# NETWORK PROGRAMMING LINUX SOCKET PART 6: THE APIs

My Training Period: xx hours

This is a continuation from Part I series, Introduction to Socket Programming. Working program examples if any compiled using gcc, tested using the public IPs, run on **Linux**/**Fedora Core 3**, with several times of update, as normal user. The Fedora machine used for the testing having the "No Stack Execute" disabled and the SELinux set to default configuration.

**The abilities that supposed to be acquired:**

- Able to understand and use the Unix / Linux C language socket APIs.
- Able to understand and implement several simple TCP and UDP Client and server basic designs.

**Client Design Consideration**

- Some of the information in this section is a repetition from the previous one.

**Identifying a Server's Address**

- A server's IP address must be used in connect.
- Usually the name is used to get the address.
- The name could be in the code mailhost for an email program.
- The user could specify the name common because it is flexible and simple.
- The name or address could be in a file.
- Can broadcast on the network to ask for a server.
- The following is an example of telneting the telserv.test.com server through the standard telnet port 25:

```
telnet   telserv.test.com
```

- Or using the IP address of the telnet server:

```
telnet    131.95.115.204
```

- Client software typically allows either names or numbers.
- Ports usually have a default value in the code if not explicitly mentioned.

**Looking Up a Computer Name**

```
NAME
       gethostbyname() - get network host entry
SYNOPSIS
       #include <netdb.h>
       extern int h_errno;
       struct hostent
       *gethostbyname(const char *name);
struct hostent {
   char  *h_name;
   char  **h_aliases;
   int   h_addrtype;
   int   h_length;
   char  **h_addr_list;
};
#define h_addr h_addr_list[0]
```

- name could be a name or dotted decimal address.
- Hosts can have many names in h_aliases.
- Hosts can have many addresses in h_addr_list.
- Addresses in h_addr_list are not strings network order addresses ready to copy and use.

**Looking Up a Port Number by Name**

```
NAME
   getservbyname() - get service entry
SYNOPSIS
   #include <netdb.h>
   struct servent *getservbyname(const char *name, const char *proto
struct servent {
   char  *s_name;
   char  **s_aliases;
   int   s_port;
   char  *s_proto;
}
```

- s_port: port number for the service given in network byte order.

**Looking Up a Protocol by Name**

```
NAME
```

```
       getprotobyname() - get protocol entry
SYNOPSIS
       #include <netdb.h>
       struct protoent
       *getprotobyname(const char *name);
struct protoent {
   char  *p_name;
   char  **p_aliases;
   int   p_proto;
}
```

- p_proto: the protocol number (can be used in socket call).

**getpeername()**

- The function getpeername() will tell you who is at the other end of a connected stream socket.
- The prototype:

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

- sockfd is the descriptor of the connected stream socket.
- addr is a pointer to a struct sockaddr (or a struct sockaddr_in) that will hold the information about the other side of the connection.
- addrlen is a pointer to an int, which should be initialized to sizeof(struct sockaddr).
- The function returns -1 on error and sets errno accordingly.
- Once you have their address, you can use inet_ntoa() or gethostbyaddr() to print or get more information.

**Allocating a Socket**

```
#include <sys/types.h>
#include <sys/socket.h>
int s;
s = socket(PF_INET, SOCK_STREAM, 0);
```

- Specifying PF_INET and SOCK_STREAM leaves the protocol parameter irrelevant.

**Choosing a Local Port Number**

- The server will be using a well-known port.
- Once a client port is set, the server will be aware as needed.
- You could bind to a random port above 1023.
- A simpler choice is to leave out the bind call.
- connect() will choose a local port if required.

**Connecting to a Server with TCP**

- connect().

```
NAME
       connect - initiate a connection on a socket
SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>
       int connect(int s, struct sockaddr *serv_addr, int addrlen);
RETURN VALUE
       If the connection or binding succeeds, zero is returned.
       On error, -1 is returned, and errno is set appropriately.
```

- We will use a sockaddr_in structure (possibly cast).
- After connect, s is available to read/write.

### Communicating with TCP

- Code segment example:

```
char *req = "send cash";
char buf[100], *b;
write (s, req, strlen(req));
left = 100;
b = buf;
while (left && (n = read(s, buf, left)) > 0)
{
   b += n;
   left -= n;
}
```

- The client and server can not know how many bytes are sent in each write.
- Delivered chunks are not always the same size as in the original write.
- Reads must be handled in a loop to cope with stream sockets.

