| Winsock & .NET | Winsock | < TCP/IP Client Server Model | Linux Socket Index | More TCP/IP Programming Interfaces (APIs) > |

# NETWORK PROGRAMMING
# LINUX SOCKET PART 3: MORE ON APIs



My Training Period: xx hours

Note: Program examples if any, compiled using gcc on **Linux Fedora Core 3** machine with several update, as normal user. The Fedora machine used for the testing having the "No Stack Execute" disabled and the **SELinux** set to default configuration.

**Generic Socket Address Structure**

**Host IP Addresses**

- Each computer on the Internet has one or more Internet addresses, numbers which identify that computer among all those on the Internet.
- Users typically write numeric host addresses as sequences of four numbers, separated by periods, as in 128.54.46.100.
- Each computer also has one or more host names, which are strings of words separated by periods, as in www.google.com.
- Programs that let the user specify a host typically accept both numeric addresses and host names.
- But the program needs a numeric address to open a connection; to use a host name; you must convert it to the numeric address it stands for.

**Internet Host Addresses – Abstract Host Address**

- Each computer on the Internet has one or more Internet addresses, numbers which identify that computer among all those on the Internet.
- An Internet host address is a number containing four bytes of data. These are divided into two parts, a network number and a local network address number within that network.
- The network number consists of the first one, two or three bytes; the rest of the bytes are the local address.
- Network numbers are registered with the Network Information Center (NIC), and are divided into three classes as discussed before: class A, B, and C for the IPv4. The local network address numbers of individual machines are registered with the administrator of the particular network.
- Since a single machine can be a member of multiple networks, it can have multiple Internet host addresses.
- However, there is never supposed to be more than one machine with the

Advanced TCP/IP 3
Advanced TCP/IP 4
Advanced TCP/IP 5

same host address.

- There are four forms of the standard numbers-and-dots notation for Internet addresses as discussed before:

1. **a.b.c.d** - This specifies all four bytes of the address individually.
2. **a.b.c** - The last part of the address, **c**, is interpreted as a 2-byte quantity. This is useful for specifying host addresses in a Class B network with network address number **a.b**.
3. **a.b** - The last part of the address, **c**, is interpreted as a 3-byte quantity. This is useful for specifying host addresses in a Class A network with network address number **a**.
4. **a** - If only one part is given, this corresponds directly to the host address number.

- Within each part of the address, the usual C conventions for specifying the radix apply. In other words, a leading '0x' or '0X' implies hexadecimal radix; a leading '0' implies octal; and otherwise decimal radix is assumed.

**Host Address Data Type -** Data type for a host number.

- Internet host addresses are represented in some contexts as integers (type unsigned long int).
- In other contexts, the integer is packaged inside a structure of type struct in_addr. It would be better if the usages were made consistent, but it is not hard to extract the integer from the structure or put the integer into a structure.
- The following basic definitions for Internet addresses appear in the header file '`in.h`'.

```
struct in_addr
```

- This data type is used in certain contexts to contain an Internet host address. It has just one field, named s_addr, which records the host address number as an unsigned long int.

```
unsigned long int INADDR_LOOPBACK
```

- You can use this macro constant to stand for the "address of this machine" instead of finding its actual address.
- It is the Internet address '127.0.0.1', which is usually called 'localhost'. This special constant saves you the trouble of looking up the address of your own machine.
- Also, the system usually implements INADDR_LOOPBACK specially, avoiding any network traffic for the case of one machine talking to itself.

```
unsigned long int INADDR_ANY
```

- You can use this macro constant to stand for "any incoming address" when binding to an address. This is the usual address to give in the sin_addr member of struct sockaddr_in when you want your server to accept Internet connections.

```
unsigned long int INADDR_BROADCAST
```

- This macro constant is the address you use to send a broadcast message.

```
unsigned long int INADDR_NONE
```

- This macro constant is returned by some functions to indicate an error.

**Host Address Functions - Functions to operate on them**

- These additional functions for manipulating Internet addresses are declared in 'arpa/inet.h'. They represent Internet addresses in network byte order; they represent network numbers and local-address-within-network numbers in host byte order.

```
int inet_aton(const char *name, struct in_addr *addr)
```

- This function converts the Internet host address name from the standard numbers-and-dots notation into binary data and stores it in the struct in_addr that addr points to.
- inet_aton returns nonzero if the address is valid, zero if not.

```
unsigned long int inet_addr(const char *name)
```

- This function converts the Internet host address name from the standard numbers-and-dots notation into binary data. If the input is not valid, inet_addr returns INADDR_NONE.
- This is an obsolete interface to inet_aton, described above; it is obsolete because INADDR_NONE is a valid address (255.255.255.255), and inet_aton provides a cleaner way to indicate error return.

```
unsigned long int inet_network(const char *name)
```

