaranteed to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and possibly in an order other than the one in which they were sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet-switched networks (e.g. Ethernet).
3. Raw socket
Provides users access to the underlying communication protocols that support socket abstractions. These sockets are normally datagram-oriented, though their exact characteristics depend on the interface provided by the protocol. Raw sockets aren't intended for the general user; they've been provided mainly for anyone interested in developing new communication protocols or in gaining access to some of the more esoteric facilities of an existing protocol. Using raw sockets is discussed in the ``Advanced topics" section in this chapter.

Advanced Sockets
For most programmers, the mechanisms already described are enough to build distributed applications. Others need some of the additional features in this section.

Out-of-Band Data
The stream socket abstraction includes out-of-band data. Out-of-band data is a logically independent transmission channel between a pair of connected stream sockets. Out-of-band data is delivered independent of normal data. The out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message can contain at least one byte of data, and at least one message can be pending delivery at any time.

For communications protocols that support only in-band signaling (that is, urgent data is delivered in sequence with normal data), the message is extracted from the normal data stream and stored separately. This lets users choose between receiving the urgent data in order and receiving it out of sequence, without having to buffer the intervening data.

You can peek (with MSG_PEEK) at out-of-band data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of its existence. A process can set the process group or process ID to be informed by SIGURG with the appropriate fcntl(2) call, as described in "Interrupt-Driven Socket I/O" for SIGIO. If multiple sockets have out-of-band data waiting delivery, a select(3C) call for exceptional conditions can be used to determine the sockets with such data pending.

A logical mark is placed in the data stream at the point at which the out-of-band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal is received, all data up to the mark in the data stream is discarded.
To send an out-of-band message, the MSG_OOB flag is applied to send(3SOCKET) or sendto(3SOCKET). To receive out-of-band data, specify MSG_OOB to recvfrom(3SOCKET) or recv(3SOCKET) (unless out-of-band data is taken in line, in which case the MSG_OOB flag is not needed). The SIOCATMARK ioctl(2) tells whether the read pointer currently points at the mark in the data stream:

```
int yes;
```

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is 1 on return, the next read returns data after the mark. Otherwise, assuming out-of-band data has arrived, the next read provides data sent by the client before sending the out-of-band signal. The routine in the remote login process that flushes output on receipt of an interrupt or quit signal is shown in Example 2-12. This code reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

A process can also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (for example, TCP, the protocol used to provide socket streams in the Internet family). With such protocols, the out-of-band byte might not yet have arrived when a recv(3SOCKET) is done with the MSG_OOB flag. In that case, the call returns the error of EWOULDBLOCK. Also, there might be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data before the urgent data can be delivered.

Example 2-12 Flushing Terminal I/O on Receipt of Out-of-Band Data

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
        int out = FWRITE;
        char waste[BUFSIZ];
        int mark = 0;
        /* flush local terminal output */
        ioctl(1, TIOCFLUSH, (char *) &out);
        while(1) {
            if (ioctl(rem, SIOCATMARK, &mark) == -1) {
                perror("ioctl");
                break;
            }
            if (mark)
                break;
            (void) read(rem, waste, sizeof waste);
        }
        if (recv(rem, &mark, 1, MSG_OOB) == -1) {
            perror("recv");
            ...
        }
        ...
}
```

There is also a facility to retain the position of urgent in-line data in the socket stream. This is available as a socket-level option, SO_OOBINLINE. See the getsockopt(3SOCKET) manpage for usage. With this option, the position of urgent data (the mark) is retained, but the urgent data immediately follows the mark in the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

Nonblocking Sockets

Some applications require sockets that do not block. For example, requests that cannot complete immediately and would cause the process to be suspended (awaiting completion) are not executed. An error code would be returned. After a socket is created and any connection to another socket is made, it can be made nonblocking by issuing a fcntl(2) call, as shown in Example 2-13.

Example 2-13 Set Nonblocking Socket

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL, 0) == -1)
                perror("fcntl F_GETFL");
                exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY) == -1)
                perror("fcntl F_SETFL, FNDELAY");
                exit(1);
}
...
```

When doing I/O on a nonblocking socket, check for the error EWOULDBLOCK (in errno.h), which occurs when an operation would normally
block. accept(3SOCKET), connect(3SOCKET), send(3SOCKET), recv(3SOCKET), read(2), and write(2) can all return EWOULDBLOCK. If an operation such as a send(3SOCKET) cannot be done in its entirety, but partial writes work (such as when using a stream socket), the data that can be sent immediately are processed, and the return value is the amount actually sent.

Asynchronous Socket I/O

Asynchronous communication between processes is required in applications that handle multiple requests simultaneously. Asynchronous sockets must be SOCK_STREAM type. To make a socket asynchronous, you issue a fcntl(2) call, as shown in Example 2-14.

Example 2-14 Making a Socket Asynchronous

```
#include <fcntl.h>
#include <sys/file.h>
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL ) == -1)
                perror("fcntl F_GETFL");
                exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY | FASYNC) == -1)
                perror("fcntl F_SETFL, FNDELAY | FASYNC");
                exit(1);
}
...
```

After sockets are initialized, connected, and made asynchronous, communication is similar to reading and writing a file asynchronously. A send(3SOCKET), write(2), recv(3SOCKET), or read(2) initiates a data transfer. A data transfer is completed by a signal-driven I/O routine, described in the next section.

Interrupt-Driven Socket I/O

The SIGIO signal notifies a process when a socket (actually any file descriptor) has finished a data transfer. The steps in using SIGIO are:

Set up a SIGIO signal handler with thesignal(3C) or sigvec(3UCB) calls.

Use fcntl(2) to set the process ID or process group ID to route the signal to its own process ID or process group ID (the default process group of a socket is group 0).
Convert the socket to asynchronous, as shown in "Asynchronous Socket I/O".
Example 2-15 shows some sample code to allow a given process to receive information on pending requests as they occur for a socket. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

After sockets are initialized, connected, and made asynchronous, communication is similar to reading and writing a file asynchronously. A send(3SOCKET), write(2), recv(3SOCKET), or read(2) initiates a data transfer. A data transfer is completed by a signal-driven I/O routine, described in the next section.

Interrupt-Driven Socket I/O
The SIGIO signal notifies a process when a socket (actually any file descriptor) has finished a data transfer. The steps in using SIGIO are:
Set up a SIGIO signal handler with thesignal(3C) or sigvec(3UCB) calls.
Use fcntl(2) to set the process ID or process group ID to route the signal to its own process ID or process group ID (the default process group of a socket is group 0).
Convert the socket to asynchronous, as shown in "Asynchronous Socket I/O".

