Network Programming in UNIX
A glance at the Sockets API

Björn Lindberg
Department of Computer Engineering
Mälardalen University
Västerås, Sweden

1 About this Document

This document briefly covers the basics of utilising the sockets API in the context of UNIX network programming. It's purpose is to serve as a reference or a starting point for learning elementary TCP and UDP sockets, and parts of it can be used as student handouts or as a basis for a tutorial. The reader is assumed to be somewhat familiar with UNIX programming, as well as with the TCP, UDP, and IPv4 protocols.

Turn to the on-line manual for information on the various error codes that might be returned by the socket functions, as they are *not* covered in this text.

2 Introduction to Sockets

Sockets can be thought of as communication endpoints. Two processes, a web browser and a web server, for example, exchange data through their respective sockets.

We begin describing the sockets API by examining the IPv4 and generic socket address structures.

2.1 The IPv4 Socket Address Structure

The structure for an IPv4 socket address, also known as an "Internet socket address structure", is named `sockaddr_in` and is defined in the `<netinet/in.h>` header. Figure 1 shows the Posix.1g definition.

Some points to make are:

- Only three members are actually required by Posix.1g: `sin_family`, `sin_addr`, and `sin_port`. Posix-compliant implementations may define additional structure members; in fact, this is normal for an Internet socket address structure. The `sin_zero` member is added by most implementations so that socket address structures are at least 16 bytes in size.

- Even if the `sin_length` field is present, we neither set nor examine it (unless we are handling routing sockets [Ste98]).

```
struct in_addr {
  in_addr_t   s_addr;      /* IPv4 address (32 bits) */
};                         /* network byte ordered   */

struct sockaddr_in {
  uint8_t         sin_len;     /* Structure length (16)           */
  sa_family_t     sin_family;  /* AF_INET                         */
  in_port_t       sin_port;    /* TCP or UDP port number (16 bits) */
                               /* network byte ordered            */
  struct in_addr  sin_addr;    /* IPv4 address (32 bits)          */
                               /* network byte ordered            */
  char            sin_zero[8]; /* Unused                          */
};
```

Figure 1: The IPv4 socket address structure.

```
struct sockaddr {
  uint8_t      sa_len;
  sa_family_t  sa_family;      /* Address family. AF_INET for IPv4, */
                               /* AF_INET6 for IPv6, etc.           */
  char         sa_data[14];    /* Protocol-specific address         */
};
```

Figure 2: The generic socket address structure.

- The IPv4 address and the TCP or UDP port number are stored in the structure in network byte order. (Big-endian for Internet protocols.)
- The `sin_zero` array is unused, but should always be set to 0. When binding a specific IPv4 address to a socket, this member *must* be 0, although most uses of the socket address structure do not require this.

2.2 The Generic Socket Address Structure

When passing socket structures as arguments to any of the socket functions, they are passed by reference. This is because socket functions that take one of these pointers as an argument must be able to support different kinds of socket address structures. (For example, instead of using IPv4 socket addresses, we could be using IPv6 socket addresses, etc.) The chosen solution (in 1982, before ANSI C) was to define a generic socket address structure in the `<sys/socket.h>` header. The structure is shown in figure 2.

Posix.1g datatypes used in both the socket address structures and socket functions are shown in figure 3.

2.3 Byte order conversion functions

The Internet protocols use big-endian byte ordering for the 16-bit port number and the 32-bit IPv4 address in a TCP segment. Posix.1g specifies that certain members of the socket address structure be maintained in network byte order.

Datatype	Description	Header
`uint8_t`	unsigned 8-bit integer	`<sys/types.h>`
`uint16_t`	unsigned 16-bit integer	`<sys/types.h>`
`uint32_t`	unsigned 32-bit integer	`<sys/types.h>`
`sa_famliy_t`	address family of socket address structure	`<sys/socket.h>`
`socklen_t`	length of socket address structure, normally `unit32_t`	`<sys/socket.h>`
`in_addr_t`	IPv4 address, normally `uint32_t`	`<netinet/in.h>`
`in_port_t`	TCP or UDP port, normally `uint16_t`	`<netinet/in.h>`

Figure 3: Some Posix.1g datatypes.

Therefore we use the following functions to convert between host and byte order respectively.

```
#include <netinet/in.h>
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);

Returns: value in network byte order

uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);

Returns: value in host byte order
```

2.4 `inet_aton`, `inet_addr`, and `inet_ntoa` functions

An IP address is often displayed as a dotted-decimal string, for example "`204.138.245.1`". The value that goes into the `sin_addr.s_addr` member of the IPv4 socket address structure, however, is a binary value in network byte order.

