MODULE 40
NETWORK PROGRAMMING
SOCKET PART II

My Training Period: hours

Note:
This is a continuation from Part I, Module 39. Working program examples compiled using gcc, tested using the public IPs, run on Fedora 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.

Abilities

- 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.
- Broadcast on the network to ask for a server.
- Example 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.

Example Clients – Some variations

A Simple Client Library

- To make a connection, a client must:

    ▪ select UDP or TCP...
    ▪ determine a server's IP address...
    ▪ determine the proper port...
    ▪ make the socket call...
    ▪ 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 *transport)
{
    struct hostent      *phe;   /* pointer to host information entry    */
    struct servent      *pse;   /* pointer to service information entry */
    struct protoent     *ppe;   /* pointer to protocol information entry*/
    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, strerror(errno));
        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.
    - Iterative server is simpler to write.
    - Concurrent server is faster.
    - Use iterative if it is fast enough.
- Real vs Apparent Concurrency.
    - Writing a single process concurrent server is harder.
    - Use a single process if data must be shared between clients.

- Use multiple processes if each slave is isolated or if you have multiple CPUs.

- Connection-Oriented vs Connectionless.

    - Use connectionless if the protocol handles reliability.
    - 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.

Iterative, Connectionless Servers (UDP)

Creating a Passive UDP Socket

- The following is a sample codes for a passive UDP socket.

```
int passiveUDP(const char *service)
{
    return passivesock(service, "udp", 0);
}

u_short portbase = 0;

int passivesock(const char *service, const char *transport, int qlen)
{
    struct servent  *pse;
    struct protoent *ppe;
    struct sockaddr_in sin;
    int     s, type;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;

    /* Map service name to port number */
    if(pse = getservbyname(service, transport))
        sin.sin_port = htons(ntohs((u_short)pse->s_port) + portbase);
    else if((sin.sin_port = htons((u_short)atoi(service))) == 0)
        errexit("can't get \"%s\" service entry\n", service);

    /* Map 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));

    /* Bind the socket */
    if(bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        errexit("can't bind to %s port: %s\n", service, strerror(errno));
    if(type == SOCK_STREAM && listen(s, qlen) < 0)
        errexit("can't listen on %s port: %s\n", service, strerror(errno));
    return s;
}
```

A TIME Server

- The following is a sample codes for Time server.

```
/* main() - Iterative UDP server for TIME service */
int main(int argc, char *argv[])
```

```
{
    struct sockaddr_in fsin;
    char *service = "time";
    char buf[1];
    int sock;
    time_t now;
    int alen;

    sock = passiveUDP(service);

 while (1)
{
     alen = sizeof(fsin);
     if(recvfrom(sock, buf, sizeof(buf), 0, (struct sockaddr *)&fsin, &alen) < 0)
        errexit("recvfrom: %s\n", strerror(errno));
     time(&now);
     now = htonl((u_long)now);
     sendto(sock, (char *)&now, sizeof(now), 0, (struct sockaddr *)&fsin, sizeof(fsin));
 }
}
```

Iterative, Connection-Oriented Servers (TCP)

A DAYTIME Server

- The following is a sample codes for Daytime server.

```
int passiveTCP(const char *service, int qlen)
{
    return passivesock(service, "tcp", qlen);
}

int main(int argc, char *argv[])
{
    struct sockaddr_in fsin;
    char *service = "daytime";
    int msock, ssock;
    int alen;

    msock = passiveTCP(service, 5);

    while (1) {
        ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
        if(ssock < 0)
            errexit("accept failed: %s\n", strerror(errno));
        TCPdaytimed(ssock);
        close(ssock);
  }
}

void TCPdaytimed(int fd)
{
    char *pts;
    time_t now;
    char *ctime();

    time(&now);
    pts = ctime(&now);
    write(fd, pts, strlen(pts));
    return;
}
```

- Close call requests a graceful shutdown.
- Data in transit is reliably delivered.
- Close requires messages and time.
- If the server closes you may be safe.
- If the client must close, the client may not cooperate.
- In our simple server, a client can make rapid calls and use resources associated with TCP shutdown timeout.

Concurrent, Connection-Oriented Servers (TCP)

The Value of Concurrency

- An iterative server may block for excessive time periods.
- An example is an echo server. A client could send many megabytes blocking other clients for substantial periods.
- A concurrent echo server could handle multiple clients simultaneously. Abusive clients would not affect polite clients as much.

A Concurrent Echo Server Using `fork()`

- The following is a sample codes for concurrent Echo server using `fork()`.

```
int main(int argc, char *argv[])
{
    char   *service = "echo";   /* service name or port number*/
    struct sockaddr_in fsin;    /* the address of a client    */
    int alen;                   /* length of client's address */
    int msock;                  /* master server socket       */
    int ssock;                  /* slave server socket        */

    msock = passiveTCP(service, QLEN);

    signal(SIGCHLD, reaper);

 while (1)
 {
     alen = sizeof(fsin);
     ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
     if(ssock < 0) {
        if(errno == EINTR)
            continue;
       errexit("accept: %s\n", strerror(errno));
  }
  switch (fork())
 {
     /* child */
     case 0:
       close(msock);
     exit(TCPechod(ssock));
     /* parent */
     default:
       close(ssock);
       break;
     case -1:
        errexit("fork: %s\n", strerror(errno));
   }
   }
}

int TCPechod(int fd)
{
    char  buf[BUFSIZ];
    int   cc;

 while (cc = read(fd, buf, sizeof buf))
 {
        if(cc < 0)
            errexit("echo read: %s\n", strerror(errno));
        if(write(fd, buf, cc) < 0)
            errexit("echo write: %s\n", strerror(errno));
  }
    return 0;
}

void reaper(int sig)
{
    int status;

    while (wait3(&status, WNOHANG, (struct rusage *)0) >= 0)
    /* empty */;
}
```

Single-Process, Concurrent Servers (TCP)

Data-driven Processing

- Arrival of data triggers processing.
- A message is typically a request.
- Server replies and awaits additional requests.
- If processing time is small, the requests may be possible to handle sequentially.
- Timesharing would be necessary only when the processing load is too high for sequential processing.
- Timesharing with multiple slaves is easier.

