Introduction to User Datagram Protocol.

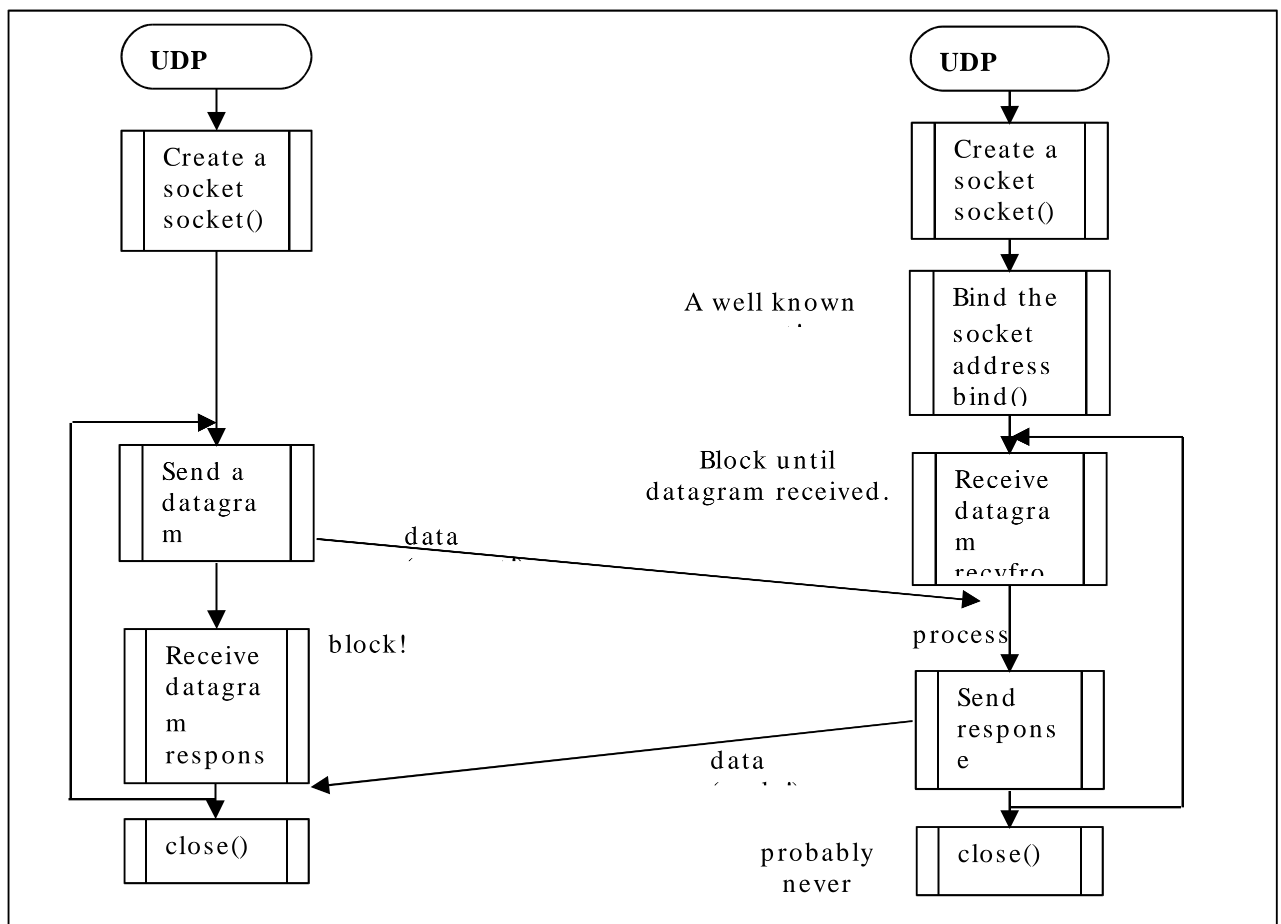
Aims.

The aim of this worksheet is to experiment with sockets using **U**ser **D**atagram **P**rotocol (UDP). It introduces the UDP communication functions **sendto()** and **recvfrom().**

User Datagram Protocol.

This protocol is often referred to as **U**nreliable **D**atagram **P**rotocol (UDP) because, unlike TCP streams, there is no guarantee that data sent will reach its destination. When using a TCP socket a connection is established between the end point before communication takes place and this connection is maintained until one end decides to actively close it. With a UDP socket a connection is NOT made, instead the sender just issues a message to its destination and hopes it gets there! The message uses a datagram of fixed length, often termed a record. Since there is no connection between client and server the client can send a datagram to one server and then immediately send a datagram to another server using the same socket! UDP is a *connectionless* protocol.

The client/ server relationship using UDP is shown by the following diagram.