In this and the next section, we describe two groups of address conversion functions that convert a string representation of an IPv4 address to its 32-bit, network byte ordered, binary counterpart.

```
#include <arpa/inet.h>
int inet_aton(const char *strptr, struct in_addr *addrptr);

Returns: 1 if string was valid, 0 on error

in_addr_t inet_addr(const char *strptr);

Returns: 32-bit binary network byte ordered IPv4 address; INADDR_NODE on error

char *inet_ntoa(struct in_addr inaddr);

Returns: pointer to dotted-decimal string
```

The `inet_aton` function converts the C character string pointed to by `strptr` to a corresponding 32-bit binary value in network byte order. This value is stored through the `addrptr` argument.

The same conversion is also done by the `inet_addr` function, but this function returns the binary value instead of storing it through a pointer. A serious problem with this function is that all 2^{32} possible binary values are valid IP addresses, but the function returns the constant `INADDR_NONE` (typically 32 one-bits) on error. This means that the dotted-decimal string `255.255.255.255` cannot be handled by this function, since its value appears to indicate function failure. This function is deprecated and any new code should use `inet_aton` or `inet_pton` (described in the next section) instead.

The `inet_ntoa` converts the 32-bit binary, network byte ordered, IPv4 address into its string counterpart. The string pointed to by the return value resides in static memory, that is, the function is *not* reentrant. Note that the function's argument is a structure, and not a pointer to a structure.

2.5 `inet_pton` and `inet_ntop` functions

These two functions can be used with IPv6 as well as IPv4. The letters `p` and `n` stand for *presentation* and *numeric*.

```
#include <arpa/inet.h>
int inet_pton(int family, const char *strptr, void *addrptr);
```

Returns: 1 if OK, 0 if input is not a valid presentation format, -1 on error

```
const char *inet_ntop(int family, const void *addrptr,
                      char *strptr, size_t len);
```

Returns: pointer to result if OK, `NULL` on error

The `family` argument for both functions is either `AF_INET` or `AF_INET6` (for IPv6). These functions are new with IPv6, and are not available in systems without IPv6 support.

The first function tries to convert the string pointed to by `strptr`, storing the result through the `addrptr` argument. The second function does the reverse conversion. The `len` argument is the the size of the destination, preventing the function from overflowing the caller's buffer. The following two macros are defined in `<netinet/in.h>` to help speciy the size:

```
#define INET_ADDRSTRLEN  16 /* for IPv4 dotted-decimal */
#define INET6_ADDRSTRLEN 48 /* for IPv6 hex string     */
```

3 Basic TCP Sockets

We now turn to describing the basic socket functions required to write a basic TCP client and server.

3.1 `socket` function

The first thing a process that wants to perform network I/O must do is call the `socket` function.

```
#include<sys/socket.h>
int socket(int family, int type, int protocol);

Returns: nonnegative descriptor if OK, -1 on error
```

The `family` argument specifies the protocol family. For our purposes, we set it to `AF_INET`. The socket `type` is set to `SOCK_STREAM` for TCP sockets or, in the case of UDP sockets, to `SOCK_DGRAM`. The `protocol` argument is normally set to 0 unless we are dealing with raw sockets, which we are not.

You may also encounter the constant `PF_INET` passed as the first argument to the `socket` function. The `PF_` prefix stands for "protocol family", as opposed to the `AF_` prefix, that stands for "address family". This topic will not be delved into any deeper, but we note that the `<sys/socket.h>` header defines `PF_INET` to be equal to the value of `AF_INET`.

3.2 `connect` function

The `connect` function is used by a TCP client to establish a connection with a TCP server.

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

Returns: 0 if OK, -1 on error
```

`sockfd` is a socket descriptor returned by the `socket` function. The second and third arguments are a pointer to a socket address structure, and it's size. The socket address structure must contain the IP address and port number of the server. When using TCP sockets, `connect` initialises TCP's three-way handshake; the function returns when a connection is established or when an error occurs.

3.3 `bind` function

The `bind` function assigns a local protocol address to a socket. Using the Internet protocols, this address is a combination of a 32-bit IPv4 address or a 128-bit IPv6 address, along with a a 16-bit TCP or UDP port number.

```
#include <sys/socket.h>
int bind(int sockfd, 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. With TCP, calling bind lets us specify a port number, an IP address, both, or neither. Servers bind their well-known

sin_addr	sin_port	Result of bind call
INADDR_ANY	0	kernel chooses IP address and port
INADDR_ANY	nonzero	kernel chooses IP address, process specifies port
local IP address	0	process specifies IP address, kernel chooses port
local IP address	nonzero	process specifies IP address and port

Figure 4: Specifying IP address or port number or both.

port when they start. (For example, a day-time server—that returns the current date and time to the client—would bind port 13.)

A process can `bind` a specific IP address to its socket. The IP address must belong to an interface on the host. For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket. For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address. Usually, a TCP client does not `bind` an IP address to its socket, letting the kernel decide the source IP address to use when the socket is connected.

Before calling bind, the socket address structure pointed to by the `myaddr` argument must be set up to contain the desired IP address and port number. Figure 4 lists the values to which we set the these members, depending on the desired result.