Using Select for Data-driven Processing

- A process calls select to wait for one (or more) of a collection of open files (or sockets) to be ready for I/O.

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval
*timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

- `select()` returns the number of fd's ready for I/O.
- `FD_ISSET` is used to determine which fd's are ready.
- `select()` returns 0 if the timer expires.
- `select()` returns -1 if there is an error.

An ECHO Server using a Single Process

- The following is a sample codes for Echo server using a single process.

```
int main(int argc, char *argv[])
{
    char   *service = "echo";
    struct sockaddr_in fsin;
    int    msock;
    fd_set rfds;
    fd_set afds;
    int    alen;
    int    fd, nfds;
    msock = passiveTCP(service, QLEN);

    nfds = getdtablesize();
    FD_ZERO(&afds);
    FD_SET(msock, &afds);

    while (1) {
        memcpy(&rfds, &afds, sizeof(rfds));

      if(select(nfds, &rfds, (fd_set *)0, (fd_set *)0, (struct timeval *)0) < 0)
            errexit("select: %s\n", strerror(errno));
        if(FD_ISSET(msock, &rfds))
 {
    int ssock;

    alen = sizeof(fsin);
    ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
    if(ssock < 0)
       errexit("accept: %s\n", strerror(errno));
     FD_SET(ssock, &afds);
   }
   for(fd=0; fd < nfds; ++fd)
     if(fd != msock && FD_ISSET(fd, &rfds))
        if(echo(fd) == 0)
          {
              (void) close(fd);
              FD_CLR(fd, &afds);
           }
    }
}

int echo(int fd)
{
    char buf[BUFSIZ];
    int  cc;

    cc = read(fd, buf, sizeof buf);
    if(cc < 0)
        errexit("echo read: %s\n", strerror(errno));
    if(cc && write(fd, buf, cc) < 0)
        errexit("echo write: %s\n", strerror(errno));
    return cc;
}
```

Multiprotocol Servers

Why use multiple protocols in a server?

- Using separate UDP and TCP servers gives the system administrator more flexibility.
- Using separate servers result in 2 moderately simple servers.
- Using one server eliminates duplicate code simplifying software maintenance.
- Using one server reduces the number of active processes.

A Multiprotocol DAYTIME Server

- The following is a sample codes for Multiprotocol Daytime server.

```
int main(int argc, char *argv[])
{
    char   *service = "daytime"; /* service name or port number */
    char   buf[LINELEN+1];       /* buffer for one line of text */
    struct sockaddr_in fsin;     /* the request from address    */
    int    alen;                 /* from-address length         */
    int    tsock;                /* TCP master socket           */
    int    usock;                /* UDP socket                  */
    int    nfds;
    fd_set rfds;                 /* readable file descriptors   */

    tsock = passiveTCP(service, QLEN);
    usock = passiveUDP(service);
    /* bit number of max fd */
    nfds = MAX(tsock, usock) + 1;

    FD_ZERO(&rfds);

    while (1) {
        FD_SET(tsock, &rfds);
        FD_SET(usock, &rfds);

        if(select(nfds, &rfds, (fd_set *)0, (fd_set *)0, (struct timeval *)0) < 0)
            errexit("select error: %s\n", strerror(errno));
        if(FD_ISSET(tsock, &rfds))
       {
            /* TCP slave socket */
            int ssock;
            alen = sizeof(fsin);
            ssock = accept(tsock, (struct sockaddr *)&fsin, &alen);
            if(ssock < 0)
                errexit("accept failed: %s\n", strerror(errno));
            daytime(buf);
            (void) write(ssock, buf, strlen(buf));
            (void) close(ssock);
        }
  if(FD_ISSET(usock, &rfds))
 {
   alen = sizeof(fsin);
   if(recvfrom(usock, buf, sizeof(buf), 0, (struct sockaddr *)&fsin, &alen) < 0)
      errexit("recvfrom: %s\n", strerror(errno));
   daytime(buf);
   (void) sendto(usock, buf, strlen(buf), 0, (struct sockaddr *)&fsin, sizeof(fsin));
   }
  }
}

int daytime(char buf[])
{
    char    *ctime();
    time_t now;
```

```
        (void) time(&now);
        sprintf(buf, "%s", ctime(&now));
    }
```

Multiservice Servers

Why combine services into one server?

- Fewer processes.
- Less memory.
- Less code duplication.
- Server complexity is really a result of accepting connections and handling concurrency.
- Having one server means the complex code does not need to be replicated.

Iterative Connectionless Server Design

- Server opens multiple UDP sockets each bound to a different port.
- Server keeps an array of function pointers to associate each socket with a service functions.
- Server uses select to determine which socket (port) to service next and calls the proper service function.

Iterative Connection-Oriented Server Design

- Server opens multiple passive TCP sockets each bound to a different port.
- Server keeps an array of function pointers to associate each socket with a service functions.
- Server uses select to determine which socket (port) to service next.
- When a connection is ready, server calls accept to start handling a connection.
- Server calls the proper service function.

Concurrent Connection-Oriented Server Design

- Master uses select to wait for connections over a set of passive TCP sockets.
- Master forks after accept.
- Slave handles communication with the client.

Single-Process Server Design

- Master uses select to wait for connections over a set of passive TCP sockets.
- After each accepts the new socket is added to the `fd_set`(s) as needed to handle client communication.
- Complex if the client protocols are not trivial.

Invoking Separate Programs from a Server

- Master uses `select()` to wait for connections over a set of passive TCP sockets.
- Master forks after accept.
- Child process uses execve to start a slave program to handle client communication.
- Different protocols are separated making it simpler to maintain.
- Changes to a slave program can be implemented without restarting the master.

