Berkeley Sockets

Study and use of Berkeley sockets in network programming

Pratap Vikram Singh

December 7th, 1997

Contents

1. Berkeley Sockets

1.1 Introduction

Berkeley sockets are one of the most prevalent communication APIs in UNIX. It is developed for the C language. Some points to be considered for network programming are:

1. The network program should know what role it is playing, client or server.
2. The connection can be of two types, connection-oriented or connection-less.
3. A network application must know the *names* of its peer processes to verify that those processes have to authority to request services.
4. There are a lot of parameters required for a network connection like protocol, local address, local process, foreign address and foreign process.
5. Support for multiple protocols must be provided e.g. TCP/IP, XNS, etc.

1.2 UNIX Domain Protocols

Sockets in the Unix domain can only be used to communicate with processes in the same Unix system. There are both connection-less and connection-oriented protocols. Both are considered reliable as both exist within the kernel and are not transmitted across communication mediums. The only difference is that the connection-oriented protocol provides flow control while the connection-less does not. The Unix domain protocols provide a feature that is not currently provided by any other protocol family. This is the ability to pass access rights from one process to another e.g. passing of file descriptors. The 5 tuple for this protocol is slightly different.

1. The protocol is same as before PF_UNIX.
2. The local and foreign address are zero as they are in the same system.
3. The local and foreign processes are specified as pathnames.

The system creates some files as specified in the pathname. These are not regular files and cannot be opened by normal system calls. The call `stat` reports them as S_IFSOCK type.

1.3 Socket Addresses

The socket system call requires a pointer to a socket address structure. The structure is defined in `<sys/socket.h>` as:

```
struct sockaddr {
               u_short        sa_family;     // Address family: AF_XXX value
               char           sa_data[14];   // Upto 14 bytes of protocol specific address
};
```

The contents of the 14 bytes of protocol specific address are interpreted according to the type of address.

For the *internet* family it is defined in `<netinet/in.h>` as:

```
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/host id defined above
               char           sin_zero[8];   // Unused
};
```

For the *Xerox NS* family it is defined in `<sys/ns.h>` as :

```
union ns_host {
                u_char          c_host[6];      // Host id address as 6 bytes
                u_short         s_host[3];      // Hostid address as three 16-bit shorts
};

union ns_net {
                u_char          c_net[4];       // Netid as 4 bytes
                u_short         s_net[2];       // Netid as two 16-bit shorts
};

struct ns_addr {                                // Combined 12 byte XNS address
                union ns_net    x_net;          // 4 byte netid
                union ns_host   x_host;         // 6 byte hostid
                u_short         x_port;         // 2 byte port
};

struct sockaddr_ns {
                u_short         sns_family;     // AF_NS
                struct ns_addr  sns_addr;       // 12 byte XNS address
                char            sns_zero[2];    // Unused
};
```

For the Unix domain it is defined in `<sys/un.h>` as :

```
struct sockaddr_un {
                short           sun_family;     // AF_UNIX
                char            sun_path[108];  // pathname
};
```

The system calls have to take a `struct sockaddr` type structure. So before passing the protocol specific structure it must be typecasted. e.g.

```
struct sockaddr_in    serv_addr;
connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
```

2. Socket System Calls

2.1 Socket()

To do network I/O, the first thing a process must do is call the `socket()` system call specifying the type of communication protocol desired.

Syntax:

```
int socket(
        int family,             // Family of protocols
        int type,               // Type of connection
        int protocol);          // Protocol to use
```

The *family* is defined as :

TYPE	DESCRIPTION
PF_UNIX	Unix internal protocols
PF_INET	ARPA internet protocols

The socket *type* is defined as:

TYPE	DESCRIPTION
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_RAW	Raw socket
SOCK_RDM	Reliably delivered message socket
SOCK_SEQPACKET	Sequenced packet socket

The *protocol* argument specifies the particular protocol to be used with the socket. Generally only a single protocol exists to support a protocol family. This parameter is passed as 0.

This call returns an integer value similar to a file descriptor. This is the *socket descriptor*. A value of -1 indicates an error. The *errno* values are:

EACCES	Permission to create a socket of the specified type and/or protocol is denied.
EMFILE	The peer-process descriptor table is full.
ENOMEM	Insufficient user memory is available.
ENOSR	There were insufficient STREAMS resources available to complete the operation.
EPROTONOSUPPORT	The protocol type or the specified protocol is not supported within this domain.

In the 5 tuple association, this system call defines the *protocol* value. Before using the connection, we must specify the rest of the four values.

2.2 Socketpair()

This call has been implemented for the Unix domain only. It returns a pair of socket descriptors and is similar to the *pipe* system call. The sockets returned by this system call are bi-directional and are called Unix *stream pipes*.

Syntax:
```
int socketpair(
        int family,             // Family of protocol
        int type,               // Type of connection
        int protocol,           // Type of protocol
        int sockvec[2]);        // Array to store the socket descriptors
```

The *family* is PF_UNIX.