### Closing a TCP Connection

- In the simplest case close works well.
- Sometimes it is important to tell the server that a client will send no more requests, while still keeping the socket available for reading.

```
res = shutdown(s, 1);
```

- The 1 means no more writes will happen.
- The server detects end of file on the socket.
- After the server sends all the replies it can close.

### Connected vs Unconnected UDP Sockets

- A client can call connect with a UDP socket or not.
- If connect is called read and write will work.
- Without connect the client needs to send with a system call specifying a remote endpoint.
- Without connect it might be useful to receive data with a system call which tells the remote endpoint.

- Connect with TCP involves a special message exchange sequence.
- Connect with UDP sends no messages. You can connect to non-existent servers.

### Communicating with UDP

- UDP data is always a complete message (datagram).
- Whatever is specified in a write becomes a datagram.
- Receiver receives the complete datagram unless fewer bytes are read.
- Reading in a loop for a single datagram is pointless with UDP.
- close() is adequate, since shutdown() does not send any messages.
- UDP is unreliable. UDP software needs an error protocol.

### Client Example – Some variations

### A Simple Client Library

- To make a connection, a client must:

  1. select UDP or TCP...
  2. determine a server's IP address...
  3. determine the proper port...
  4. make the socket call...
  5. make the connect call...

- Frequently the calls are essentially the same.
- A library offers normal capability with a simple interface.

### connectTCP()

- The following is a code segment example using the connectTCP() function.

```
int connectTCP(const char *host, const char *service)
{ return connectsock(host, service, "tcp"); }
```

### connectUDP()

- The following is a code segment example using the connectUDP() function.

```
int connectUDP(const char *host, const char *service)
{ return connectsock(host, service, "udp"); }
```

### connectsock()

- The following is a code segment example using the connectsock() function.

```
int connectsock(const char *host, const char *service, const char *t
{
    struct hostent      *phe;   /* pointer to host information entry
    struct servent      *pse;   /* pointer to service information en
    struct protoent     *ppe;   /* pointer to protocol information e
    struct sockaddr_in sin;     /* an Internet endpoint address
    int s, type;                /* socket descriptor and socket type
    memset(&sin, 0, sizeof(sin));
```

```
        sin.sin_family = AF_INET;
        /* Map service name to port number */
        if(pse = getservbyname(service, transport))
            sin.sin_port = pse->s_port;
        else if ((sin.sin_port = htons((u_ short)atoi(service))) == 0)
            errexit("can't get \"%s\" service entry\n", service);
        /* Map host name to IP address, allowing for dotted decimal */
        if(phe = gethostbyname(host))
            memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
        else if ((sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE)
            errexit("can't get \"%s\" host entry\n", host);
        /* Map transport protocol name to protocol number */
        if((ppe = getprotobyname(transport)) == 0)
            errexit("can't get \"%s\" protocol entry\n", transport);
        /* Use protocol to choose a socket type */
        if(strcmp(transport, "udp") == 0)
            type = SOCK_DGRAM;
        else
            type = SOCK_STREAM;
        /* Allocate a socket */
        s = socket(PF_INET, type, ppe->p_proto);
        if(s < 0)
           errexit("can't create socket: %s\n", strerror(errno));
        /* Connect the socket */
        if(connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
           errexit("can't connect to %s.%s: %s\n", host, service, strer
        return s;
    }
```

### A TCP DAYTIME Client

- DAYTIME service prints date and time.
- TCP version sends upon connection.  Server reads no client data.
- UDP version sends upon receiving any message.
- The following is a code segment example implementing the TCP Daytime.

```
    #define LINELEN 128
    int main(int argc, char *argv[ ])
    {
        /* host to use if none supplied */
        char *host = "localhost";
        /* default service port */
        char *service = "daytime";
        switch (argc) {
        case 1:
            host = "localhost";
            break;
        case 3:
            service = argv[2];
            /* FALL THROUGH */
        case 2:
            host = argv[1];
            break;
        default:
            fprintf(stderr, "usage: TCPdaytime [host [port]]\n");
            exit(1);
        }
        TCPdaytime(host, service);
        exit(0);
}
void TCPdaytime(const char *host, const char *service)
{
        /* buffer for one line of text */
        char buf[LINELEN+1];
        /* socket, read count */
        int s, n;
        s = connectTCP(host, service);
 while((n = read(s, buf, LINELEN)) > 0)
 {
      /* ensure null-terminated */
      buf[n] = '\0';
      (void) fputs(buf, stdout);
 }
}
```