Multiservice, Multiprotocol Servers

- Master uses select to wait for connections over a set of passive TCP sockets.
- In addition the `fd_set` includes a set of UDP sockets awaiting client messages.
- If a UDP message arrives, the master calls a handler function which formulates and issues a reply.
- If a TCP connection is needed the master calls accept.
- For simpler TCP connections, the master can handle read and write requests iteratively.
- The master can also use select.
- Lastly the master can use fork and let the child handle the connection.

Code Example of Super Server

- The following is a sample codes for super server.

```
struct service {
    char  *sv_name;
    char  sv_useTCP;
    int   sv_sock;
    int   (*sv_func)(int);
};

struct service svent[] = {
    { "echo", TCP_SERV, NOSOCK, TCPechod },
    { "chargen", TCP_SERV, NOSOCK, TCPchargend },
    { "daytime", TCP_SERV, NOSOCK, TCPdaytimed },
    { "time", TCP_SERV, NOSOCK, TCPtimed },
    { 0, 0, 0, 0 },
};

int main(int argc, char *argv[])
{
    struct service  *psv,    /* service table pointer */
            *fd2sv[NOFILE];  /* map fd to service pointer */
    int     fd, nfds;
    fd_set  afds, rfds;      /* readable file descriptors */

    nfds = 0;
    FD_ZERO(&afds);
    for(psv = &svent[0]; psv->sv_name; ++psv)
 {
        if(psv->sv_useTCP)
            psv->sv_sock = passiveTCP(psv->sv_name, QLEN);
        else
            psv->sv_sock = passiveUDP(psv->sv_name);
        fd2sv[psv->sv_sock] = psv;
        nfds = MAX(psv->sv_sock+1, nfds);
        FD_SET(psv->sv_sock, &afds);
  }

  (void) signal(SIGCHLD, reaper);

    while (1) {
      memcpy(&rfds, &afds, sizeof(rfds));
      if(select(nfds, &rfds, (fd_set *)0, (fd_set *)0, (struct timeval *)0) < 0)
       {
         if(errno == EINTR)
           continue;
         errexit("select error: %s\n", strerror(errno));
       }
        for(fd=0; fd<nfds; ++fd)
        {
            if(FD_ISSET(fd, &rfds))
            {
                psv = fd2sv[fd];
                if(psv->sv_useTCP)
                   doTCP(psv);
                else
                   psv->sv_func(psv->sv_sock);
            }
          }
    }
}

/* doTCP() - handle a TCP service connection request */
void doTCP(struct service *psv)
{
    /* the request from address */
    struct  sockaddr_in fsin;
    /* from-address length */
    int alen;
    int fd, ssock;

    alen = sizeof(fsin);
    ssock = accept(psv->sv_sock, (struct sockaddr *)&fsin, &alen);
    if(ssock < 0)
        errexit("accept: %s\n", strerror(errno));
    switch (fork())
   {
    case 0:
        break;
    case -1:
        errexit("fork: %s\n", strerror(errno));
    default:
        (void) close(ssock);
        /* parent */
        return;
    }
    /* child */
    for(fd = NOFILE; fd >= 0; --fd)
        if(fd != ssock) (void) close(fd);
    exit(psv->sv_func(ssock));
}

/* reaper() - clean up zombie children */
void reaper(int sig)
{
    int status;
    while(wait3(&status, WNOHANG, (struct rusage *)0) >= 0)
    /* empty */;
}
```

Some Idea In Managing Server Concurrency

Concurrency vs Iteration

Making the decision

- Program design is vastly different.
- Programmer needs to decide early.
- Network and computer speeds change.
- Optimality is a moving target.
- Programmer must use insight based on experience to decide which is better.

Level of Concurrency

- Number of concurrent clients.
- Iterative means 1 client at a time.
- Unbounded concurrency allows flexibility.
- TCP software limits the number of connections.
- OS limits each process to a fixed number of open files.
- OS limits the number of processes.

Problems with Unbounded Concurrency

- OS can run out of resources such as memory, processes, sockets, buffers causing blocking, thrashing, crashing...
- Demand for one service can inhibit others e.g. web server may prevent other use.
- Over-use can limit performance e.g. ftp server could be so slow that clients cancel requests wasting time spent doing a partial transfer.

Cost of Concurrency

- Assuming a forking concurrent server, each connection requires time for a process creation (c).
- Each connection also requires some time for processing requests (p).
- Consider 2 requests arriving at the same time.
- Iterative server completes both at time 2p.
- Concurrent server completes both perhaps at time 2c+p.
- If $p < 2c$ the iterative server is faster.
- The situation can get worse with more requests.
- The number of active processes can exceed the CPU capacity.
- Servers with heavy loads generally try to dodge the process creation cost.

Process Pre-allocation to Limit Delays

- Master server process forks n times.
- The n slaves handle up to n clients.
- Operates like n iterative servers.

- Due to child processes inheriting the parent's passive socket, the slaves can all wait in accept on the same socket.
- For UDP, the slaves can all call recvfrom on the same socket.
- To avoid problems like memory leaks, the slaves can be periodically replaced.
- For UDP, bursts can overflow buffers causing data loss. Pre-allocation can limit this problem.

Dynamic Pre-allocation

- Pre-allocation can cause extra processing time if many slaves are all waiting on the same socket.
- If the server is busy, it can be better to have many slaves pre-allocated.
- If the server is idle, it can be better to have very few slaves pre-allocated.
- Some servers (Apache) adjust the level of concurrency according to service demand.

Delayed Allocation

- Rather than immediately forking, the master can quickly examine a request.
- It may be faster for some requests to handle them in the master rather than forking.
- Longer requests may be more appropriate to handle in a child process.
- If it is hard to quickly estimate processing time, the server can set a timer to expire after a small time and then fork to let a child finish the request.