The valid *types* are:
1. SOCK_STREAM
2. SOCK_DGRAM

Since this call is limited for the Unix domain, we do not have any defined protocols so it is always passed as zero. The third parameter is as array of two integers which store the two unnamed and connected socket descriptors on a successful return.

The function returns 0 on success and -1 on error. The *errno* values are:

EAFNOSUPPORT	The specified address family is not supported on this machine.
EMFILE	Too many descriptors are in use by this process.
ENOMEM	There was insufficient user memory for the operation to complete.
ENOSR	There were insufficient STREAMS resources for the operation to complete.
EOPNOSUPPORT	The specified protocol does not support creation of socket pairs.
EPROTONOSUPPORT	The specified protocol is not supported on this machine.

2.3 Bind()

This call assigns a name to an unnamed socket. It is used by a process to register itself to the system so any messages sent to its socket must be directed to it. This call fills the *local address* and *local process* fields of the 5 tuple.

Syntax:
```
int bind(
        int                     sockfd,         // Socket descriptor
        struct sockaddr                 *myaddr,        // Pointer to a protocol specific structure
        int                     addrlen);       // Size of structure
```

This function returns 0 on success and -1 on error. The *errno* values are :

EACCES	Search permission is denied for a component of the path prefix of the pathname in name.
EIO	An I/O error occurred while making the directory entry or allocating the inode.
EISDIR	A null pathname was specified.
ELOOP	Too many symbolic links were encountered in translating the pathname in name.
ENOENT	A component of the path prefix of the pathname in name does not exist.
ENOTDIR	A component of the path prefix of the pathname in name is not a directory.
EROFS	The inode would reside on a read-only file system.

2.4 Connect()

A client process connects a socket descriptor following a *socket* call to establish a connection with the server.

Syntax:
```
int connect(
        int                     sockfd,         // Socket descriptor
        struct sockaddr                 *servaddr,      // Pointer to structure with server address
        int                     addrlen);       // Sizeof address structure
```

Stream sockets will use this call once while datagram sockets will use it every time they send a packet.

This function call returns 0 on success and -1 on error. The *errno* values are:

EACCES	Search permission is denied for a component of the path prefix of the path name in name.
EADDRINUSE	The address is already in use.
EADDRNOTAVAIL	The specified address is not available on the remote machine.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EALREADY	The socket is non-blocking and a previous connection attempt has not yet been completed.
EBADF	sockfd is not a valid descriptor.
ECONNREFUSED	The attempt to connect was forcefully rejected. The calling program should close the socket descriptor, and issue another socket call to obtain a new descriptor before attempting another connect call.
EINPROGRESS	The socket is non-blocking and the connection cannot be completed immediately. It is possible to select for completion by selecting the socket for writing. However, this is only possible if the socket STREAMS module is the topmost module on the protocol stack with a write service procedure. This will be the normal case.
EINTR	The connection attempt was interrupted before any data arrived by the delivery of a signal.
EINVAL	addrlen is not the size of a valid address for the specified address family.
EIO	An I/O error occurred while reading from or writing to the file system.
EISCONN	The socket is already connected.
ELOOP	Too many symbolic links were encountered in translating the pathname in name.
ENETUNREACH	The network is not reachable from this host.
ENOENT	A component of the path prefix of the pathname in name does not exist.
ENOENT	The socket referred to by the pathname in name does not exist.
ENOSR	There were insufficient STREAMS resources available to complete the operation.
ETIMEDOUT	Connection establishment timed out

2.5 Listen()

This call is used by a connection oriented server to indicate that it is willing to receive connections. This call is executed immediately after the socket and the *bind* system calls and immediately before the *accept* system call. It is possible for some client requests to arrive between these two. So the system should store the pending messages for the server till it makes the *accept* system call.

Syntax:
```
int listen(
       int sockfd,    // Socket descriptor
       int backlog);  // No of messages that the system should track
```

The *backlog* parameter cannot be more than 5.

2.6 Accept()

An actual connection from some client is awaited for by having the server execute this call.

Syntax:
```
int accept(
       int             sockfd,        // Socket descriptor
       struct sockaddr       *peer,          // Address of peer to connect
       int             *addrlen);     // Size of address structure
```

This call takes the first connection request from the queue and creates another socket with the same properties as *sockfd*. If there are no pending requests, this call blocks.

This function returns:
1. on error.
2. New socket descriptor on success.
3. The *peer* address.
4. The size of *peer* address.

The new socket descriptor returned by this call completes the 5 tuple.

2.7 Send(), Sendto(), Recv() and Recvfrom()

Used to send and receive messages. The `sendto` and `recvfrom` calls are for connection-less processes while the other two are for connection-oriented.