- This function extracts the network number from the address name, given in the standard numbers-and-dots notation. If the input is not valid, inet_network returns -1.

```
char * inet_ntoa(struct in_addr addr)
```

- This function converts the Internet host address addr to a string in the standard numbers-and-dots notation. The return value is a pointer into a statically-allocated buffer.
- Subsequent calls will overwrite the same buffer, so you should copy the string if you need to save it.

```
struct in_addr inet_makeaddr(int net, int local)
```

- This function makes an Internet host address by combining the network number net with the local-address-within-network number local.

```
int inet_lnaof(struct in_addr addr)
```

- This function returns the local-address-within-network part of the Internet host address addr.

```
Function int inet_netof(struct in_addr addr)
```

- This function returns the network number part of the Internet host address addr.

**Host Names - Translating host names to host IP numbers**

- Besides the standard numbers-and-dots notation for Internet addresses, you can also refer to a host by a symbolic name.
- The advantage of a symbolic name is that it is usually easier to remember. For example, the machine with Internet address '128.52.46.32' is also known as 'testo.google.com'; and other machines in the 'google.com' domain can refer to it simply as 'testo'.
- Internally, the system uses a database to keep track of the mapping between host names and host numbers.

- This database is usually either the file '/etc/hosts' or an equivalent provided by a name/DNS server. The functions and other symbols for accessing this database are declared in 'netdb.h'. They are BSD features, defined unconditionally if you include 'netdb.h'.
- The IP address to name and vice versa is called name resolution. It is done by Domain Name Service. Other than the hosts file, in Windows platform it is called DNS (Domain Name Service) and other Microsoft specifics may use WINS or lmhost file. Keep in mind that the general term actually Domain Name System also has DNS acronym. In UNIX it is done by BIND.
- The complete process or steps taken for name resolution quite complex but Windows normally use DNS service and UNIX/Linux normally use BIND.

**Data Type struct hostent**

- This data type is used to represent an entry in the hosts database. It has the following members:

| Data Type | Description |
|---|---|
| `char *h_name` | This is the "official" name of the host. |
| `char **h_aliases` | These are alternative names for the host, represented as a null-terr |
| `int h_addrtype` | This is the host address type; in practice, its value is always AF_IN represented in the data base as well as Internet addresses; if this w than AF_INET. |
| `int h_length` | This is the length, in bytes, of each address. |
| `char **h_addr_list` | This is the vector of addresses for the host. Recall that the host mi different addresses on each one. The vector is terminated by a nul |
| `char *h_addr` | This is a synonym for h_addr_list[0]; in other words, it is the first ho |

Table 5

- As far as the host database is concerned, each address is just a block of memory h_length bytes long.
- But in other contexts there is an implicit assumption that you can convert this to a struct in_addr or an unsigned long int. Host addresses in a struct hostent structure are always given in network byte order.
- You can use gethostbyname() or gethostbyaddr() to search the hosts database for information about a particular host. The information is returned in a statically-allocated structure.
- You must copy the information if you need to save it across calls.

```
struct hostent * gethostbyname(const char *name)
```

- The gethostbyname() function returns information about the host named name. If the lookup fails, it returns a null pointer.

```
struct hostent * gethostbyaddr(const char *addr, int length, int
format)
```

- The gethostbyaddr() function returns information about the host with Internet address addr. The length argument is the size (in bytes) of the address at addr.
- format specifies the address format; for an Internet address, specify a value of AF_INET.
- If the lookup fails, gethostbyaddr() returns a null pointer.
- If the name lookup by gethostbyname() or gethostbyaddr() fails, you can find out the reason by looking at the value of the variable h_errno.
- Before using h_errno, you must declare it like this:

```
extern  int   h_errno;
```

- Here are the error codes that you may find in h_errno:

| h_errno | Description |
|---|---|
| HOST_NOT_FOUND | No such host is known in the data base. |
| TRY_AGAIN | This condition happens when the name server could not be contacte |
| NO_RECOVERY | A non-recoverable error occurred. |
| NO_ADDRESS | The host database contains an entry for the name, but it doesn't hav |

Table 6

- You can also scan the entire hosts database one entry at a time using sethostent(), gethostent(), and endhostent(). Be careful in using these functions, because they are not reentrant.

**Function void sethostent(int stayopen)**

- This function opens the hosts database to begin scanning it. You can then call gethostent() to read the entries.
- If the stayopen argument is nonzero, this sets a flag so that subsequent calls to gethostbyname() or gethostbyaddr() will not close the database (as they usually would).
- This makes for more efficiency if you call those functions several times, by avoiding reopening the database for each call.

**Function struct hostent * gethostent()**

- This function returns the next entry in the hosts database. It returns a null pointer if there are no more entries.

**Function void endhostent()**

- This function closes the hosts database.