Client Concurrency

- Shorter Response Time.
- Increased Throughput.
- Concurrency Allows Better Control.
- Communicating with Multiple Servers.
- Achieving Concurrency with a Single Client Process.

-----------------------Program Examples---------------------

DNS

- DNS stands for "Domain Name System" (for Windows implementation it is called Domain Name Service). For socket it has three major components:

    - Domain name space and resource records: Specifications for a tree-structured name space and the data associated with the names.
    - Name servers: Server programs that hold information about the domain tree structure and that set information.
    - Resolvers: Programs that extract information from name servers in response to client requests.

- DNS used to translate the IP address to domain name and vice versa. This way, when someone enters:

```
telnet serv.google.com
```

- `telnet` can find out that it needs to `connect()` to let say, "198.137.240.92". To get these information we can use `gethostbyname()`:

```
#include <netdb.h>

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

- As you see, it returns a pointer to a `struct hostent`, and `struct hostent` is shown below:

```
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]
```

- And the descriptions:

Member	Description
h_name	Official name of the host.
h_aliases	A NULL-terminated array of alternate names for the host.
h_addrtype	The type of address being returned; usually AF_INET.
h_length	The length of the address in bytes.
h_addr_list	A zero-terminated array of network addresses for the host. Host addresses are in Network Byte Order.
h_addr	The first address in h_addr_list.

Table 40.1

- gethostbyname() returns a pointer to the filled struct hostent, or NULL on error but errno is not set, h_errno is set instead.
- As said before in implementation we use Domain Name Service in Windows and BIND in Unix/Linux. Here, we configure the Forward Lookup Zone for name to IP resolution and Reverse Lookup Zone for the reverse.
- The following is a program example using the gethostname().

```
/*****getipaddr.c ******/
/****a hostname lookup program example******/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
struct hostent *h;

/*error check the command line*/
if(argc != 2)
{
fprintf(stderr, "Usage: %s <domain_name>\n", argv[0]);
exit(1);
}

/*get the host info*/
if((h=gethostbyname(argv[1])) == NULL)
{
herror("gethostbyname(): ");
exit(1);
}
else
printf("gethostbyname() is OK.\n");

printf("The host name is: %s\n", h->h_name);
printf("The IP Address is: %s\n", inet_ntoa(*((struct in_addr *)h->h_addr)));
printf("The address length is: %d\n", h->h_length);

printf("Sniffing other names...sniff...sniff...sniff...\n");
int j = 0;
do
{
printf("An alias #%d is: %s\n", j, h->h_aliases[j]);
j++;
}
while(h->h_aliases[j] != NULL);

printf("Sniffing other IPs...sniff....sniff...sniff...\n");
int i = 0;
do
{
printf("Address #%i is: %s\n", i, inet_ntoa(*((struct in_addr *)(h->h_addr_list[i]))));
i++;
}
while(h->h_addr_list[i] != NULL);
return 0;
}
```

- Compile and link the program.

```
[bodo@bakawali testsocket]$ gcc -g getipaddr.c -o getipaddr
```

- Run the program. Because of the server used in this testing is using public IP address, we can test it querying the public domain such as `www.yahoo.com` **:o)**.

```
[bodo@bakawali testsocket]$ ./getipaddr www.yahoo.com
The host name is: www.yahoo.akadns.net
The IP Address is: 66.94.230.50
The address length is: 4
Sniffing other names...sniff...sniff...sniff...
An alias #0 is: www.yahoo.com
Sniffing other IPs...sniff....sniff...sniff...
Address #0 is: 66.94.230.50
Address #1 is: 66.94.230.36
Address #2 is: 66.94.230.41
Address #3 is: 66.94.230.34
Address #4 is: 66.94.230.47
Address #5 is: 66.94.230.32
Address #6 is: 66.94.230.35
Address #7 is: 66.94.230.45
```

- Again, running the program testing another domain.

```
[bodo@bakawali testsocket]$ ./getipaddr www.google.com
The host name is: www.l.google.com
The IP Address is: 66.102.7.104
The address length is: 4
Sniffing other names...sniff...sniff...sniff...
An alias #0 is: www.google.com
Sniffing other IPs...sniff....sniff...sniff...
Address #0 is: 66.102.7.104
Address #1 is: 66.102.7.99
Address #2 is: 66.102.7.147
[bodo@bakawali testsocket]$
```

- With `gethostbyname()`, you can't use `perror()` to print error message since `errno` is not used instead, call `herror()`.
- You simply pass the string that contains the machine name ("`www.google.com`") to `gethostbyname()`, and then grab the information out of the returned `struct hostent`.
- The only possible weirdness might be in the printing of the IP address. Here, `h->h_addr` is a `char*`, but `inet_ntoa()` wants a `struct in_addr` passed to it. So we need to cast `h->h_addr` to a `struct in_addr*`, then dereference it to get the data.

Some Client-Server Background

- Just about everything on the network deals with client processes talking to server processes and vice-versa. For example take a `telnet`.
- When you telnet to a remote host on port 23 at client, a program on that server normally called `telnetd` (telnet daemon), will respond. It handles the incoming telnet connection, sets you up with a login prompt, etc. In Windows this daemon normally called a service. The daemon or service must be running in order to do the communication.
- Note that the client-server pair can communicate using `SOCK_STREAM`, `SOCK_DGRAM`, or anything else (as long as they're using the same protocol). Some good examples of client-server pairs are `telnet/telnetd`, `ftp/ftpd`, or `bootp/bootpd`. Every time you use `ftp`, there's a remote program, `ftpd` that will serve you.
- Often, there will only be one server, and that server will handle multiple clients using `fork()` etc. The basic routine is: server will wait for a connection, `accept()` it and `fork()` a child process to handle it. The following program example is what our sample server does.

A Simple Stream Server Program Example

- What this server does is send the string "`This is a test string from server!`" out over a stream connection.

- To test this server, run it in one window and telnet to it from another window or run it in a server and telnet to it from another machine with the following command.