Syntax:
```
int send(
        int                     sockfd, // Socket descriptor
        const char              *buff,  // Message buffer
        int                     nbytes, // No of bytes to send
        int                     flags); // Flags

int sendto(
        int                     sockfd, // Socket descriptor
        const char              *buff,  // Message buffer
        int                     nbytes, // No of bytes to send
        int                     flags,          // Flags
        struct sockaddr         *to,            // Address of destination
        int                     addrlen);       // Length of address structure

int revc(
        int                     sockfd,         // Socket descriptor
        const char              *buff,          // Message buffer
        int                     nbytes,         // No of bytes to receive
        int                     flags);         // Flags

int recvfrom(
        int                     sockfd,         // Socket descriptor
        const char              *buff,          // Message buffer
        int                     nbytes,         // No of bytes to receive
        int                     flags,          // Flags
        struct sockaddr                 *from,          // Address of source filled
        int                     *addrlen);      // Length of address structure filled
```

The *flags* argument is either zero or is or'red using the following constants:

MSG_OOB	Send or receive out-of-band data.
MSG_PEEK	Peek at incoming message. Used by `recv` and `recvfrom`.

This function returns the length of data that was sent or received.

2.8 Close()

The normal Unix *close* system call is used for closing a socket.

Syntax:
```
int close(int sockfd);
```

If the socket being closed is associated with a protocol that promises reliable delivery, the system must ensure that any data within the kernel that still has to be transmitted or acknowledged, is sent.

3. Byte Ordering Routines

These routines handle the potential byte order difference between different computer architectures and different network protocols.

Syntax:

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

ulong htonl(u_long hostlong);
u_short htons(u_short hostshort);
u_long ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

The following table describes the functions :

FUNCTION	DESCRIPTION
`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

Implicit to these functions are that a short integer occupies 16 bits and a long integer occupies 32 bits.

4. Address Conversion Routines

Address conversion routines are used for internet address conversions.

Syntax:

```
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long  inet_addr(const char *cp);
char           *inet_ntoa(const struct in_addr in);
```

The first function takes the stringified IP address and returns the 32 bit value.

The second function takes the *in_addr* structure and returns a pointer to the stringified IP address form.

5. Advanced Socket System Calls

5.1 Readv() and Writev()

These calls are called *scatter read* and *gather write*. They can be used to write or read multiple buffers in a single operation. They are atomic calls.

Syntax:

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

int writev( int fd, struct iovec iov[], int iovcount );
int readv ( int fd, struct iovec iov[], int iovcount );
```

The structure `iovec` is defined as:

```
struct iovec {
        caddr_t iov_base;       // Starting address of buffer
        int     iovlen;         // Size of buffer in bytes
};
```

Each element of the *iovec* array specifies a base address of a buffer and the amount of data to be written or read from it. The parameter *iovcount* specifies the number of separate buffers. In case of a datagram system, each *write* will generate separate datagrams but if combined in one, a *writev* will generate only one.

5.2 Recvmsg() and Sendmsg()

These are the most general of the *read* and *write* system calls.

Syntax:

```
int sendmsg( int sockfd, struct msghdr msg[], int flags );
int recvmsg( int sockfd, struct msghdr msg[], int flags );
```

The structure `msghdr` is defined as:

```
struct msghdr{
        caddr_t      msg_name;                // Optional address
        int          msg_namelen;             // Size of address
        struct iovec *msg_iov;                // Scatter or gather array
        int          msg_iovlen;              // Elements in msg_iov
        caddr_t      msg_accrights;           // Access rights sent or received
        int          msg_accrightslen;        // Length of the access rights buffer
};
```

The fields `msg_name` and `msg_namelen` are not used for connected processes.

5.3 Getpeername()

This system call returns the name of the peer process that is connected to a given socket.

Syntax:

```
int getpeername( int sockfd, struct sockaddr *peer, int *addrlen );
```

This function returns the *foreign address* and the *foreign process* elements of the 5 tuple association of `sockfd`.

5.4 Getsockname()

This function call returns the name associated with the socket.

Syntax:

```
int getsockname( int sockfd, struct sockaddr *peer, int *addrlen );
```

This call returns the *local address* and the *local process* of the association.

5.5 Shutdown()

This call provides more control over a connection termination.

Syntax:

```
int shutdown( int sockfd, int howto );
```

The control is defined using different values of `howto`. They are:

VALUE	FUNCTION
0	No more data can be received on the socket
1	No more data can be sent through the socket
2	Both sends and receives are disallowed

6. Reserved Ports

There are two ways for a process to have a port assigned to it

1. The process can itself specify the port number.
2. The process can leave it to the system to assign it any available port.

The system call `rresvport` is used by a process for assigning a port.

Syntax:

```
int rresvport( int *aport );
```

The port scenario is shown below:

	Internet	**XNS**
Reserved Ports	1-1023	1-2999
Ports automatically assigned by the system	1024-5000	3000-65535
Ports assigned by rresvport	512-1023	NA

This function checks the value of *aport* passed to it and assigns any port available equal to or below the value till 512. If successful it creates a socket, binds it to the port and returns the value of the socket descriptor. Otherwise it returns -1 and sets the value of *errno* to EAGAIN.