**The API Details**

- In this section and that follows we will discuss the socket APIs details: the structures, functions, macros and types.

**struct sockaddr**

```
struct sockaddr {
    u_char  sa_len;
    u_short sa_family;      // address family, AF_xxx
    char    sa_data[14];    // 14 bytes of protocol address
};
```

- sockaddr consists of the following parts:

1. The short integer that defines the address family (the value that is specified for address family on the socket() call).
2. Fourteen bytes that are reserved to hold the address itself.

- Originally sa_len was not there.
- Depending on the address family, sa_data could be a file name or a socket endpoint.
- sa_family can be a variety of things, but it'll be AF_INET for everything we do in this Tutorial.

- sa_data contains a destination address and port number for the socket. This is rather unwieldy since you don't want to tediously pack the address in the sa_data by hand.
- To deal with struct sockaddr, programmers created a parallel structure: struct sockaddr_in ("in" for "Internet".)

**struct sockaddr_in**

```
struct sockaddr_in {
    u_char  sin_len;
    u_short sin_family;        // Address family
    u_short sin_port;          // Port number
    struct  in_addr sin_addr;  // Internet or IP address
    char    sin_zero[8];       // Same size as struct sockaddr
};
```

- The sin_family field is the address family (always AF_INET for TCP and UDP).
- The sin_port field is the port number, and the sin_addr field is the Internet address. The sin_zero field is reserved, and you must set it to hexadecimal zeroes.
- Data type struct in_addr - this data type is used in certain contexts to contain an Internet host address. It has just one field, named s_addr, which records the host address number as an unsigned long int.
- sockaddr_in is a "specialized" sockaddr.
- sin_addr could be u_long.
- sin_addr is 4 bytes and 8 bytes are unused.
- sockaddr_in is used to specify an endpoint.
- The sin_port and sin_addr must be in Network Byte Order.

**Socket System Calls**

**socket()**

```
NAME
       socket() - create an endpoint for communication
SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>
       int socket(int domain, int type, int protocol);
```

- domain should be set to "AF_INET", just like in the struct sockaddr_in.
- The type argument tells the kernel what kind of socket this is. For example SOCK_STREAM or SOCK_DGRAM.
- Just set protocol to "0" to have socket() choose the correct protocol based on the type.
- protocol is frequently 0 if only one protocol in the family supports the specified type. You can look at /etc/protocols.
- There are many more domains and types that you will find later on.
- Also, there's a "better" way to get the protocol. See the getprotobyname() man page.
- socket() simply returns to you an integer of the socket descriptor that you can use in later system calls, or -1 on error. The global variable errno is set to the error's value (see the perror() man page).
- In some documentation, you'll see the mentioning of a mystical "PF_INET.
- Once a long time ago, it was thought that maybe an **address family** (what the "AF" in "AF_INET" stands for) might support several protocols that were referenced by their **protocol family** (what the "PF" in "PF_INET" stands for). That didn't happen.
- So the correct thing to do is to use AF_INET in your struct sockaddr_in and PF_INET in your call to socket(). But practically speaking, you can use AF_INET everywhere.

**bind()**

```
NAME
       bind() - bind a name to a socket
SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>
       int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- sockfd is the socket file descriptor returned by socket().
- my_addr is a pointer to a struct sockaddr that contains information about your address, namely, port and IP address.
- addrlen can be set to sizeof(struct sockaddr).
- Bind attaches a local endpoint to a socket.
- Once you have a socket, you might have to associate that socket with a port on your local machine.
- Typically servers use bind() to attach to well-known ports so clients can connect.
- This is commonly done if you're going to listen() for incoming connections on a specific port.
- The port number is used by the kernel to match an incoming packet to a certain process's socket descriptor. If you're going to only be doing a connect() that is just a client, this may be unnecessary.
- Let's have an example:

```
[bodo@bakawali testsocket]$ cat test1.c
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define MYPORT 3334

int main()
{
int sockfd;  /* socket file descriptor */
struct sockaddr_in my_addr;

sockfd = socket(AF_INET, SOCK_STREAM, 0);
if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
```

```
  perror("Server-socket() error lol!");
  exit(1);
}
else
  printf("Server-socket() sockfd is OK...\n");

/* host byte order */
my_addr.sin_family = AF_INET;
/* short, network byte order */
my_addr.sin_port = htons(MYPORT);
my_addr.sin_addr.s_addr = INADDR_ANY;
/* zero the rest of the struct */
memset(&(my_addr.sin_zero), 0, 8);

if(bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct
sockaddr)) == -1)
{
  perror("Server-bind() error lol!");
  exit(1);
}
else
  printf("Server-bind() is OK...\n");

/*....other codes....*/

return 0;
}

[bodo@bakawali testsocket]$ gcc test1.c -o test1
[bodo@bakawali testsocket]$ ./test1
Server-socket() sockfd is OK...
Server-bind() is OK...
```

- my_addr.sin_port and my_addr.sin_addr.s_addr are in Network Byte Order.
- For bind(), some of the process of getting your own IP address and/or port can be automated:

```
/* choose an unused port at random */
my_addr.sin_port = 0;
/* use my IP address */
my_addr.sin_addr.s_addr = INADDR_ANY;
```

- By setting my_addr.sin_port to zero, you are telling bind() to choose the port for you.
- Likewise, by setting my_addr.sin_addr.s_addr to INADDR_ANY, you are telling it to automatically fill in the IP address of the machine the process is running on.
- INADDR_ANY is actually zero. For 0.0.0.0, it means any IP.