```
telnet the_remote_hostname 3490
```

- Where `the_remote_hostname` is the name of the machine you're running it on. The following is the server source code:

```
/*serverprog.c - a stream socket server demo*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

/* the port users will be connecting to */
#define MYPORT 3490
/* how many pending connections queue will hold */
#define BACKLOG 10

void sigchld_handler(int s)
{
while(wait(NULL) > 0);
}

int main(int argc, char *argv[])
{
/*listen on sock_fd, new connection on new_fd*/
int sockfd, new_fd;
/*my address information*/
struct sockaddr_in my_addr;
/*connector's address information*/
struct sockaddr_in their_addr;
int sin_size;
struct sigaction sa;
int yes = 1;

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
perror("Server-socket() error lol!");
exit(1);
}
else
printf("Server-socket() sockfd is OK...\n");

if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
{
perror("Server-setsockopt() error lol!");
exit(1);
}
else
printf("Server-setsockopt is OK...\n");

/* host byte order*/
my_addr.sin_family = AF_INET;
/* short, network byte order*/
my_addr.sin_port = htons(MYPORT);
/* automatically fill with my IP*/
my_addr.sin_addr.s_addr = INADDR_ANY;

printf("Server-Using %s and port %d...\n", inet_ntoa(my_addr.sin_addr), MYPORT);

/* 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");
exit(1);
}
else
printf("Server-bind() is OK...\n");
if(listen(sockfd, BACKLOG) == -1)
{
perror("Server-listen() error");
exit(1);
}
printf("Server-listen() is OK...Listening...\n");

/*clean all the dead processes*/
sa.sa_handler = sigchld_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;

if(sigaction(SIGCHLD, &sa, NULL) == -1)
{
perror("Server-sigaction() error");
exit(1);
}
else
printf("Server-sigaction() is OK...\n");

/*accept() loop*/
while(1)
{
sin_size = sizeof(struct sockaddr_in);
if((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1)
{
perror("Server-accept() error");
continue;
}
else
printf("Server-accept() is OK...\n");
printf("Server-new socket, new_fd is OK...\n");
printf("Server: Got connection from %s\n", inet_ntoa(their_addr.sin_addr));

/* this is the child process */
if(!fork())
{
/* child doesn't need the listener */
close(sockfd);

if(send(new_fd, "This is a test string from server!\n", 37, 0) == -1)
perror("Server-send() error lol!");
close(new_fd);
exit(0);
}
else
printf("Server-send is OK...!\n");

/* parent doesn't need this*/
close(new_fd);
printf("Server-new socket, new_fd closed successfully...\n");
}
return 0;
}
```

- Compile and link the program.

```
[bodo@bakawali testsocket]$ gcc -g serverprog.c -o serverprog
```

- Run the program.

```
[bodo@bakawali testsocket]$ ./serverprog
Server-socket() sockfd is OK...
Server-setsockopt() is OK...
Server-Using 0.0.0.0 and port 3490...
Server-bind() is OK...
Server-listen() is OK...Listening...
Server-sigaction() is OK...

[1]+  Stopped                 ./serverprog

[bodo@bakawali testsocket]$
```

- Verify that the program is running in the background. You may do this from another terminal.

```
[bodo@bakawali testsocket]$ bg
[1]+ ./serverprog &
```

- Verify that the program/process is listening on the specified port, waiting for connection.

```
[bodo@bakawali testsocket]$ netstat -a | grep 3490
tcp        0      0 *:3490        *:*                   LISTEN
```

- Again, verify that the program/process is listening, waiting for connection.

```
[bodo@bakawali testsocket]$ ps aux | grep serverprog
bodo      2586  0.0  0.2  2940  296 pts/3    S    14:04   0:00 ./serverprog
bodo      2590  0.0  0.5  5432  660 pts/3    R+   14:04   0:00 grep serverprog
```

- Then, trying the telnet. Open another terminal, telnet itself with the specified port number. Here we use the server name, `bakawali`. When the string is displayed press the Escape character `Ctrl+]` (^]). Then we have a real telnet session.

```
[bodo@bakawali testsocket]$ telnet bakawali 3490
Trying 203.106.93.94...
Connected to bakawali.jmti.gov.my (203.106.93.94).
Escape character is '^]'.
This is the test string from server!
^]
telnet> ?
Commands may be abbreviated.  Commands are:

close           close current connection
logout          forcibly logout remote user and close the connection
display         display operating parameters
mode            try to enter line or character mode ('mode ?' for more)
open            connect to a site
quit            exit telnet
send            transmit special characters ('send ?' for more)
set             set operating parameters ('set ?' for more)
unset           unset operating parameters ('unset ?' for more)
status          print status information
toggle          toggle operating parameters ('toggle ?' for more)
slc             change state of special charaters ('slc ?' for more)
auth            turn on (off) authentication ('auth ?' for more)
encrypt         turn on (off) encryption ('encrypt ?' for more)
forward         turn on (off) credential forwarding ('forward ?' for more)
z               suspend telnet
!               invoke a subshell
environ         change environment variables ('environ ?' for more)
?               print help information
telnet>
```

- Type `quit` to exit the session.

```
...
telnet> quit
Connection closed.
[bodo@bakawali ~]$
```

- If we do not stop the server program/process (Ctrl+Z), at the server terminal the following messages should be displayed. Press `Enter` (Carriage Return) key back to the prompt.

```
[bodo@bakawali testsocket]$ ./serverprog
Server-socket() sockfd is OK...
Server-setsockopt() is OK...
Server-Using 0.0.0.0 and port 3490...
Server-bind() is OK...
Server-listen() is OK...Listening...
Server-sigaction() is OK...
Server-accept() is OK...
Server-new socket, new_fd is OK...
Server: Got connection from 203.106.93.94
Server-send() is OK...!
Server-new socket, new_fd closed successfully...
```

- To stop the process just issue a normal `kill` command. Before that verify again.

