


UDP

1. User Datagram Protocol

1.1 CSCI 330

CSCI 330

UNIX and Network Programming



UDP
User Datagram Protocol



1.2 Unit Overview

Unit Overview

- Transport layer
- User datagram protocol
- UDP programming
- Example UDP client/server programs

1.3 Review: Network Layer

Review: Network Layer

- also called: Internet Protocol (IP) Layer
 - provides host to host transmission service, where hosts are not necessarily adjacent
- IP layer provides services:
 - addressing
 - hosts have global addresses: IPv4, IPv6
 - routing and forwarding
 - find path from host to host

1.4 Transport Layer

Transport Layer

- provides end-to-end communication services for applications
- provides multiple endpoints on a single node: port
- TCP: transmission control protocol
 - connection oriented, guaranteed delivery
 - stream oriented: basis for: http, ftp, smtp, ssh
- UDP: user datagram protocol
 - best effort
 - datagram oriented: basis for: dns, rtp

1.5 UDP

UDP

- simple message-based connection-less protocol
 - transmits information in one direction from source to destination without verifying the readiness or state of the receiver
- uses datagram as message
- stateless and fast

1.6 UDP packet format

UDP packet format

bits	0 – 7	8 – 15	16 – 23	24 – 31
0	Source IP address			
32	Destination IP address			
64	Zeros	Protocol	UDP length	
96	Source Port		Destination Port	
128	Length		Checksum	
160+	Data			

1.7 UDP programming

UDP programming

- common abstraction: socket
 - first introduced in BSD Unix in 1981
- socket is end-point of communication link
 - identified as IP address + port number
 - can receive data, can send data
- typical logic: server vs. client
 - server ready to receive datagram from any client
 - client sends datagram to specific server
 - server responds with datagram to client

1.8 Socket system calls

Socket system calls

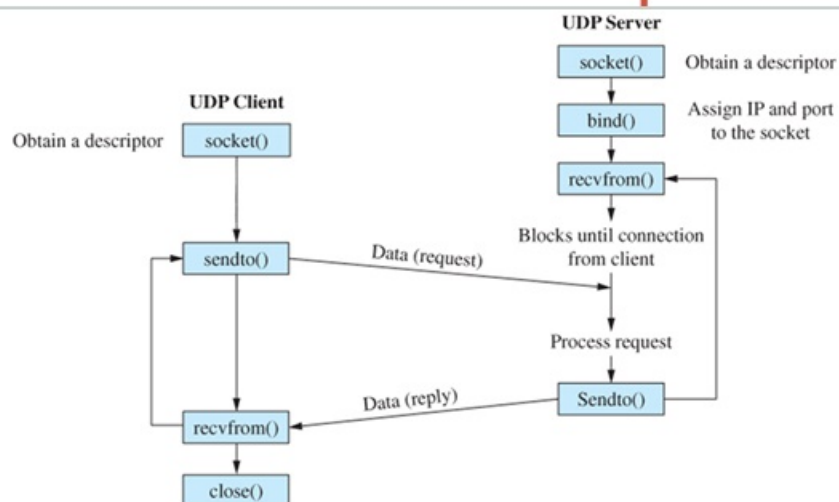
server

client

System call	Meaning
socket	Create a new communication endpoint
bind	Attach a local address to a socket
recvfrom	Receive(read) some data over the connection
sendto	Send(write) some data over the connection
close	Release the connection

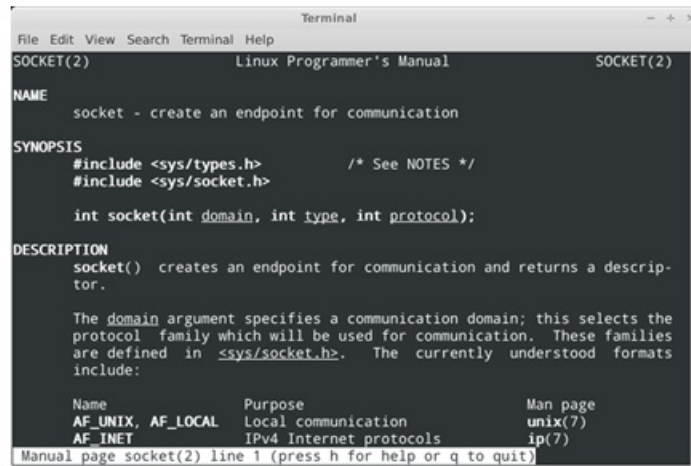
1.9 UDP communications pattern

UDP communications pattern



1.10 System call: socket

System call: socket



```
Terminal
File Edit View Search Terminal Help
SOCKET(2) Linux Programmer's Manual SOCKET(2)

NAME
    socket - create an endpoint for communication

SYNOPSIS
    #include <sys/types.h>          /* See NOTES */
    #include <sys/socket.h>

    int socket(int domain, int type, int protocol);

DESCRIPTION
    socket() creates an endpoint for communication and returns a descriptor.

    The domain argument specifies a communication domain; this selects the
    protocol family which will be used for communication. These families
    are defined in <sys/socket.h>. The currently understood formats
    include:

    Name                Purpose                Man page
    AF_UNIX, AF_LOCAL    Local communication    unix(7)
    AF_INET               IPv4 Internet protocols    ip(7)

Manual page socket(2) line 1 (press h for help or q to quit)
```