Example 2-15 shows some sample code to allow a given process to receive information on pending requests as they occur for a socket. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

Signals and Process Group ID
For SIGURG and SIGIO, each socket has a process number and a process group ID. These values are initialized to zero, but can be redefined at a later time with the F_SETOWN fcntl(2), as in the previous example. A positive third argument to fcntl(2) sets the socket's process ID. A negative third argument to fcntl(2) sets the socket's process group ID. The only allowed recipient of SIGURG and SIGIO signals is the calling process. A similar fcntl(2), F_GETOWN, returns the process number of a socket.

Reception of SIGURG and SIGIO can also be enabled by using ioctl(2) to assign the socket to the user's process group:

```
/* oobdata is the out-of-band data handling routine */
sigset(SIGURG, oobdata);
int pid = -getpid();
if (ioctl(client, SIOCSPGRP, (char *) &pid) < 0) {
                perror("ioctl: SIOCSPGRP");
}
```

Another signal that is useful in server processes is SIGCHLD. This signal is delivered to a process when any child process changes state. Normally, servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodically polling for exit status. For example, the remote login server loop shown previously can be augmented as shown in Example 2-16.

Example 2-16 SIGCHLD Signal

```
int reaper();
...
sigset(SIGCHLD, reaper);
listen(f, 5);
while (1) {
                int g, len = sizeof from;
                g = accept(f, (struct sockaddr *) &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                syslog(LOG_ERR, "rlogind: accept: %m");
                        continue;
                }
}

#include <wait.h>
reaper()
{
                int options;
                int error;
                siginfo_t info;

                options = WNOHANG | WEXITED;
                bzero((char *) &info, sizeof(info));
                error = waitid(P_ALL, 0, &info, options);
}
```

If the parent server process fails to reap its children, zombie processes result.

Selecting Specific Protocols
If the third argument of the socket(3SOCKET) call is 0, socket(3SOCKET) selects a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. When using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, it can be important for the protocol argument to set up de-multiplexing. For example, raw sockets in the Internet family can be used to implement a new protocol on IP, and the socket receives packets only for the protocol specified. To obtain a particular protocol, determine the protocol number as defined in the protocol family. For the Internet family, use one of the library routines discussed in "Standard Routines", such as getprotobyname(3SOCKET):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET6, SOCK_STREAM, pp->p_proto);
```

This results in a socket **s** using a stream-based connection, but with protocol type of newtcp instead of the default tcp.

Address Binding
TCP and UDP use a 4-tuple of *local IP address*, *local port number*, *foreign IP address*, and *foreign port number* to do their addressing. TCP requires these 4-tuples to be unique. UDP does not. It is unrealistic to expect user programs to always know proper values to use for the local address and local port, since a host can reside on multiple networks and the set of allocated port numbers is not directly accessible to a user. To avoid these problems, you can leave parts of the address unspecified and let the system assign the parts appropriately when needed. Various portions of these tuples may be specified by various parts of the sockets API.

bind(3SOCKET)
Local address or local port or both

connect(3SOCKET)
Foreign address and foreign port
A call to accept(3SOCKET) retrieves connection information from a foreign client, so it causes the local address and port to be specified to the system (even though the caller of accept(3SOCKET) didn't specify anything), and the foreign address and port to be returned.

A call to listen(3SOCKET) can cause a local port to be chosen. If no explicit bind(3SOCKET) has been done to assign local information, listen(3SOCKET) causes an ephemeral port number to be assigned.

A service that resides at a particular port, but which does not care what local address is chosen, can bind(3SOCKET) itself to its port and leave the local address unspecified (set to in6addr_any, a variable with a constant value in <netinet/in.h>). If the local port need not be fixed, a call to listen(3SOCKET) causes a port to be chosen. Specifying an address of in6addr_any or a port number of 0 is known as wildcarding. (For AF_INET, INADDR_ANY is used in place of in6addr_any.)

The wildcard address simplifies local address binding in the Internet family. The sample code below binds a specific port number, MYPORT, to a socket, and leaves the local address unspecified.

Example 2-17 Bind Port Number to Socket

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in6 sin;
...
          s = socket(AF_INET6, SOCK_STREAM, 0);
          bzero (&sin6, sizeof (sin6));
          sin.sin6_family = AF_INET6;
          sin.sin6_addr.s6_addr = in6addr_any;
          sin.sin6_port = htons(MYPORT);
          bind(s, (struct sockaddr *) &sin, sizeof sin);
```

Each network interface on a host typically has a unique IP address. Sockets with wildcard local addresses can receive messages directed to the specified port number and sent to any of the possible addresses assigned to a host. For example, if a host has two interfaces with addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as in Example 2-17, the process can accept connection requests addressed to 128.32.0.4 or 10.0.0.78. To allow only hosts on a specific network to connect to it, a server binds the address of the interface on the appropriate network.
Similarly, a local port number can be left unspecified (specified as 0), in which case the system selects a port number. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
bzero (&sin, sizeof (sin));
(void) inet_pton (AF_INET6, ":ffff:127.0.0.1", sin.sin6_addr.s6_addr);
sin.sin6_family = AF_INET6;
sin.sin6_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The system uses two criteria to select the local port number:
The first is that Internet port numbers less than 1024 (IPPORT_RESERVED) are reserved for privileged users (that is, the superuser). Nonprivileged users can use any Internet port number greater than 1024. The largest Internet port number is 65535.

The second criterion is that the port number is not currently bound to some other socket.
The port number and IP address of the client is found through either accept(3SOCKET) (the *from* result) or getpeername(3SOCKET).

In certain cases, the algorithm used by the system to select port numbers is unsuitable for an application. This is because associations are created in a two-step process. For example, the Internet file transfer protocol specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation, the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, you must perform an option call before address binding:

```
 ...
int on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

With this call, local addresses already in use can be bound. This does not violate the uniqueness requirement, because the system still verifies at connect time that any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned.

Using Multicast

IP multicasting is only supported on AF_INET6 and AF_INETsockets of type SOCK_DGRAM and SOCK_RAW, and only on subnetworks for which the interface driver supports multicasting.

Sending IPv4 Multicast Datagrams

To send a multicast datagram, specify an IP multicast address in the range 224.0.0.0 to 239.255.255.255 as the destination address in asendto(3SOCKET) call.

By default, IP multicast datagrams are sent with a time-to-live (TTL) of 1, which prevents them from being forwarded beyond a single subnetwork. The socket option IP_MULTICAST_TTL allows the TTL for subsequent multicast datagrams to be set to any value from 0 to 255, to control the scope of the multicasts:

```
    u_char ttl;
    setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,sizeof(ttl))
```

Multicast datagrams with a TTL of 0 are not transmitted on any subnet, but can be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket (see below). Multicast datagrams with TTL greater than one can be delivered to more than one subnet if one or more multicast routers are attached to the first-hop subnet. To provide meaningful scope control, the multicast routers support the notion of TTL "thresholds", which prevent datagrams with less than a certain TTL from traversing certain subnets. The thresholds enforce the following convention that multicast datagrams with initial TTL:

0	Are restricted to the same host
1	Are restricted to the same subnet
32	Are restricted to the same site
64	Are restricted to the same region
128	Are restricted to the same continent
255	Are unrestricted in scope

"Sites" and "regions" are not strictly defined, and sites can be subdivided into smaller administrative units, as a local matter.

An application can choose an initial TTL other than the ones listed above. For example, an application might perform an "expanding-ring search" for a network resource by sending a multicast query, first with a TTL of 0, and then with larger and larger TTLs, until a reply is received, using (for example) the TTL sequence 0, 1, 2, 4, 8, 16, 32.

The multicast router refuses to forward any multicast datagram with a destination address between 224.0.0.0 and 224.0.0.255, inclusive, regardless of its TTL. This range of addresses is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership reporting.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. (If the host is also a multicast router and the TTL is greater than 1, a multicast can be **forwarded** to interfaces other than originating interface.) A socket option is available to override the default for subsequent transmissions from a given socket:

```
    struct in_addr addr;
    setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr))
```

where addr is the local IP address of the outgoing interface you want. Revert to the default interface by specifying the address INADDR_ANY. The local IP address of an interface is obtained with the SIOCGIFCONF ioctl. To determine if an interface supports multicasting, fetch the interface flags with the SIOCGIFFLAGS ioctl and test if the IFF_MULTICAST flag is set. (This option is intended primarily for multicast routers and other system services specifically concerned with internet topology.)

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
    u_char loop;
    setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop))
```

where loop is 0 to disable loopback, and 1 to enable loopback. This option provides a performance benefit for applications that have only one instance on a single host (such as a router or a mail demon), by eliminating the overhead of receiving their own transmissions. It should not normally be used by applications that can have more than one instance on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time querying program).

If the sending host belongs to the destination group on another interface, a multicast datagram sent with an initial TTL greater than 1 can be delivered to the sending host on the other interface. The loopback control option has no effect on such delivery.

Receiving IPv4 Multicast Datagrams

Before a host can receive IP multicast datagrams, it must become a member of one, or more, IP multicast group. A process can ask the host to join a multicast group by using the following socket option:

```
    struct ip_mreq mreq;
    setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq))
```

where mreq is the structure

```
    struct ip_mreq {
        struct in_addr imr_multiaddr;   /* multicast group to join */
        struct in_addr imr_interface;   /* interface to join on */
    }
```

Each membership is associated with a single interface, and it is possible to join the same group on more than one interface. Specify imr_interface to be in6addr_any to choose the default multicast interface, or one of the host's local addresses to choose a particular (multicast-capable) interface.

To drop a membership, use

```
    struct ip_mreq mreq;
    setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq))
```

where mreq contains the same values used to add the membership. The memberships associated with a socket are also dropped when the socket is closed or the process holding the socket is killed. More than one socket can claim a membership in a particular group, and the host remains a member of that group until the last claim is dropped.

Incoming multicast packets are accepted by the kernel IP layer if any socket has claimed a membership in the destination group of the datagram. Delivery of a multicast datagram to a particular socket is based on the destination port and the memberships associated with the socket (or protocol type, for raw sockets), just as with unicast datagrams. To receive multicast datagrams sent to a particular port, bind to the local port, leaving the local address unspecified (such as, INADDR_ANY).

More than one process can bind to the same SOCK_DGRAM UDP port if the bind(3SOCKET) is preceded by:

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one))
```

In this case, every incoming multicast or broadcast UDP datagram destined to the shared port is delivered to all sockets bound to the port. For backwards compatibility reasons, **this does not apply to incoming unicast datagrams**. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination port. SOCK_RAW sockets do not require the SO_REUSEADDRoption to share a single IP protocol type.

The definitions required for the new, multicast-related socket options are found in <netinet/in.h>. All IP addresses are passed in network byte-order.

Sending IPv6 Multicast Datagrams

To send a multicast datagram, specify an IP multicast address in the range ff00::0/8 as the destination address in a sendto(3SOCKET) call.

By default, IP multicast datagrams are sent with a hop limit of 1, which prevents them from being forwarded beyond a single subnetwork. The socket option IPV6_MULTICAST_HOPS allows the hoplimit for subsequent multicast datagrams to be set to any value from 0 to 255, to control the scope of the multicasts:

```
uint_l;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &hops,sizeof(hops))
```

Multicast datagrams with a hoplimit of 0 are not transmitted on any subnet, but can be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket (see below). Multicast datagrams with hoplimit greater than one can be delivered to more than one subnet if one or more multicast routers are attached to the first-hop subnet. The IPv6 multicast addresses, unlike their IPv4 counterparts, contain explicit scope information encoded in the first part of the address. The defined scopes are (where X is unspecified):

ffX1::0/16
Node-local scope -- restricted to the same node
ffX2::0/16
Link-local scope
ffX5::0/16
Site-local scope
ffX8::0/16
Organization-local scope
ffXe::0/16
Global scope

An application can, separately from the scope of the multicast address, use different hoplimit values. For example, an application might perform an "expanding-ring search" for a network resource by sending a multicast query, first with a hoplimit of 0, and then with larger and larger hoplimits, until a reply is received, using (for example) the hoplimit sequence 0, 1, 2, 4, 8, 16, 32.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. (If the host is also a multicast router and the hoplimit is greater than 1, a multicast can be **forwarded** to interfaces other than originating interface.) A socket option is available to override the default for subsequent transmissions from a given socket:

```
uint_t ifindex;
ifindex = if_nametoindex )"hme3");
```

```
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_IF, &ifindex, sizeof(ifindex))
```

where ifindex is the interface index for the desired outgoing interface. Revert to the default interface by specifying the value 0.

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
uint_t loop;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop, sizeof(loop))
```

where loop is 0 to disable loopback, and 1 to enable loopback. This option provides a performance benefit for applications that have only one instance on a single host (such as a router or a mail demon), by eliminating the overhead of receiving their own transmissions. It should not normally be used by applications that can have more than one instance on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time querying program).

If the sending host belongs to the destination group on another interface, a multicast datagram sent with an initial hoplimit greater than 1 can be delivered to the sending host on the other interface. The loopback control option has no effect on such delivery.

