NETWORK PROGRAMMING
LINUX SOCKET PART 5: APIs & HEADER FILES

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.

sendto() and recvfrom() for DATAGRAM (UDP)

- Since datagram sockets aren't connected to a remote host, we need to give the destination address before we send a packet. The prototype is:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const
struct sockaddr *to, int tolen);
```

- This call is basically the same as the call to send() with the addition of two other pieces of information.
- to is a pointer to a struct sockaddr (which you'll probably have as a struct sockaddr_in and cast it at the last minute) which contains the destination IP address and port.
- tolen can simply be set to sizeof(struct sockaddr).
- Just like with send(), sendto() returns the number of bytes actually sent (which, again, might be less than the number of bytes you told it to send!), or -1 on error.
- Equally similar are recv() and recvfrom(). The prototype of recvfrom() is:

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct
sockaddr *from, int *fromlen);
```

- Again, this is just like recv() with the addition of a couple fields.
- from is a pointer to a local struct sockaddr that will be filled with the IP address and port of the originating machine.
- fromlen is a pointer to a local int that should be initialized to sizeof(struct sockaddr). When the function returns, fromlen will contain the length of the address actually stored in from. recvfrom() returns the number of bytes received, or -1 on error (with errno set accordingly).
- Remember, if you connect() a datagram socket, you can then simply use send() and recv() for all your transactions.
- The socket itself is still a datagram socket and the packets still use UDP, but the socket interface will automatically add the destination and source information for you.

A sample of the client socket call flow

```
socket()
connect()
while (x)
{
   write()
   read()
}
close()
```

A sample of the server socket call flow

```
socket()
bind()
listen()
while (1)
{
   accept()
   while (x)
{
      read()
```

```
        write()
}
   close()
}
close()
```

Network Integers versus Host Integers

- Little Endian and big Endian issue regarding the use of the different processor architectures.
- Usually integers are either most-significant byte first or least-significant byte first.
- On Intel based machines the hex value 0x01020304 would be stored in 4 successive bytes as:

  04, 03, 02, 01. This is a little endian.

- On an Most Significant Bit (MSB)-first (big endian) machine (IBM RS6000), this would be: 01, 02, 03, 04.
- It is important to use network byte order (MSB-first) and the conversion functions available for this task are listed below:

`htons()`	Host to network short.
`ntohs()`	Network to host short.
`htonl()`	Host to network long.
`ntohl()`	Network to host long.

Table 7

- Use these functions to write portable network code.
- Fortunately for you, there are a bunch of functions that allow you to manipulate IP addresses. No need to figure them out by hand and stuff them in a long with the « operator.
- First, let's say you have a:

```
struct sockaddr_in ina
```

- And you have an IP address "10.12.110.57" that you want to store into it.
- The function you want to use, inet_addr(), converts an IP address in numbers-and-dots notation into an unsigned long. The assignment can be made as follows:

```
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

- Notice that inet_addr() returns the address in Network Byte Order already so you don't have to call htonl().
- Now, the above code snippet isn't very robust because there is no error checking. inet_addr() returns -1 on error.
- For binary numbers (unsigned)-1 just happens to correspond to the IP address 255.255.255.255! That's the broadcast address! Remember to do your error checking properly.
- Actually, there's a cleaner interface you can use instead of inet_addr(): it's called inet_aton() ("aton" means "ascii to network"):

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
```

- And here's a sample usage, while packing a struct sockaddr_in is shown below:

```
struct sockaddr_in my_addr;
/* host byte order */
my_addr.sin_family = AF_INET;
/* short, network byte order */
my_addr.sin_port = htons(MYPORT);
inet_aton("10.12.110.57", &(my_addr.sin_addr));
/* zero the rest of the struct */
memset(&(my_addr.sin_zero), 0, 8);
```

- inet_aton(), unlike practically every other socket-related function, returns non-zero on success, and zero on failure. And the address is passed back in inp.
- Unfortunately, not all platforms implement inet_aton() so, although its use is preferred, normally the older more common inet_addr() is used.
- All right, now you can convert string IP addresses to their binary representations. What about the other way around?
- What if you have a struct in_addr and you want to print it in numbers-and-dots notation? In this case, you'll want to use the function

inet_ntoa() ("ntoa" means "network to ascii") something like this:

```
printf("%s", inet_ntoa(ina.sin_addr));
```

- That will print the IP address. Note that inet_ntoa() takes a struct in_addr as an argument, not a long. Also notice that it returns a pointer to a char.
- This points to a statically stored char array within inet_ntoa() so that each time you call inet_ntoa() it will overwrite the last IP address you asked for. For example:

```
char *a1, *a2;
...
...
a1 = inet_ntoa(ina1.sin_addr); /* this is 192.168.4.1 */
a2 = inet_ntoa(ina2.sin_addr); /* this is 10.11.110.55 */
printf("address 1: %s\n", a1);
printf("address 2: %s\n", a2);
```

- Will print:

```
address 1: 10.11.110.55
address 2: 10.11.110.55
```

- If you need to save the address, strcpy() it to your own character array.

----------**SOME SUMMARY**----------

- Let see, what we have covered till now.

Socket Library Functions

- System calls:
  1. Startup / close.
  2. Data transfer.
  3. Options control.
  4. Other.
- Network configuration lookup:
  1. Host address.
  2. Ports for services.
  3. Other.
- Utility functions:
  1. Data conversion.
  2. Address manipulation.
  3. Error handling.

Primary Socket Calls

`socket()` `bind()`	Create a new socket and return its descriptor. Associate a socket with a port and address.
`listen()`	Establish queue for connection requests.

`accept()`
`connect()`
`recv()`
	Accept a connection request.
	Initiate a connection to a remote host.
	Receive data from a socket descriptor.