Notice that we do not use the function *connect()*. It can be used but the nature of the processes change slightly and the functions *send()* and *recv()* are used instead of *sendto()* and *recvfrom()*. Use **man** pages to investigate these functions further.

Let us create a simple UDP client/ server pair where the client just sends a message which the server displays.

A Simple UDP Server.

Use **emacs** to create the server source code:

```
emacs listener.c &
```

Enter the following program:

```
/* listener.c - a datagram socket 'server'
 * simply displays message received then dies!
 */
#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>

#define MYPORT 4950        /* the port users connect to */
#define MAXBUFLEN 100

int main( void) {
  int sockfd;
  struct sockaddr_in my_addr;    /* info for my addr i.e. server */
  struct sockaddr_in their_addr; /* client's address info */
  int addr_len, numbytes;
  char buf[ MAXBUFLEN];

  if( (sockfd = socket( AF_INET, SOCK_DGRAM, 0)) == -1) {
    perror( "Listener socket");
    exit( 1);
  }

  memset( &my_addr, 0, sizeof( my_addr));     /* zero struct */
  my_addr.sin_family = AF_INET;               /* host byte order ... */
  my_addr.sin_port = htons( MYPORT); /* ... short, network byte order */
  my_addr.sin_addr.s_addr = INADDR_ANY;    /* any of server IP addrs */

  if( bind( sockfd, (struct sockaddr *)&my_addr,
                           sizeof( struct sockaddr)) == -1) {
    perror( "Listener bind");
    exit( 1);
  }

  addr_len = sizeof( struct sockaddr);
```

```
  if( (numbytes = recvfrom( sockfd, buf, MAXBUFLEN - 1, 0,
             (struct sockaddr *)&their_addr, &addr_len)) == -1) {
    perror( "Listener recvfrom");
    exit( 1);
  }

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

  close( sockfd);
  return 0;
}
```

Compile the server by:

```
gcc -Wall listener.c -o listener
```

A Simple UDP Client.

Using emacs enter the following client code:

```
    emacs talker.c &
```

```
/* talker.c - a datagram 'client'
 * need to supply host name/IP and one word message,
 * e.g. talker localhost hello
 */
#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>         /* for gethostbyname() */

#define PORT 4950     /* server port the client connects to */

int main( int argc, char * argv[]) {
  int sockfd, numbytes;
  struct hostent *he;
  struct sockaddr_in their_addr;    /* server address info */

  if( argc != 3) {
    fprintf( stderr, "usage: talker hostname message\n");
    exit( 1);
  }
  /* resolve server host name or IP address */
  if( (he = gethostbyname( argv[ 1])) == NULL) {
    perror( "Talker gethostbyname");
    exit( 1);
  }
```

```
  if( (sockfd = socket( AF_INET, SOCK_DGRAM, 0)) == -1) {
    perror( "Talker socket");
    exit( 1);
  }

  memset( &their_addr,0, sizeof(their_addr)); /* zero struct */
  their_addr.sin_family = AF_INET;    /* host byte order .. */
  their_addr.sin_port = htons( PORT); /* .. short, netwk byte order */
  their_addr.sin_addr = *((struct in_addr *)he -> h_addr);

  if( (numbytes = sendto( sockfd, argv[ 2], strlen( argv[ 2]), 0,
      (struct sockaddr *)&their_addr, sizeof( struct sockaddr))) == -1) {
    perror( "Talker sendto");
    exit( 1);
  }

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

  close( sockfd);
  return 0;
}
```

Notice that *sendto()* and *recvfrom()* use a socket number, a message buffer and length of message just as *write()* and *read()* do but they have extra fields. The fourth field specifies some flag bits to indicate such things as blocking and routing, we set this to zero to show no special needs. The fifth field is a pointer to the generic socket address of the recipient/sender of the datagram. The last field is the length of this generic socket address. This generic socket address is needed because we may use a different destination (or have a different source) each time the function is called within the program!

Compile the client by:

```
gcc -Wall talker.c -o talker
```

In one window run the server as a background process:

```
listener &
```

Use **ps** to check that it is running.

In another window run the client:

```
talker localhost hello
```

Note that, as soon as the communication is complete both server and client die. The server is no longer running (check with **ps**) so try running the client again. What happens? How is this different to the TCP datastream connection? Start the server up in the background. Does it manage to connect to the client?

Modifications.

1. Add code to the server and the client so that the message sent by the talker is echoed by the server to the client, which then displays it.

2. Modify the server to loop waiting to receive messages from clients.

3. Modify the client so that the server host name and the message to be sent may be entered as the program runs rather than being given at the command line.

A Simple Web Server.