```
[bodo@bakawali testsocket]$ netstat -a | grep 3490
tcp        0      0 *:3490        *:*                  LISTEN

[bodo@bakawali testsocket]$ ps aux | grep ./serverprog
bodo  3184  0.0  0.2  1384  324 pts/3  S   23:46   0:00 ./serverprog
bodo  3188  0.0  0.5  3720  652 pts/3  R+  23:48   0:00 grep ./serverprog

[bodo@bakawali testsocket]$ kill -9 3184
[bodo@bakawali testsocket]$ netstat -a | grep 3490
[1]+  Killed                  ./serverprog

[bodo@bakawali testsocket]$
```

- The server program seems OK. Next section is a client program, `clientprog.c` that we will use to test our server program, `serverprog.c`.
- The `sigaction()` code is responsible for cleaning the zombie processes that appear as the `fork()`ed child processes. You will get the message from this server by using the client program example presented in the next section.

A Simple Stream Client Program Example

- This client will connect to the host that you specify in the command line, with port 3490. It will get the string that the previous server sends. The following is the source code.

```
/*** clientprog.c ****/
/*** a stream socket client demo ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

//the port client will be connecting to
#define PORT 3490
//max number of bytes we can get at once
#define MAXDATASIZE 300

int main(int argc, char *argv[])
{
int sockfd, numbytes;
char buf[MAXDATASIZE];
struct hostent *he;
//connector's address information
struct sockaddr_in their_addr;

//if no command line argument supplied
if(argc != 2)
{
fprintf(stderr, "Client-Usage: %s the_client_hostname\n", argv[0]);
//just exit
exit(1);
}

//get the host info
if((he=gethostbyname(argv[1])) == NULL)
{
perror("gethostbyname()");
exit(1);
}
else
printf("Client-The remote host is: %s\n", argv[1]);

if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
perror("socket()");
exit(1);
}
else
printf("Client-The socket() sockfd is OK...\n");
//host byte order
their_addr.sin_family = AF_INET;
//short, network byte order
printf("Server-Using %s and port %d...\n", argv[1], PORT);
their_addr.sin_port = htons(PORT);
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
//zero the rest of the struct
memset(&(their_addr.sin_zero), '\0', 8);

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

if((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1)
{
perror("recv()");
exit(1);
}
else
printf("Client-The recv() is OK...\n");

buf[numbytes] = '\0';
printf("Client-Received: %s", buf);

printf("Client-Closing sockfd\n");
close(sockfd);
return 0;
}
```

- Compile and link the program.

```
[bodo@bakawali testsocket]$ gcc -g clientprog.c -o clientprog
```

- Run the program without argument.

```
[bodo@bakawali testsocket]$ ./clientprog
Client-Usage: ./clientprog the_client_hostname
```

- Run the program with server IP address or name as an argument. Here we use IP address.
- Make sure your previous `serverprog` program is running. We will connect using the same server. You can try running the server and client program at different machines.

```
[bodo@bakawali testsocket]$ ./clientprog 203.106.93.94
…
[bodo@bakawali testsocket]$ ./clientprog bakawali
Client-The remote host is: bakawali
Client-The socket() sockfd is OK...
Server-Using bakawali and port 3490...
Client-The connect() is OK...
Client-The recv() is OK...
Client-Received: This is the test string from server!
Client-Closing sockfd
```

- Verify the connection.

```
[bodo@bakawali testsocket]$ netstat -a | grep 3490
tcp        0      0 *:3490            *:*                   LISTEN
tcp        0      0 bakawali.jmti.gov.my:3490   bakawali.jmti.gov.my:1358
TIME_WAIT
[bodo@bakawali testsocket]$
```

- At server’s console, we have the following messages.

```
[bodo@bakawali testsocket]$ ./serverprog
Server-socket() sockfd is OK...
Server-setsockopt() is OK...
Server-Using 0.0.0.0 and port 3490...
```

```
Server-bind() is OK...
Server-listen() is OK...Listening...
Server-sigaction() is OK...
Server-accept() is OK...
Server-new socket, new_fd is OK...
Server: Got connection from 203.106.93.94
Server-send() is OK...!
Server-new socket, new_fd closed successfully...
```

- Well, our server and client programs work! Here we run the server program and let it listens for connection. Then we run the client program. They got connected!
- Notice that if you don't run the server before you run the client, `connect()` returns "`Connection refused`" message as shown below.

```
[bodo@bakawali testsocket]$ ./clientprog bakawali
Client-The remote host is: bakawali
Client-The socket() sockfd is OK...
Server-Using bakawali and port 3490...
connect: Connection refused
```

Datagram Sockets: The Connectionless

- The following program examples use the UDP, the connectionless datagram. The `senderprog.c` (client) is sending a message to `receiverprog.c` (server) that acts as listener.

```
/*receiverprog.c - a server, datagram sockets*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
/* the port users will be connecting to */
#define MYPORT 4950
#define MAXBUFLEN 500

int main(int argc, char *argv[])
{
int sockfd;
/* my address information */
struct sockaddr_in my_addr;
/* connector's address information */
struct sockaddr_in their_addr;
int addr_len, numbytes;
char buf[MAXBUFLEN];

if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
perror("Server-socket() sockfd 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);
/* automatically fill with my IP */
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");

addr_len = sizeof(struct sockaddr);
if((numbytes = recvfrom(sockfd, buf, MAXBUFLEN-1, 0, (struct sockaddr *)&their_addr,
&addr_len)) == -1)
{
perror("Server-recvfrom() error lol!");
/*If something wrong, just exit lol...*/
exit(1);
}
else
{
printf("Server-Waiting and listening...\n");
printf("Server-recvfrom() is OK...\n");
}

printf("Server-Got packet from %s\n", inet_ntoa(their_addr.sin_addr));
printf("Server-Packet is %d bytes long\n", numbytes);
buf[numbytes] = '\0';
printf("Server-Packet contains \"%s\"\n", buf);

if(close(sockfd) != 0)
printf("Server-sockfd closing failed!\n");
else
printf("Server-sockfd successfully closed!\n");
return 0;
}
```

- Compile and link the program.