Receiving IPv6 Multicast Datagrams

Before a host can receive IP multicast datagrams, it must become a member of one, or more, IP multicast group. A process can ask the host to join a multicast group by using the following socket option:

```
struct ipv6_mreq mreq;
setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP, &mreq, sizeof(mreq))
```

where mreq is the structure

```
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr;   /* IPv6 multicast addr */
    unsigned int    ipv6mr_interface;   /* interface index */
}
```

Each membership is associated with a single interface, and it is possible to join the same group on more than one interface. Specify ipv6_interface to be 0 to choose the default multicast interface, or an interface index for one of the host's interfaces to choose that (multicast capable) interface.

To leave a group, use

```
struct ipv6_mreq mreq;
setsockopt(sock, IPPROTO_IPV6, IP_LEAVE_GROUP, &mreq, sizeof(mreq))
```

where mreq contains the same values used to add the membership. The memberships associated with a socket are also dropped when the socket is closed or the process holding the socket is killed. More than one socket can claim a membership in a particular group, and the host remains a member of that group until the last claim is dropped.

Incoming multicast packets are accepted by the kernel IP layer if any socket has claimed a membership in the destination group of the datagram. Delivery of a multicast datagram to a particular socket is based on the destination port and the memberships associated with the socket (or protocol type, for raw sockets), just as with unicast datagrams. To receive multicast datagrams sent to a particular port, bind to the local port, leaving the local address unspecified (such as, INADDR_ANY).

More than one process can bind to the same SOCK_DGRAM UDP port if the bind(3SOCKET) is preceded by:

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one))
```

In this case, every incoming multicast UDP datagram destined to the shared port is delivered to all sockets bound to the port. For backwards compatibility reasons, **this does not apply to incoming unicast datagrams**. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination
port. SOCK_RAW sockets do not require the SO_REUSEADDR option to share a single IP protocol type. The definitions required for the new, multicast-related socket options are found in <netinet/in.h>. All IP addresses are passed in network byte-order.

Zero Copy and Checksum Off-load
In SunOS 5.6 and later, the TCP/IP protocol stack has been enhanced to support two new features: zero copy and TCP checksum off-load. Zero copy uses virtual memory MMU remapping and a copy-on-write technique to move data between the application and the kernel space.Checksum off-loading relies on special hardware logic to off-load the TCP checksum calculation.

Note -
Although zero copy and checksum off-loading are functionally independent of one another, they have to work together to obtain the optimal performance. Checksum off-loading requires hardware support from the network interface and, without this hardware support, zero copy is not enabled.

Zero copy requires that the applications supply page-aligned buffers before VM page remapping can be applied. Applications should use large, circular buffers on the transmit side to avoid expensive copy-on-write faults. A typical buffer allocation is sixteen 8K buffers.

SOCKET OPTIONS
You can set and get several options on sockets
through setsockopt(3SOCKET) and getsockopt(3SOCKET); for example by changing the send or receive buffer space. The general forms of the calls are:
setsockopt(s, level, optname, optval, optlen);
andgetsockopt(s, level, optname, optval, optlen);

In some cases, such as setting the buffer sizes, these are only hints to the operating system. The operating system reserves the right to adjust the values appropriately.

Table 2-4 shows the arguments of the calls.
Table 2-4 setsockopt(3SOCKET) and getsockopt(3SOCKET) Arguments

Arguments	**Description**
s	Socket on which the option is to be applied
level	Specifies the protocol level, such as socket level, indicated by the symbolic constant SOL_SOCKET in sys/socket.h
optname	Symbolic constant defined in sys/socket.h that specifies the option
optval	Points to the value of the option
optlen	Points to the length of the value of the option

For getsockopt(3SOCKET), *optlen* is a value-result argument, initially set to the size of the storage area pointed to by *optval* and set on return to the length of storage used.
It is sometimes useful to determine the type (for example, stream or datagram) of an existing socket. Programs invoked by inetd(1M) can do this by using the SO_TYPE socket option and
the getsockopt(3SOCKET) call:

```
#include <sys/types.h>
#include <sys/socket.h>
 int type, size;
 size = sizeof (int);
if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) <0) {
        ...
}
```

After getsockopt(3SOCKET), type is set to the value of the socket type, as defined in sys/socket.h. For a datagram socket, type would be SOCK_DGRAM.

inetd(1M) Daemon

One of the daemons provided with the system is inetd(1M). It is invoked at start-up time, and gets the services for which it listens from the /etc/inet/inetd.conf file. The daemon creates one socket for each service listed in /etc/inet/inetd.conf, binding the appropriate port number to each socket. See the inetd(1M) man page for details.

inetd(1M) polls each socket, waiting for a connection request to the service corresponding to that socket. For SOCK_STREAM type sockets, inetd(1M) does an accept(3SOCKET) on the listening socket, fork(2)s, dup(2)s the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and exec(2)s the appropriate server.

The primary benefit of inetd(1M) is that services that are not in use are not taking up machine resources. A secondary benefit is that inetd(1M)does most of the work to establish a connection. The server started by inetd(1M) has the socket connected to its client on file descriptors 0 and 1, and can immediately read(2), write(2), send(3SOCKET), or recv(3SOCKET). Servers can use buffered I/O as provided by the stdioconventions, as long as they use fflush(3C) when appropriate.

getpeername(3SOCKET) returns the address of the peer (process) connected to a socket; it is useful in servers started by inetd(1M). For example, to log the Internet address (such as fec0::56:a00:20ff:fe7d:3dd2, which is conventional for representing the IPv6 address of a client), an inetd(1M) server could use the following:

```
   struct sockaddr_storage name;
   int namelen = sizeof (name);
   char abuf[INET6_ADDRSTRLEN];
   struct in6_addr addr6;
   struct in_addr addr;
   if (getpeername(fd, (struct sockaddr *)&name, &namelen) == -1) {
      perror("getpeername");
      exit(1);
   } else {
      addr = ((struct sockaddr_in *)&name)->sin_addr;
      addr6 = ((struct sockaddr_in6 *)&name)->sin6_addr;
      if (name.ss_family == AF_INET) {
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
      } else if (name.ss_family == AF_INET6 && IN6_IS_ADDR_V4MAPPED(&addr6)) {
            /* this is a IPv4-mapped IPv6 address */
            IN6_MAPPED_TO_IN(&addr6, &addr);
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
      } else if (name.ss_family == AF_INET6) {
            (void) inet_ntop(AF_INET6, &addr6, abuf, sizeof (abuf));
      }
      syslog("Connection from %s\n", abuf);
   }
```

Broadcasting and Determining Network Configuration

Broadcasting is not supported in IPv6. It is supported only in IPv4. Messages sent by datagram sockets can be broadcast to reach all of the hosts on an attached network. The network must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Broadcasting is usually used for either of two reasons: to find a resource on a local network without having its address, or functions like routing require that information be sent to all accessible neighbors.

To send a broadcast message, create an Internet datagram socket:

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

and bind a port number to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The datagram can be broadcast on only one network by sending to the network's broadcast address. A datagram can also be broadcast on all attached networks by sending to the special address INADDR_BROADCAST, defined in netinet/in.h.

The system provides a mechanism to determine a number of pieces of information (including the IP address and broadcast address) about the network interfaces on the system.
The SIOCGIFCONF ioctl(2) call returns the interface configuration of a host in a single ifconf structure. This structure contains an array of ifreq structures, one for each address family supported by each network interface to which the host is connected. Example 2-18 shows these structures defined in net/if.h.

Example 2-18 net/if.h Header File