### A UDP TIME Client

- The TIME service is for computers.
- Returns seconds since 1-1-1900.
- Useful for synchronizing and time-setting.
- TCP and UDP versions return time as an integer.
- Need to use ntohl to convert.
- The following is a code segment example implementing UDP Time.

```
#define MSG  "What time is it?\n"

int main(int argc, char *argv[ ])
{
    char   *host = "localhost"; /* host to use if none supplied */
    char   *service = "time";   /* default service name   */
    time_t now;                 /* 32-bit integer to hold time  */
    int    s, n;                /* socket descriptor, read count */
    switch (argc) {
    case 1:
        host = "localhost";
        break;
    case 3:
        service = argv[2];
        /* FALL THROUGH */
    case 2:
        host = argv[1];
        break;
```

```
            default:
                fprintf(stderr, "usage: UDPtime [host [port]]\n");
                exit(1);
            }
            s = connectUDP(host, service);
            (void) write(s, MSG, strlen(MSG));
            /* Read the time */
            n = read(s, (char *)&now, sizeof(now));
            if(n < 0)
                errexit("read failed: %s\n", strerror(errno));
            /* put in host byte order */
            now = ntohl((u_long)now);
            printf("%s", ctime(&now));
            exit(0);
        }
```

### TCP and UDP Echo Clients

- main() is like the other clients.

### TCPecho() function

- The following is a code segment example for using the TCPecho() function.

```
        int TCPecho(const char *host, const char *service)
        {
            char buf[LINELEN+1];   /* buffer for one line of text  */
            int s, n;              /* socket descriptor, read count*/
            int outchars, inchars; /* characters sent and received */
            s = connectTCP(host, service);
        while(fgets(buf, sizeof(buf), stdin))
        {
           /* insure line null-terminated */
           buf[LINELEN] = '\0';
           outchars = strlen(buf);
           (void) write(s, buf, outchars);
          /* read it back */
         for(inchars = 0; inchars < outchars; inchars+=n)
         {
            n = read(s, &buf[inchars], outchars - inchars);
            if(n < 0)
              errexit("socket read failed: %s\n", strerror(errno));
         }
         fputs(buf, stdout);
        }
        }
```

### UDPecho() function

- The following is a code segment example for using the UDPecho() function.

```
int UDPecho(const char *host, const char *service)
{
    /* buffer for one line of text */
    char buf[LINELEN+1];
    /* socket descriptor, read count */
    int s, nchars;
    s = connectUDP(host, service);
 while(fgets(buf, sizeof(buf), stdin))
 {
    /* ensure null-terminated */
    buf[LINELEN] = '\0';
    nchars = strlen(buf);
    (void) write(s, buf, nchars);
    if(read(s, buf, nchars) < 0)
        errexit("socket read failed: %s\n", strerror(errno));
    fputs(buf, stdout);
}
}
```

### Server Design Consideration

### Concurrent vs Iterative Servers

- An iterative server processes one request at a time.
- A concurrent server processes multiple requests at a time real or apparent concurrency.
- A single process can use asynchronous I/O to achieve concurrency.
- Multiple server processes can achieve concurrency.
- Concurrent servers are more complex.
- Iterative servers cause too much blocking for most applications.
- Avoiding blocking results in better performance.

### Connection-Oriented vs Connectionless Servers

- TCP provides a connection-oriented service.
- UDP provides a connectionless service.

### Connection-oriented Servers

- Easy to program, since TCP takes care of communication problems.

- Also a single socket is used for a single client exclusively (connection).
- Handling multiple sockets is intense juggling.
- For trivial applications the 3-way handshake is slow compared to UDP.
- Resources can be tied up if a client crashes.

### Connectionless Servers

- No resource depletion problem.
- Server/client must cope with communication errors. Usually client sends a request and resends if needed.
- Selecting proper timeout values is difficult.
- UDP allows broadcast/multicast.