Just to show that we can do something 'useful' with our sockets try this simple Web Server! It uses TCP datastream sockets and can interact with Web Browsers such as Netscape and Mozilla.

```
emacs web8000.c &
```

```
/* web8000.c a simple web server */
#define WEBROOT "/home/netlab/heinz/html"  /* base dir for html or txt
files */
#define HTTP_PORT 8000

#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>
#include <unistd.h>

int create_tcp_endpoint( int port);  /* prototype for fn to open socket */

void text( int fd, char * s) {       /* simple fn to write to file! */
  write( fd, s, strlen( s));
}

int main( void) {
  int fdnet, fd, count, sock;
  char filename[ 100], request[ 1024], reply[ 1024];

  /* Create an endpoint to listen on */
  if( (sock = create_tcp_endpoint( HTTP_PORT)) < 0) {
    fprintf( stderr, "Cannot create endpoint\n");
    exit( 1);
  }

  /* Enter the main service loop */
  while( 1) {
    /* Get a connection from a client */
    fdnet = accept( sock, NULL, NULL);
```

```
    fprintf( stderr, "connected on fd %d\n", fdnet);

    /* Read the GET request & build the file name */
    /* i.e. something like request[] = "GET /file.html HTTP/1.1" */
    read( fdnet, request, sizeof( request));

    strtok( request, " ");
    strcpy( filename, WEBROOT);
    strcat( filename, strtok( NULL, " "));
    printf( "Requested file is %s\n", filename);
    fd = open( filename, O_RDONLY);

    if( fd <0) {          /* check to see if the file is there */
      text( fdnet, "HTTP/1.1 404 Not Found\n");
      text( fdnet, "Content-Type: text/html\n\n");
      text( fdnet, "<HTML><BODY> Not Found </BODY></HTML>");
    }
    else {
      text( fdnet, "HTTP/1.1 200 OK\n");
      text( fdnet, "Content-Type: text/html\n\n");
      while( (count = read( fd, reply, 1024)) > 0)
        write( fdnet, reply, count);
    }
    close( fd);
    close( fdnet);
  } /* while( 1) */
}  /* main */


/* Useful function to create server endpoint */
int create_tcp_endpoint( int port) {
  int sock;
  struct sockaddr_in server;
  /* make socket with TCP streams. Kernel choose a suitable protocol */
  sock = socket( AF_INET, SOCK_STREAM, 0);
  if( sock < 0) return -1;       /* failed to make socket */

  memset( &server, 0, sizeof( server));        /* zero the struct */
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = htonl( INADDR_ANY); /* any server's IP addr */
  server.sin_port = htons( port);

  if( bind( sock, (struct sockaddr *) &server, sizeof( server)) < 0)
    return -2;                    /* failed to bind */

  listen( sock, 5); /* a listening socket with a max queue of 5 clients */
  return sock;
}
```

Notice that we have used a function to perform the standard act of creating, binding and setting up the listening queue. If an error occurs the function returns a negative value on which the main program can act (we just ignore the detail!).

Also notice that we must encapsulate our messages in HTTP protocol and possibly some HTML.

The WEBROOT defines the directory in which to find the html or text files the client can access. If this is left as “”, i.e. an empty string, then the full pathname of any file would need to be sent.

Compile the web server with:

```
gcc -Wall web8000.c -o web8000
```

Run the server in the background (`web8000 &`) and test it by using **telnet**:

```
telnet localhost 8000
```

We are using port number 8000 not the standard port 80!
We need to enter a simple HTTP request so try:

```
GET /name.html HTTP/1.1
```

If name.html does not exist in the WEBROOT we should get the HTTP response showing that the file was not found. Try getting a simple file (you could try getting the source code of one of your programs!).

Once satisfied that the sever works start a web browser (Netscape or Mozilla say) and enter the URL of the file, e.g.

```
http://localhost:8000/name.html
```

Note that we must supply the server hostname and the port number the server uses.

If the file does not exist we should see the Not Found message, otherwise we should get the displayed HTML page!

Try connecting to the simple web server of a colleague.

Modify your web server to display the message that it receives. This will show the format of the HTTP request message issued by the server. Does Netscape send the same request message as Mozilla?

Finally.

This has been just a taste of the functions available for socket programming. Have fun experimenting with these programs.

There is a lot of information on the internet and many books published on the subject. Look around try things out, don’t be afraid to experiment more than we have done here.

References.

Beej's Guide to Network Programming, Brian "Beej" Hall, www.ecst.csuchico.edu/~beej/guide/net.

Linux Format Magazine Jan 2003, www.linuxformat.co.uk.

Unix Network Programming,Vol. 1. 2nd Edn., W. Richard Stevens, Prentice- Hall Pearson Education.

Linux Socket Programming by Example, Warren W. Gay, Que.