```
struct ifreq {
#define IFNAMSIZ 16
char ifr_name[IFNAMSIZ]; /* if name, e.g., "en0" */
union {
                struct sockaddr ifru_addr;
                struct sockaddr ifru_dstaddr;
                char ifru_oname[IFNAMSIZ]; /* other if name */
                struct sockaddr ifru_broadaddr;
                short ifru_flags;
                int ifru_metric;
                char ifru_data[1]; /* interface dependent data */
                char ifru_enaddr[6];
} ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr
#define ifr_dstaddr ifr_ifru.ifru_dstaddr
#define ifr_oname ifr_ifru.ifru_oname
#define ifr_broadaddr ifr_ifru.ifru_broadaddr
#define ifr_flags ifr_ifru.ifru_flags
#define ifr_metric ifr_ifru.ifru_metric
#define ifr_data ifr_ifru.ifru_data
#define ifr_enaddr ifr_ifru.ifru_enaddr
};
```

The call that obtains the interface configuration is:

```
/*
 * Do SIOCGIFNUM ioctl to find the number of interfaces
 *
 * Allocate space for number of interfaces found
 *
 * Do SIOCGIFCONF with allocated buffer
 *
 */
if (ioctl(s, SIOCGIFNUM, (char *)&numifs) == -1) {
        numifs = MAXIFS;
}
bufsize = numifs * sizeof(struct ifreq);
reqbuf = (struct ifreq *)malloc(bufsize);
if (reqbuf == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
}
ifc.ifc_buf = (caddr_t)&reqbuf[0];
ifc.ifc_len = bufsize;
if (ioctl(s, SIOCGIFCONF, (char *)&ifc) == -1) {
        perror("ioctl(SIOCGIFCONF)");
        exit(1);
}
...
}
```

After this call, *buf* contains an array of ifreq structures, one for each network to which the host is connected. These structures are ordered first by interface name, then by supported address families. ifc.ifc_len is set to the number of bytes used by the ifreq structures.
Each structure has a set of interface flags that tell whether the corresponding network is up or down, point-to-point or broadcast, and so on. Example 2-19 shows the SIOCGIFFLAGS ioctl(2) returning these flags for an interface specified by an ifreq structure.

Example 2-19 Obtaining Interface Flags

```
struct ifreq *ifr;
ifr = ifc.ifc_req;
for (n = ifc.ifc_len/sizeof (struct ifreq); --n >= 0; ifr++) {
   /*
    * Be careful not to use an interface devoted to an address
    * family other than those intended.
    */
   if (ifr->ifr_addr.sa_family != AF_INET)
      continue;
   if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
      ...
   }
   /* Skip boring cases */
   if ((ifr->ifr_flags & IFF_UP) == 0 ||
      (ifr->ifr_flags & IFF_LOOPBACK) ||
      (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
      continue;
}
```

Example 2-20 shows the broadcast of an interface can be obtained with
the SIOGGIFBRDADDR ioctl(2).
Example 2-20 Broadcast Address of an Interface

```
if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
                ...
}
memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,
                sizeof ifr->ifr_broadaddr);
```

the SIOGGIFBRDADDR ioctl(2) can also be used to get the destination address of a point-to-point interface. After the interface broadcast address is obtained, transmit the broadcast datagram with sendto(3SOCKET):

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof dst);
```

Use one sendto(3SOCKET) for each interface to which the host is connected that supports the broadcast or point-to-point addressing.

DAEMON PROCESSES

Daemons are processes that are often started when the system is bootstrapped and terminate only when the system is shut down. Because they don’t have a controlling terminal, they run in the background. UNIX systems have numerous daemons that perform day-to-day activities.

This chapter details the process structure of daemons and explores how to write a daemon. Since a daemon does not have a controlling terminal, we need to see how a daemon can report error conditions when something goes wrong.

Internet SuperServer

There are two ways to offer TCP/IP services: by running server applications standalone as a daemon or by using the Internet super server, inetd(8). inetd is a daemon which monitors a range of ports. If a client attempts to connect to a port inetd handles the connection and forwards the connection to the server software which handles that kind of connection. The advantage of this approach is that it adds an extra layer of security and it makes it easier to log incoming connections. The disadvantage is that it is somewhat slower than using a standalone daemon. It is thus a good idea to run a standalone daemon on, for example, a heavily loaded FTP server.

inetd can be configured using the /etc/inetd.conf file. Let's have a look at an example line from inetd.conf:

```
# File Transfer Protocol (FTP) server:
ftp     stream  tcp     nowait  root    /usr/sbin/tcpd  proftpd
```

This line specifies that inetd should accept FTP connections and pass them to tcpd. This may seem a bit odd, because proftpd normally handles FTP connections. You can also specify to use proftpd directly in inetd.conf, but Slackware Linux normally passes the connection to tcpd. This program passes the connection to proftpd in turn, as specified. tcpd is used to monitor services and to provide host based access control.

Services can be disabled by adding the comment character (#) at the beginning of the line. It is a good idea to disable all services and enable services you need one at a time. After changing /etc/inetd.conf inetd needs to be restarted to activate the changes. This can be done by sending the HUP signal to the inetd process:

```
# ps ax | grep 'inetd'
   64 ?        S      0:00 /usr/sbin/inetd
# kill -HUP 64
```

Or you can use the rc.inetd initialization script to restart inetd:

```
 /etc/rc.d/rc.inetd restart
```

TCP wrappers

As you can see in /etc/inetd.conf connections for most protocols are made through tcpd, instead of directly passing the connection to a service program. For example:

```
# File Transfer Protocol (FTP) server:
```

```
ftp     stream  tcp     nowait  root    /usr/sbin/tcpd  proftpd
```