`send()`
`read()`
`write()`
	Send data to a socket descriptor.
	Reads from files, devices, sockets etc.
	Writes to files, devices, sockets etc.

`close()` `shutdown()`	“One-way” close of a socket descriptor. Allows you to cut off communication in a certain direction, or both ways just like close() does.

Table 8

Network Database Administration functions

- gethostbyname() - given a hostname, returns a structure which specifies its DNS name(s) and IP address (es).
- getservbyname() - given service name and protocol, returns a structure which specifies its name(s) and its port address.
- gethostname() - returns hostname of local host.
- getservbyname(), getservbyport(), getservent().
- getprotobyname(), getprotobynumber(), getprotobyent(), getnetbyname(), getnetbyaddr(), getnetent().

Socket Utility Functions

ntohs()/ntohl()	Convert short/long from network byte order (big endian) to host byte order.
htons()/htonl()	Convert short/long from host byte order to network byte order.
inet_ntoa()/inet_addr()	Convert 32-bit IP address (network byte order to/from a dotted decimal string).
perror()	Print error message (based on "errno") to stderr.
herror()	Print error message for gethostbyname() to stderr (used with DNS).

Table 9

Primary Header Files

- Include file sequence may affect processing (order is important!). Other header files that define macro, data type, structure and functions are given in the summary Table at the end of this Tutorial.

<sys/types.h>	Prerequisite typedefs.
<errno.h>	Names for "errno" values (error numbers).
<sys/socket.h>	struct sockaddr; system prototypes and constants.
<netdb.h.h>	Network info lookup prototypes and structures.
<netinet/in.h>	struct sockaddr_in; byte ordering macros.
<arpa/inet.h>	Utility function prototypes.

Table 10

Ancillary Socket Topics

- UDP versus TCP.
- Controlling/managing socket characteristics.

  1. get/setsockopt() - keepalive, reuse, nodelay.
  2. fcntl() - async signals, blocking.
  3. ioctl() - file, socket, routing, interface options.

- Blocking versus Non-blocking socket.
- Signal based socket programming (SIGIO).
- Implementation specific functions.

Socket header files

- Programs that use the socket functions must include one or more header files that contain information that is needed by the functions, such as:

  1. Macro definitions.
  2. Data type definitions.
  3. Structure definitions.
  4. Function prototypes.

- The following Table is a summary of the header files used in conjunction with the socket APIs. However, different kernel version will have slightly different header files and the path as well. Please refer to the online Linux source code repository for the desired kernel version.

Header file name	Description
<arpa/inet.h>	Defines prototypes for those network library routines that convert Internet address and dotted-decimal notation, for example, inet_makeaddr().
<arpa/nameser.h>	Defines Internet name server macros and structures that are needed when the system uses the resolver routines.
<error.h>	Defines macros and variables for error reporting.
<fcntl.h>	Defines prototypes, macros, variables, and structures for control-type functions, for example, fcntl().
<net/if.h>	Defines prototypes, macros, variables, and the ifreq and ifconf structures that are associated with ioctl() requests that affect interfaces.
<net/route.h>	Defines prototypes, macros, variables, and the rtentry and rtconf structures that are associated with ioctl() requests that affect routing entries.
<netdb.h>	Contains data definitions for the network library routines. Defines the following structures: ▪ hostent and hostent_data. ▪ netent and netent_data. ▪ servent and servent_data. ▪ protoent and protoent_data.
<netinet/in.h>	Defines prototypes, macros, variables, and the sockaddr_in structure to use with Internet domain sockets.
<netinet/ip.h>	Defines macros, variables, and structures that are associated with setting IP options.
<netinet/ip_icmp.h>	Defines macros, variables, and structures that are associated with the Internet Control Message Protocol (ICMP).
<netinet/tcp.h>	Defines macros, variables, and structures that are associated with setting TCP options.
<netns/idp.h>	Defines IPX packet header. May be needed in AF_NS socket applications.
<netns/ipx.h>	Defines ioctl structures for IPX ioctl() requests. May be needed in AF_NS socket applications.
<netns/ns.h>	Defines AF_NS socket structures and options. You must include this file in AF_NS socket applications.
<netns/sp.h>	Defines SPX packet header. May be needed in AF_NS socket applications.

<nettel/tel.h>	Defines sockaddr_tel structure and related structures and macros. You must include this file in AF_TELEPHONY socket applications.
<resolv.h>	Contains macros and structures that are used by the resolver routines.
<ssl.h>	Defines Secure Sockets Layer (SSL) prototypes, macros, variables, and the following structures: ▪ SSLInit ▪ SSLHandle
<sys/ioctl.h>	Defines prototypes, macros, variables, and structures for I/O control-type functions, for example, ioctl().
<sys/param.h>	Defines some limits to system fields, in addition to miscellaneous macros and prototypes.
<sys/signal.h>	Defines additional macros, types, structures, and functions that are used by signal routines.
<sys/socket.h>	Defines socket prototypes, macros, variables, and the following structures: ▪ sockaddr ▪ msghdr ▪ linger You must include this file in all socket applications.
<sys/time.h>	Defines prototypes, macros, variables, and structures that are associated with time functions.
<sys/types.h>	Defines various data types. Also includes prototypes, macros, variables, and structures that are associated with the select() function. You must include this file in all socket applications.
<sys/uio.h>	Defines prototypes, macros, variables, and structures that are associated with I/O functions.
<sys/un.h>	Defines prototypes, macros, variables, and the sockaddr_un structure to use with UNIX domain sockets.
<unistd.h>	Contains macros and structures that are defined by the integrated file system. Needed when the system uses the read() and write() system functions.

Table 11: Header files for Linux sockets APIs

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

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.