3.4 `listen` function

The `listen function` is only called by TCP servers. When a socket has been created with the `socket` function, it is assumed to be an *active* socket, that is, a client socket that will connect to a server. By calling `listen`, the active socket is converted to a *passive* socket, indicating that the kernel should listen for connection requests directed to this socket.

```
#include<sys/socket.h>
int listen(int sockfd, int backlog);
```

Returns: 0 if OK, -1 on error

The second argument specifies the maximum number of connections that the kernel should queue for this socket. (The *actual* meaning of the `backlog` argument is a bit complicated. We will not go into any details here, but [Ste98] does.)

3.5 `accept` function

A TCP server calls `accept` to be returned the next completed connection from the front of the completed connection queue. If the completed connection queue is empty, the server is put to sleep (assuming the default of a blocking socket).

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

Returns: nonnegative descriptor if OK, -1 on error

The `sockfd` argument is a passive socket, that is, a listening socket. The `cliaddr` and `addrlen` arguments are used to return the protocol address of the connected client. Note that `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.

3.6 Reading and Writing Data

TCP sockets exhibit a behaviour with the `read` and `write` functions that is different to normal file I/O. A `read` or `write` on a TCP socket might input our output fewer bytes than requested, but this is not an error condition. This is because the buffer limits for the socket might have been reached in the kernel. The caller needs only to call `read` or `write` again, until the remaining bytes have been read or written.

If we want to read *n* bytes from a TCP socket, we should always make sure that *n* bytes actually got read, and if not, call `read` again, as many times as necessary. The same goes for writing to a socket.

3.7 `close()` function

The ordinary 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 immediately return to the process. The socket is no longer usable by the process, but TCP will still send any data that has been queued to be sent to the other end. Only after this, is the normal TCP connection termination sequence begun.

Note that, unless a socket's reference count is 1, calling `close` only decrements the reference count. The TCP four-packet connection termination sequence is not initiated until the reference count is 0 after being decremented. Another way of terminating a TCP connection is to use the `shutdown` function instead of `close`. We do not discuss `shutdown` any further.

4 A little something about UDP Sockets

UDP is a connectionless, unreliable, datagram protocol, quite different from the connection-oriented, reliable byte stream provided by TCP.

As mentioned in 3.1, when creating an UDP socket with the `socket` function, we pass `SOCK_DGRAM` as the second argument. Also, when reading and writing data to and from an UDP socket, we use the `recvfrom` and `sendto` functions.

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
               const struct sockaddr *to, socklen_t addrlen);

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
               struct sockaddr *from, socklen_t *addrlen);
```

Returns: number of bytes read or written if OK, -1 on error

The first three arguments, `sockfd`, `buff`, and `nbytes` are identical to the arguments for the `read` and `write` functions: descriptor, pointer to buffer to read from, or write to, and the number of bytes to read or write.

We do not cover the `flags` argument here. For simple client-server applications, it is set to 0.

The `to` argument is a socket address structure containing the IP address and port number of where the data is to be sent. The size of this structure is specified by `addrlen`.

The `recvfrom` function fills in the socket address structure pointed to by `from` with the protocol address of who sent the datagram. The number of bytes stored in this structure is also returned to the caller in the integer pointed to by `addrlen`. If `from` is set to `NULL`; then `addrlen` must be a null pointer as well. This indicates that the caller of `recvfrom` is not interested in knowing the protocol address of whoever sends data.

5 Port numbers

Both TCP and UDP define a group of well-known ports to identify well-known services. For example, FTP servers are assigned port 21, web servers port 80, etc. On the other hand, clients normally use short-lived ports, assigned automatically by TCP or UDP.

The Internet Assigned Numbers Authority (IANA) divide port numbers into three ranges:

1. The well-known ports: 0 through 1023. These numbers are controlled and assigned by IANA.
2. The registered ports: 1024 through 49151. These numbers are not controlled by the IANA, but IANA registers and lists uses of these ports as a convenience to the community. For example, ports 6000 through 6063 are assigned for X window servers.
3. The dynamic or private ports: 49152 through 65535. The IANA says nothing about these ports. These ports are sometimes called *ephemeral* ports. (Ephemeral meaning lasting a very short time.)

UNIX systems have the concept of a *reserved* port, that is, any port less than 1024. These ports can only be assigned to a socket by a superuser process.

With respect to history, Berkeley-derived implementations have allocated ephemeral ports in the range 1024-5000. In the early 1980s, this was OK, but today it is not difficult to find a host that can support more than 3977 clients at any given time. Consequently, some systems try to provide more ephemeral ports. Solaris is an example of this, allocating ephemeral ports in the range 32768-65535.

References

[Ste98] W. Richard Stevens. *UNIX Network Programming Volume 1: Networking APIs: Sockets and XTI.* Prentice Hall, Upper Saddle River, NJ 07458, second edition, 1998.