In this example ftp connections are passed through tcpd. tcpd logs the connection through syslog and allows for additional checks. One of the most used features of tcpd is host-based access control. Hosts that should be denied are controlled via /etc/hosts.deny, hosts that should be allowed via /etc/hosts.allow. Both files have one rule on each line of the following form:

service: hosts

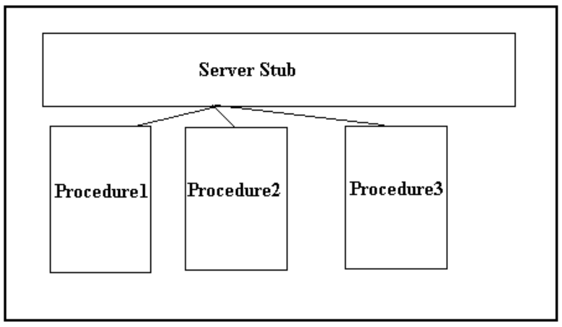
Hosts can be specified by hostname or IP address. The ALL keyword specifies all hosts or all services. Suppose we want to block access to all services managed through tcpd, except for host "trusted.example.org". To do this the following hosts.deny and hosts.allow files should be created.
/etc/hosts.deny:

```
ALL: ALL
```
/etc/hosts.allow:
```
ALL: trusted.example.org
```

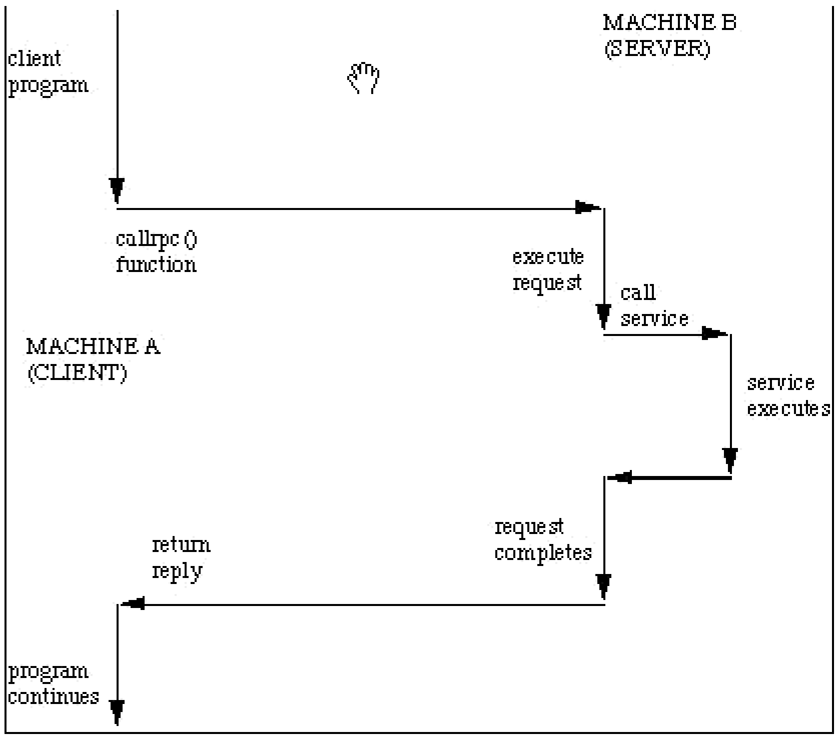
In the hosts.deny access is blocked to all (ALL) services for all (ALL) hosts. But hosts.allow specifies that all (ALL) services should be available to "trusted.example.org".

REMOTE PROCEDURE CALL (RPC)

RPC comes under the Application-Oriented Design, where the client-server communication is in the form of Procedure Calls. We call the machine making the procedure call as *client* and the machine executing the called procedure as *server*. For every procedure being called there must exist a piece of code which knows which machine to contact for that procedure. Such a piece of code is called a *Stub*. On the client side, for every procedure being called we need a unique stub. However, the stub on the server side can be more general; only one stub can be used to handle more than one procedures (see figure). Also, two calls to the same procedure can be made using the same stub.



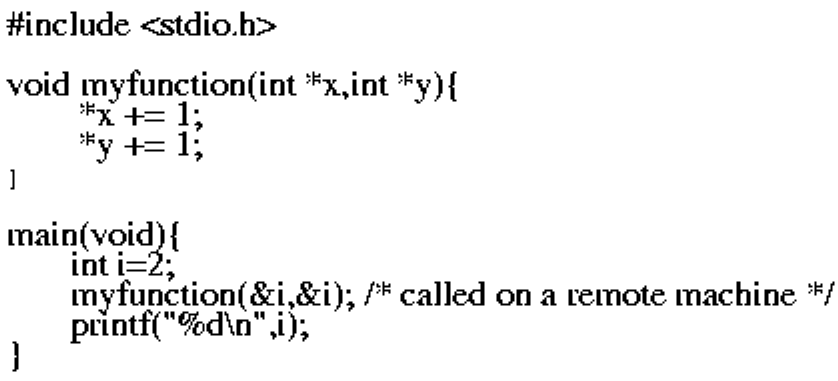
Now let us see how a typical remote procedure call gets executed:-



1. Client program calls the stub procedure linked within its own address space. It is a normal local call.
2. The client stub then collects the parameters and packs them into a message (*Parameter Marshalling*). The message is then given to the transport layer for transmission.
3. The transport entity just attaches a header to the message and puts it out on the network without further ado.
4. When the message arrives at the server the transport entity there passes it tot the server stub, which unmarshalls the parameters.
5. The server stub then calls the server procedure, passing the parameters in the standard way.
6. After it has completed its work, the server procedure returns, the same way as any other procedure returns when it is finished. A result may also be returned.
7. The server stub then marshalls the result into a message and hands it off at the transport interface.
8. The reply gets back to the client machine.
9. The transport entity hands the result to the client stub.
10. Finally, the client stub returns to its caller, the client procedure, along-with the value returned by the server in step 6.

This whole mechanism is used to give the client procedure the illusion that it is making a direct call to a distant server procedure. To the extent the illusion exceeds, the mechanism is said to be **transparent**. But the transparency fails in *parameter passing*. Passing any data (or data structure) by value is OK, but passing parameter 'by reference' causes problems. This is because the pointer in question here, points to an address in the address space of the client process, and this address space is not shared by the server process. So the server will try to search the address pointed to by this passed pointer, in its own address space. This address may not have the value same as that on the client side, or it may not lie in the server process' address space, or such an address may not even exist in the server address space.

One solution to this can be **Copy-in Copy-out**. What we pass is the value of the pointer, instead of the pointer itself. A local pointer, pointing to this value is created on the server side (*Copy-in*). When the server procedure returns, the modified 'value' is returned, and is copied back to the address from where it was taken (*Copy-out*). But this is disadvantageous when the pointer involved point to huge data structures. Also this approach is not foolproof. Consider the following example (C-code) :



```
#include <stdio.h>

void myfunction(int *x,int *y){
      *x += 1;
      *y += 1;
}

main(void){
      int i=2;
      myfunction(&i,&i); /* called on a remote machine */
      printf("%d\n",i);
}
```

The procedure 'myfunction()' resides on the server machine. If the program executes on a single machine then we must expect the output to be '4'. But when run in the client-server model we get '3'. Why ? Because 'x, and 'y' point to different memory locations with the same value. Each then increments its own copy and the incremented value is returned. Thus '3' is passed back and not '4'.