```
[bodo@bakawali testsocket]$ gcc -g receiverprog.c -o receiverprog
```

- Run the program, and then verify that it is running in background, start listening, waiting for connection.

```
[bodo@bakawali testsocket]$ ./receiverprog
Server-socket() sockfd is OK...
Server-bind() is OK...

[1]+  Stopped                 ./receiverprog
[bodo@bakawali testsocket]$ bg
[1]+ ./receiverprog &
[bodo@bakawali testsocket]$ netstat -a | grep 4950
udp        0      0 *:4950                  *:*
[bodo@bakawali testsocket]$
```

- This is UDP server, trying telnet to this server will fail because telnet uses TCP instead of UDP.

```
[bodo@bakawali testsocket]$ telnet 203.106.93.94 4950
Trying 203.106.93.94...
telnet: connect to address 203.106.93.94: Connection refused
telnet: Unable to connect to remote host: Connection refused
[bodo@bakawali testsocket]$
```

- Notice that in our call to `socket()` we're using `SOCK_DGRAM`. Also, note that there's no need to `listen()` or `accept()`. The following is the source code for `senderprog.c` (the client).

```
/*senderprog.c - a client, datagram*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
/* the port users will be connecting to */
#define MYPORT 4950

int main(int argc, char *argv[])
{
int sockfd;
/* connector's address information */
struct sockaddr_in their_addr;
struct hostent *he;
int numbytes;

if (argc != 3)
{
fprintf(stderr, "Client-Usage: %s <hostname> <message>\n", argv[0]);
exit(1);
}
/* get the host info */
if ((he = gethostbyname(argv[1])) == NULL)
{
perror("Client-gethostbyname() error lol!");
exit(1);
}
else
printf("Client-gethostname() is OK...\n");

if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
perror("Client-socket() error lol!");
exit(1);
}
else
printf("Client-socket() sockfd is OK...\n");

/* host byte order */
their_addr.sin_family = AF_INET;
/* short, network byte order */
printf("Using port: 4950\n");
their_addr.sin_port = htons(MYPORT);
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
/* zero the rest of the struct */
memset(&(their_addr.sin_zero), '\0', 8);

if((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0, (struct sockaddr *)&their_addr,
sizeof(struct sockaddr))) == -1)
{
perror("Client-sendto() error lol!");
exit(1);
}
else
printf("Client-sendto() is OK...\n");

printf("sent %d bytes to %s\n", numbytes, inet_ntoa(their_addr.sin_addr));

if (close(sockfd) != 0)
printf("Client-sockfd closing is failed!\n");
else
printf("Client-sockfd successfully closed!\n");
return 0;
}
```

- Compile and link the program.

```
[bodo@bakawali testsocket]$ gcc -g senderprog.c -o senderprog
```

- Run the program without arguments.

```
[bodo@bakawali testsocket]$ ./senderprog
Client-Usage: ./senderprog <hostname> <message>
```

- Run the program with arguments.

```
[bodo@bakawali testsocket]$ ./senderprog 203.106.93.94 "Testing UDP datagram message from client"
Client-gethostname() is OK...
Client-socket() sockfd is OK...
Using port: 4950
Server-Waiting and listening...
Server-recvfrom() is OK...
Server-Got packet from 203.106.93.94
Server-Packet is 42 bytes long
Server-Packet contains "Testing UDP datagram message from client"
Server-sockfd successfully closed!
Client-sendto() is OK...
```

```
sent 42 bytes to 203.106.93.94
Client-sockfd successfully closed!
[1]+  Done                    ./receiverprog
[bodo@bakawali testsocket]$
```

- Here, we test the UDP server and the client using the same machine. Make sure there is no restriction such as permission etc. for the user that run the programs.
- To make it really real, may be you can test these programs by running `receiverprog` on some machine, and then run `senderprog` on another. If there is no error, they should communicate.
- If `senderprog` calls `connect()` and specifies the `receiverprog`'s address then the `senderprog` may only sent to and receive from the address specified by `connect()`.
- For this reason, you don't have to use `sendto()` and `recvfrom()`; you can simply use `send()` and `recv()`.

Blocking

- In a simple word 'block' means sleep but in a standby mode. You probably noticed that when you run `receiverprog`, previously, it just sits there until a packet arrives.
- What happened is that it called `recvfrom()`, there was no data, and so `recvfrom()` is said to "block" (that is, sleep there) until some data arrives. The socket functions that can block are:

```
accept()
read()
readv()
recv()
recvfrom()
recvmsg()
send()
sendmsg()
sendto()
write()
writev()
```

- The reason they can do this is because they're allowed to. When you first create the socket descriptor with `socket()`, the kernel sets it to blocking.
- If you don't want a socket to be blocking, you have to make a call to `fcntl()` something like the following:

```
#include <unistd.h>
#include <fcntl.h>
...
...
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
...
...
```

- By setting a socket to non-blocking, you can effectively 'poll' the socket for information. If you try to read from a non-blocking socket and there's no data there, it's not allowed to block, it will return `-1` and `errno` will be set to `EWOULDBLOCK`.
- Generally speaking, however, this type of polling is a bad idea. If you put your program in a busy-wait looking for data on the socket, you'll suck up CPU time.
- A more elegant solution for checking to see if there's data waiting to be read comes in the following section on `select()`.

Using `select()` for I/O multiplexing

- One traditional way to write network servers is to have the main server block on `accept()`, waiting for a connection. Once a connection comes in, the server `fork()`s, then the child process handles the connection and the main server is able to service new incoming requests.
- With `select()`, instead of having a process for each request, there is usually only one process that multiplexes all requests, servicing each request as much as it can.
- So one main advantage of using `select()` is that your server will only require a single process to handle all requests. Thus, your server will not need shared memory or synchronization primitives for different tasks to communicate.

- As discussed before we can use the non-blocking sockets' functions but it is CPU intensive.
- One major disadvantage of using `select()`, is that your server cannot act like there's only one client, like with a `fork()`'ing solution. For example, with a `fork()`'ing solution, after the server `fork()`s, the child process works with the client as if there was only one client in the universe, the child does not have to worry about new incoming connections or the existence of other sockets.
- With `select()`, the programming isn't as transparent. The prototype is as the following:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout);
```

- The function monitors "sets" of file descriptors; in particular `readfds`, `writefds`, and `exceptfds`. If you want to see if you can read from standard input and some socket descriptor, `sockfd`, just add the file descriptors `0` and `sockfd` to the set `readfds`.
- The parameter `numfds` should be set to the values of the highest file descriptor plus one. In this example, it should be set to `sockfd+1`, since it is assuredly higher than standard input that is `0`.
- When `select()` returns, `readfds` will be modified to reflect which of the file descriptors you have selected which is ready for reading. You can test them with the macro `FD_ISSET()` listed below.
- Let see how to manipulate these sets. Each set is of the type `fd_set`. The following macros operate on this type:

  ▪ `FD_ZERO(fd_set *set)` – clears a file descriptor set.
  ▪ `FD_SET(int fd, fd_set *set)` – adds `fd` to the set.
  ▪ `FD_CLR(int fd, fd_set *set)` – removes `fd` from the set.
  ▪ `FD_ISSET(int fd, fd_set *set)` – tests to see if `fd` is in the set.

- `select()` works by blocking until something happens on a file descriptor/socket. The 'something' is the data coming in or being able to write to a file descriptor, you tell `select()` what you want to be woken up by. How do you tell it? You fill up an `fd_set` structure with some macros.
- Most `select()`based servers look quite similar:

  ▪ Fill up an `fd_set` structure with the file descriptors you want to know when data comes in on.
  ▪ Fill up an `fd_set` structure with the file descriptors you want to know when you can write on.
  ▪ Call `select()` and block until something happens.
  ▪ Once `select()` returns, check to see if any of your file descriptors was the reason you woke up. If so, 'service' that file descriptor in whatever particular way your server needs to (i.e. read in a request for a Web page).
  ▪ Repeat this process forever.

- Sometimes you don't want to wait forever for someone to send you some data. Maybe every 60 seconds you want to print something like "`Processing...`" to the terminal even though nothing has happened.
- The `timeval` structure allows you to specify a timeout period. If the time is exceeded and `select()` still hasn't found any ready file descriptors, it'll return, so you can continue processing.
- The `struct timeval` has the following fields:

```
struct timeval
{
int tv_sec; /* seconds */
int tv_usec; /* microseconds */
};
```

- Just set `tv_sec` to the number of seconds to wait, and set `tv_usec` to the number of microseconds to wait. There are 1,000,000 microseconds in a second. Also, when the function returns, `timeout` might be updated to show the time still remaining.
- Standard UNIX time slice is around 100 milliseconds, so you might have to wait that long no matter how small you set your `struct timeval`.
- If you set the fields in your `struct timeval` to `0`, `select()` will timeout immediately, effectively polling all the file descriptors in your sets. If you set the parameter `timeout` to NULL, it will never timeout, and will wait until the first file descriptor is ready.
- Finally, if you don't care about waiting for a certain set, you can just set it to NULL in the call to `select()`.
- The following code snippet waits 5.8 seconds for something to appear on standard input.

```
/*selectcp.c - a select() demo*/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
/* file descriptor for standard input */
#define STDIN 0

int main(int argc, char *argv[])
{
struct timeval tval;
fd_set readfds;
tval.tv_sec = 5;
tval.tv_usec = 800000;

FD_ZERO(&readfds);
FD_SET(STDIN, &readfds);
/* don't care about writefds and exceptfds: */
select(STDIN+1, &readfds, NULL, NULL, &tval);
if (FD_ISSET(STDIN, &readfds))
  printf("A key was pressed lor!\n");
else
  printf("Timed out lor!...\n");
return 0;
}
```

- Compile and link the program. Make sure there is no error **:o)**.

```
[bodo@bakawali testsocket]$ gcc -g selectcp.c -o selectcp
```

- Run the program and then press `k`.

```
[bodo@bakawali testsocket]$ ./selectcp
k
A key was pressed lor!
```

- Run the program and just leave it.

```
[bodo@bakawali testsocket]$ ./selectcp
Timed out lor!...
```

- If you're on a line buffered terminal, the key you hit should be `RETURN` or it will time out anyway.
- Now, some of you might think this is a great way to wait for data on a datagram socket and you are right: it might be. Some Unices can use `select()` in this manner, and some can't. You should see what your local man page says on the matter if you want to attempt it.
- Some Unices update the time in your `struct timeval` to reflect the amount of time still remaining before a timeout. But others do not. Don't rely on that occurring if you want to be portable. Use `gettimeofday()` if you need to track time elapsed.
- When a socket in the `read` set closes the connection, `select()` returns with that socket descriptor set as "ready to read". When you actually do `recv()` from it, `recv()` will return `0`. That's how you know the client has closed the connection.
- If you have a socket that is `listen()`ing, you can check to see if there is a new connection by putting that socket's file descriptor in the `readfds` set.

…Continue on next Module…More concept and program examples…

--End socket Part II--
www.tenouk.com

Further reading and digging:

1. Check the best selling C/C++, Networking, Linux and Open Source books at Amazon.com.