```
/* choose an unused port at random */
my_addr.sin_port = htons(0);
/* use my IP address */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

- Now our code quite portable.
- bind() also returns -1 on error and sets errno to the error's value.
- When calling bind(), don't go below 1024 for your port numbers. All ports below 1024 are

reserved (unless you're the superuser/root or SUID type access).
- You can have any port number above that, right up to 65535 (216) provided they aren't already being used by another program.
- Sometimes, you might notice, you try to rerun a server and bind() fails, claiming "Address already in use". This means a socket that was connected is still hanging around in the kernel, and it's hogging the port.
- You can either wait for it to clear (a minute or so), or add code to your program allowing it to reuse the port, like this:

```
int yes = 1;
/* "Address already in use" error message */
if(setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes,
sizeof(int)) == -1)
{
  perror("setsockopt() error");
  exit(1);
}
else
  printf("setsockopt() is OK.\n");
```

- There are times when you won't absolutely have to call bind(). If you are connecting to a remote machine and you don't care what your local port is (as is the case with telnet where you only care about the remote port), you can simply call connect(), it'll check to see if the socket is unbound, and will bind() it to an unused local port if necessary.
- For the ports that has been blocked or closed by the firewall for security reason, you have to open it for communication.
- And if the access to the port denied, you have to allow it.
- Standard ports with their respective services have been defined in /etc/protocol. You can define ports for specific services by editing the /etc/protocol.
- Then with the newly defined ports, the new service is defined and created in /etc/xinetd.d. Please check the man pages for xinetd service (inetd is the older one).
- The following section summarizes the sockets related function prototypes.

**connect()**

```
NAME
       connect() - initiate a connection on a socket.
SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>
       int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

- sockfd is our friendly neighborhood socket file descriptor, as returned by the socket() call, serv_addr is a struct sockaddr containing the destination port and IP address
- addrlen can be set to sizeof(struct sockaddr).
- As an example, for telnet client application, firstly, get a socket file descriptor. Then if no error, we are ready to connect to remote host, let say "127.0.0.1" on port "23", the standard telnet port. For this we need connect().
- Let's have a snippet code example:

```
[bodo@bakawali testsocket]$ cat test2.c
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#define DEST_IP "127.0.0.1"
#define DEST_PORT 80

int main(int argc, char *argv[ ])
{
int sockfd;
/* will hold the destination addr */
struct sockaddr_in dest_addr;

sockfd = socket(AF_INET, SOCK_STREAM, 0);

if(sockfd == -1)
{
  perror("Client-socket() error lol!");
  exit(1);
}
else
  printf("Client-socket() sockfd is OK...\n");

/* host byte order */
dest_addr.sin_family = AF_INET;
/* short, network byte order */
dest_addr.sin_port = htons(DEST_PORT);
dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);

/* zero the rest of the struct */
memset(&(dest_addr.sin_zero), 0, 8);

if(connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct
sockaddr)) == -1)
{
  perror("Client-connect() error lol");
  exit(1);
}
else
  printf("Client-connect() is OK...\n");

/*...other codes...*/

return 0;
}

[bodo@bakawali testsocket]$ gcc test2.c -o test2
[bodo@bakawali testsocket]$ ./test2
Client-socket() sockfd is OK...
Client-connect() error lol: Connection refused
```

- Again, be sure to check the return value from connect(). It will return -1 on error and set the variable errno.
- Also, notice that we didn't call bind(). Basically, we don't care about our local port number; we only care where we're going to connect to that is the remote port.
- The kernel will choose a local port for us, and the site we connect to will automatically get this information from us.

*Continue on next Module.....*More in-depth discussion about TCP/IP suite is given in Advanced

TCP/IP tutorials.

**Further reading and digging:**

1. Check the best selling C/C++, Networking, Linux and Open Source books at Amazon.com.
2. Protocol sequence diagram examples.
3. Another site for protocols information.
4. RFCs.
5. GCC, GDB and other related tools.

---

| Winsock & .NET | Winsock | < TCP/IP Client Server Model | Linux Socket Index | More TCP/IP Programming Interfaces (APIs) > |