### Stateless Servers

- Statelessness improves reliability at the cost of longer requests and slower performance.
- Improving performance generally adds state information. For example, adding a cache of file data.
- Crashing clients leave state information in server.
- You could use LRU replacement to re-use space.
- A frequently crashing client could dominate the state table, wiping out performance gains.
- Maintaining state information correctly and efficiently is complex.

### Request Processing Time

- Request processing time (rpt) = total time server uses to handle a single request.
- Observed response time (ort) = delay between issuing a request and receiving a response
- rpt <= ort.
- If the server has a large request queue, ort can be large.
- Iterative servers handle queued requests sequentially.
- With N items queued the average iterative (ort = N * rpt).
- With N items queued a concurrent server can do better.
- Implementations restrict queue size.
- Programmers need a concurrent design if a small queue is inadequate.

### Iterative, Connection-Oriented Server Algorithm

- The following is a sample of pseudo codes for iterative, connection oriented server.

```
create a socket
bind to a well-known port
place in passive mode
while (1)
{
    Accept the next connection
    while (client writes)
    {
        read a client request
        perform requested action
```

```
            send a reply
        }
        close the client socket
    }
    close the passive socket
```

### Using INADDR_ANY

- Some server computers have multiple IP addresses.
- A socket bound to one of these will not accept connections to another address.
- Frequently you prefer to allow any one of the computer's IP addresses to be used for connections.
- Use INADDR_ANY (0L) to allow clients to connect using any one of the host's IP addresses.

### Iterative, Connectionless Server Algorithm

- The following is a sample of pseudo codes for iterative, connectionless server.

```
create a socket
bind to a well-known port
while (1)
{
    read a request from some client
    send a reply to that client
}

recvfrom(s, buf, len, flags, from, fromlen)
sendto(s, buf, len, flags, to, to_len)
```

### Concurrent, Connectionless Server Algorithm

- The following is a sample of pseudo codes for concurrent, connectionless server.

```
create a socket
bind to a well-known port
while (1)
{
    read a request from some client
    fork
    if(child)
    {
        send a reply to that client
        exit
    }
  }
```

- Overhead of fork and exit is expensive.
- Not used much.

### Concurrent, Connection-Oriented Server Algorithm

- The following is a sample of pseudo codes for connection-oriented server.

```
create a socket
bind to a well-known port
use listen to place in passive mode
while (1)
{
    accept a client connection
    fork
    if (child)
    {
        communicate with new socket
        close new socket
        exit
     }
 else
 {close new socket}
}
```

- Single program has master and slave code.
- It is possible for slave to use execve.

## Concurrency Using a Single Process

- The following is a sample of pseudo codes for concurrency using a single process.

```
create a socket
bind to a well-known port
while (1)
{
    use select to wait for I/O
    if(original socket is ready)
    {
        accept() a new connection and add to read list
    }
 else if (a socket is ready for read)
 {
        read data from a client
        if(data completes a request)
        {
           do the request
           if(reply needed) add socket to write list
        }
   }
 else if (a socket is ready for write)
 {
   write data to a client
   if(message is complete)
   {
       remove socket from write list
   }
```

```
         else
         {
            adjust write parameters and leave in write list
         }
         }
       }
```

**When to Use the Various Server Types**

- Iterative vs Concurrent.
    1. Iterative server is simpler to write.
    2. Concurrent server is faster.
    3. Use iterative if it is fast enough.
- Real vs Apparent Concurrency.
    1. Writing a single process concurrent server is harder.
    2. Use a single process if data must be shared between clients.
    3. Use multiple processes if each slave is isolated or if you have multiple CPUs.
- Connection-Oriented vs Connectionless.
    1. Use connectionless if the protocol handles reliability.
    2. Use connectionless on a LAN with no errors.

**Avoiding Server Deadlock**

- Client connects but sends no request.  Server blocks in read call.
- Client sends request, but reads no replies.  Server blocks in write call.
- Concurrent servers with slaves are robust.

*Continue on next Module…*

**Further reading and digging:**

1. Check the best selling C / C++, Networking, Linux and Open Source books at Amazon.com.
2. GCC, GDB and other related tools.

| Winsock & .NET | Winsock | < TCP/IP Socket Programming Interfaces (APIs) | Linux Socket Index | More Socket APIs & C Code Snippets > |