Many RPC systems finesse the whole problem by prohibiting the use of reference parameters, pointers, function or procedure parameters on remote calls (**Copy-in**). This makes the implementation easier, but breaks down the transparency.

Protocol : Another key implementation issue is the protocol to be used - TCP or UDP. If TCP is used then there may be problem in case of network breakdown. No problem occurs if the breakdown happens before client sends its request (client will be notified of this), or after the request is sent and the reply is not received (time-out will occur). In case the breakdown occurs just after the server has sent the reply, then it won't be able to figure out whether its response has reached the client or not. This could be devastating for bank servers, which need to make sure that their reply has in fact reached to the client (probably an ATM machine). So UDP is generally preferred over TCP, in making remote procedure calls.

Idempotent Operations:
If the server crashes, in the middle of the computation of a procedure on behalf of a client, then what must the client do? Suppose it again sends its request, when the server comes up. So some part of the procedure will be re-computed. It may have instructions whose repeated execution may give different results each time. If the side effect of multiple execution of the procedure is exactly the same as that of one execution, then we call such procedures as **Idempotent Procedures**. In general, such operations are called **Idempotent Operations**.

For e.g. consider ATM banking. If I send a request to withdraw Rs. 200 from my account and some how the request is executed twice, then in the two transactions of 'withdrawing Rs. 200' will be shown, whereas, I will get only Rs. 200. Thus 'withdrawing is a non-idempotent operation. Now consider the case when I send a request to 'check my balance'. No matter how many times is this request executed, there will arise no inconsistency. This is an idempotent operation.
Semantics of RPC :
If all operations could be cast into an idempotent form, then time-out and retransmission will work. But unfortunately, some operations are inherently non-idempotent (e.g., transferring money from one bank account to another). So the exact semantics of RPC systems were categorized as follows:

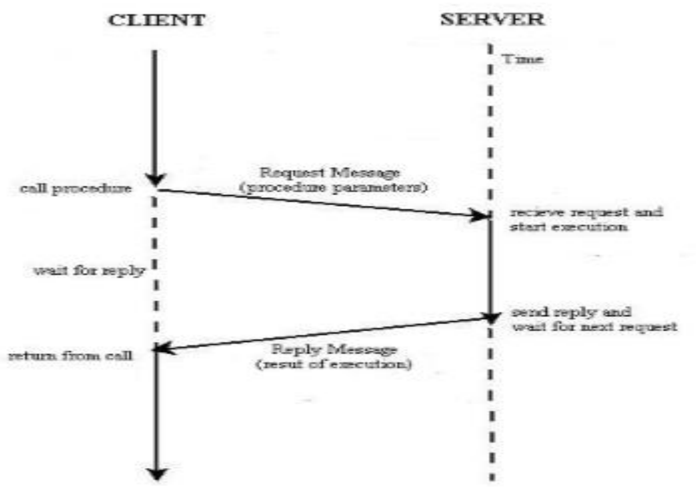
- *Exactly once :* Here every call is carried out 'exactly once', no more no less. But this goal is unachievable as after a server crash it is impossible to tell that a particular operation was carried out or not.
- *At most once :* when this form is used control always returns to the caller. If everything had gone right, then the operation will have been performed exactly once. But, if a server crash is detected, retransmission is not attempted, and further recovery is left up to the client.

- *At least once :* Here the client stub keeps trying over and over, until it gets a proper reply. When the caller gets control back it knows that the operation has been performed one or more times. This is ideal for idempotent operations, but fails for non-idempotent ones.
- *Last of many :* This a version of 'At least once', where the client stub uses a different transaction identifier in each retransmission. Now the result returned is guaranteed to be the result of the final operation, not the earlier ones. So it will be possible for the client stub to tell which reply belongs to which request and thus filter out all but the last one.

TRANSPARENCY ISSUES
A transparent RPC is one in which the local and remote procedure calls are indistinguishable to programmers. The main issue in designing an RPC facility is its transparent property.
There are two types of transparencies required:

i.Syntactic transparency: A remote procedure and a local procedure call should have the same syntax.

ii.Semantic transparency: The semantics of a remote procedure call and local procedure call are identical.

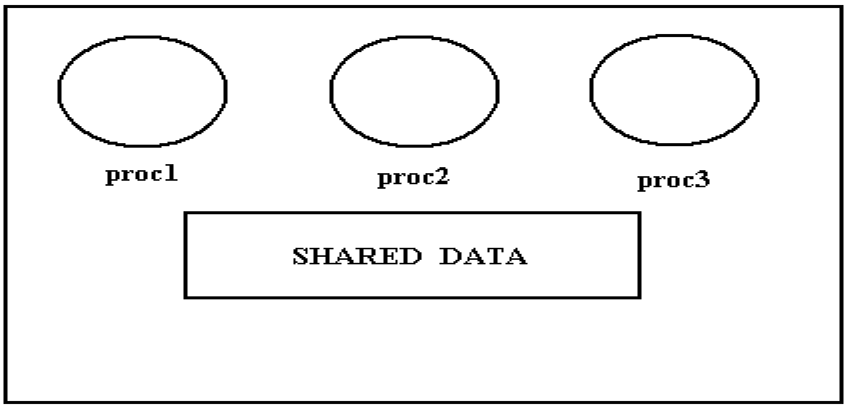


Achieving Syntactic transparency is not an issue but semantic transparency is difficult due to some differences between remote procedure calls and local procedure calls.

- The basic idea of RPC is to make a remote procedure call look transparent. The calling procedure should not be aware that the called procedure is executing on a different machine.
- RPC achieves transparency in an analogous way. When read is a remote procedure a different version of read called client stub is put in the library. It is called using the calling sequence.
- The parameters are packed into a message and sent to the server. When the message arrives at the server, the server’s operating system passes it to server stub which calls receive and blocks incoming messages.
- The server unpacks parameters and calls server procedure and performs its work and returns the result to caller in the usual way of packing the result and sending it to client stub.
- When the client receives the message from server, the operating system of client sees that it is addressed to client process. When the caller gets control following the cal to read, all it knows is that its data are available.
- It has no idea that the work was done remotely instead of the local operating system. In this way transparency is

SUN RPC Model
The basic idea behind Sun RPC was to implement NFS (Network File System). Sun RPC extends the remote procedure call model by defining a remote execution enviroment. It defines a ***remote program*** at the server side as the basic unit of software that executes on a remote machine. Each remote program consists of one or more remote procedures and global data. The global data is static data and

all the procedures inside a remote program share access to its global data. The figure below illustrates the conceptual organization of three remote procedures in a single remote program.



PROGRAM

Sun RPC allows both TCP and UDP for communication between remote procedures and programs calling them. It uses the at least once semantic i.e., the remote procedure is executed at least once. It uses copy-in method of parameter passing but does not support copy-out style. It uses XDR for data representation. It does not handle orphans(which are servers whose corresponding clients have died). Thus if a client gives a request to a server for execution of a remote procedure and eventually dies before accepting the results, the server does not know whom to reply. It also uses a tool called ***rpcgen*** to generate stubs automatically.

Let us suppose that a client (say client1) wants to execute procedure P1(in the figure above). Another client (say client2) wants to execute procedure P2(in the figure above). Since both P1 and P2 access common global variables they must be executed in a mutually exclusive manner. Thus in view of this Sun RPC provides mutual exclusion by default i.e. no two procedures in a program can be active at the same time. This introduces some amount of delay in the execution of procedures, but mutual exclusion is a more fundamental and important thing to provide, without it the results may go wrong.

Thus we see that anything which can be a threat to application programmers, is provided by SUN RPC.

How A Client Invokes A Procedure On Another Host

The remote procedure is a part of a program executing in a remote host. Thus we would have to properly locate the host, the program in it, and the procedure in the program. Each host can be specified by a unique 32-bit integer. SUN RPC standard specifies that each remote program executing on a computer must be assigned a unique 32-bit integer that the caller uses to identify it. Furthermore, Sun RPC assigns a 32-bit integer identifier for each remote procedure inside a given remote program. The procedures are numbered sequentially: 1, 2, ..., N. To help ensure that program numbers defined by separate organizations do not conflict, Sun RPC has divided the set of program numbers into eight groups.

Thus it seems sufficient that if we are able to locate the host, the program in the host, and the procedure in the program, we would be able to uniquely locate the remote procedure which is to be executed.

Accommodating Multiple Versions Of A Remote Program

Suppose somebody wants to change the version of a remote procedure in a remote program. Then as per the identification method described above, he or she would have to make sure that the newer version is compatible with the older one. This is a bottleneck on the server side. Sun RPC provides a solution to this problem. In addition to a program number, Sun RPC includes a 32-bit integer ***version number*** for each remote program. Usually, the first version of a program is assigned version 1. Later versions each receive a unique version number.

Version numbers provide the ability to change the details of a remote procedure call without obtaining a new program number. Now, the newer client and the older client are disjoint, and no compatibility is required between the two. When no request comes for the older version for a pretty long time, it is deleted. Thus, in practice, each RPC message identifies the intended recipient on a given computer by a triple:

(Program number, version number, procedure number)

Thus it is possible to migrate from one version of a remote procedure to another gracefully and to test a new version of the server while an old version of the server continues to operate.

Mapping A Remote Program To A Protocol Port

At the bottom of every communication in the RPC model there are transport protocols like UDP and TCP. Thus every communication takes place with the help of sockets. Now, how does the client know to which port to connect to the server? This is a real problem when we see that we cannot have a standard that a particular program on a particular host should communicate through a particular port. Because the program number is 32 bit and we can have 2^{32} programs whereas both TCP and UDP uses 16 bit port numbers to identify communication endpoints. Thus RPC programs can potentially outnumber protocol ports. Thus it is impossible to map RPC program numbers onto protocol ports directly. More important, because RPC programs cannot all be assigned a unique protocol port, programmers cannot use a scheme that depends on well-known protocol port assignments. Thus, at any given time, a single computer executes only a small number of remote programs. As long as the port assignments are temporary, each RPC program can obtain a protocol port number and use it for communication.

If an RPC program does not use a reserved, well-known protocol port, clients cannot contact it directly. Because, when the server (remote program) begins execution, it asks the operating system to allocate an unused protocol port number. The server uses the newly allocated protocol port for all communication. The system may choose a different protocol port number each time the server begins(i.e., the server may have a different port assigned each time the system boots).

The client (the program that issues the remote procedure call) knows the machine address and RPC program number for the remote program it wishes to contact. However, because the RPC program (server) only obtains a protocol port after it begins execution, the client cannot know which protocol port the server obtained. Thus, the client cannot contact the remote program directly.

Dynamic Port Mapping

To solve the port identification problem, a client must be able to map from an RPC program and a machine address to the protocol port that the server obtained on the destination machine when it started. The mapping must be dynamic because it can change if the machine reboots or if the RPC program starts execution again.

To allow clients to contact remote programs, the Sun RPC mechanism includes a dynamic mapping service. The RPC port mapping mechanism uses a server to maintain a small database of port mappings on each machine. This RPC server waits on a particular port number (111) and it receives the requests for all remote procedure calls. Whenever a remote program (i.e., a server) begins execution, it allocates a local port that it will use for communication. The remote program then contacts the server on its local machine for registration and adds a pair of integers to the database:

(RPC program number, protocol port number)

Once an RPC program has registered itself, callers on other machines can find its protocol port by sending a request to the server. To contact a remote program, a caller must know the address of the machine on which the remote program executes as well as the RPC program number assigned to the program. The caller first contacts the server on the target machine, and sends an RPC program number. The server returns the protocol port number that the specified program is currently using. This server is called the ***RPC port mapper*** or simply the ***port mapper***. A caller can always reach the port mapper because it communicates using the well known protocol port, 111. Once a caller knows the protocol port number the target program is using, it can contact the remote program program directly.

RPC Programming

RPC Programming can be thought in multiple levels. At one extreme, the user writing the application program uses the RPC library. He/she need not have to worry about the communication through the network. At the other end there are the low level details about network communication. To execute a remote procedure the client would have to go through a lot of overhead e.g., calling XDR for formatting of data, putting it in output buffer, connecting to port mapper and subsequently connecting to the port through which the remote procedure would communicate etc. The ***RPC library*** contains procedures that provide almost everything required to make a remote procedure call. The library contains procedures for marshaling and unmarshaling of the arguments and the results respectively. Different XDR routines are available to change the format of data to XDR from native, and from XDR to native format. But still a lot of overhead remains to properly call the library routines. To minimize the overhead faced by the application programmer to call a remote procedure a tool named ***rpcgen*** is devised which generates client and server stubs. The stubs are generated automatically, thus they have loose flexibility e.g., the timeout time, the number of retransmissions are fixed. The program specification file is given as input and both the server and client stubs are automatically generated by rpcgen. The specification file should have a .x extension attatched to it. It contains the following information:-

- Constant declarations, global data (if any), information about all remote procedures ie. Procedure argument type, return type .