1.11 System call: socket

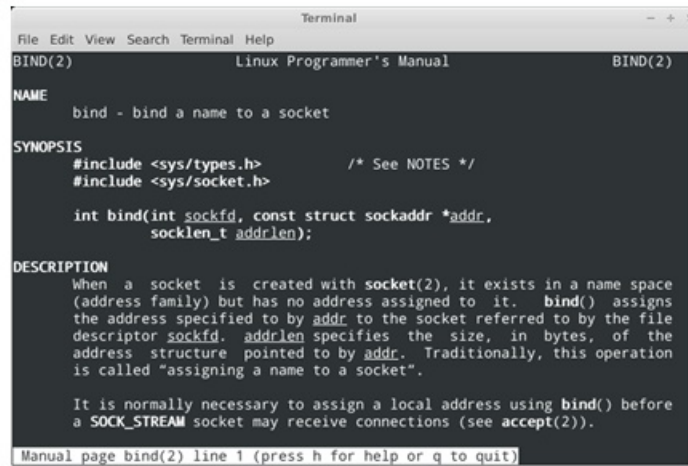
System call: socket

```
int socket(int domain, int type, int protocol)
```

- creates a new socket, as end point to a communications link
- **domain** is set to **AF_INET**
- **type** is set to **SOCK_DGRAM** for datagrams
- **protocol** is set to 0, i.e. default UDP
- returns socket descriptor:
 - used in **bind**, **sendto**, **recvfrom**, **close**

1.12 System call: bind

System call: bind



```
Terminal
File Edit View Search Terminal Help
BIND(2) Linux Programmer's Manual BIND(2)

NAME
bind - bind a name to a socket

SYNOPSIS
#include <sys/types.h>      /* See NOTES */
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);

DESCRIPTION
When a socket is created with socket(2), it exists in a name space
(address family) but has no address assigned to it. bind() assigns
the address specified to by addr to the socket referred to by the file
descriptor sockfd. addrlen specifies the size, in bytes, of the
address structure pointed to by addr. Traditionally, this operation
is called "assigning a name to a socket".

It is normally necessary to assign a local address using bind() before
a SOCK_STREAM socket may receive connections (see accept(2)).

Manual page bind(2) line 1 (press h for help or q to quit)
```

1.13 System call: bind

System call: bind

```
int bind(int sockfd,
         const struct sockaddr *addr,
         socklen_t addrlen)
```

- assigns address to socket: IP number and port
- **struct sockaddr** holds address information
 - will accept **struct sockaddr_in** pointer
- **addrlen** specifies length of **addr** structure
- returns 0 on success, -1 otherwise

1.14 structure sockaddr: 16bytes

structure sockaddr: 16bytes

```
struct sockaddr {
    short  sa_family; /* address family */
    char  sa_data[14]; /* address data */
};

struct sockaddr_in {
    short      sin_family; /* address family */
    unsigned short sin_port; /* port number: 2 bytes */
    struct in_addr sin_addr; /* IP address: 4 bytes */
    /* pad to size of struct sockaddr */
    char      sin_zero[8];
};
```

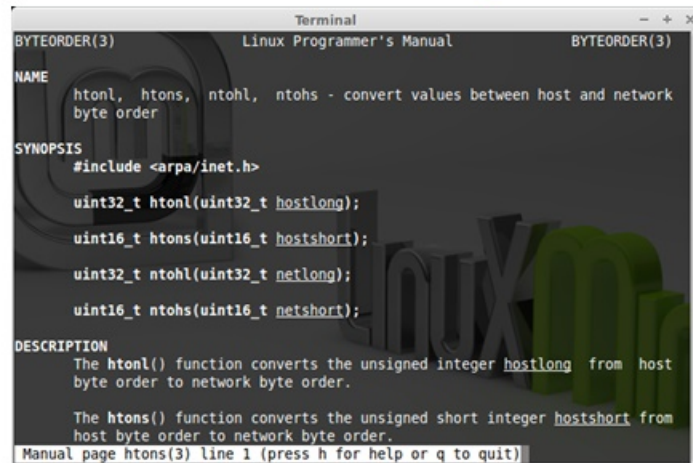
1.15 structure sockaddr_in: members

structure sockaddr_in: members

- `sin_family` /* address family */
always: `AF_INET`
- `sin_port` /* port number: 2 bytes */
`htons(4444)` /* ensure network order */
- `sin_addr` /* Internet address: 4 bytes */
`s_addr = INADDR_ANY`
`s_addr = inet_addr("127.0.0.1")`

1.16 Helper Functions: byte order

Helper Functions: byte order



```
Terminal
Linux Programmer's Manual
BYTEORDER(3)

NAME
    htonl, htons, ntohl, ntohs - convert values between host and network
    byte order

SYNOPSIS
    #include <arpa/inet.h>

    uint32_t htonl(uint32_t hostlong);
    uint16_t htons(uint16_t hostshort);
    uint32_t ntohl(uint32_t netlong);
    uint16_t ntohs(uint16_t netshort);

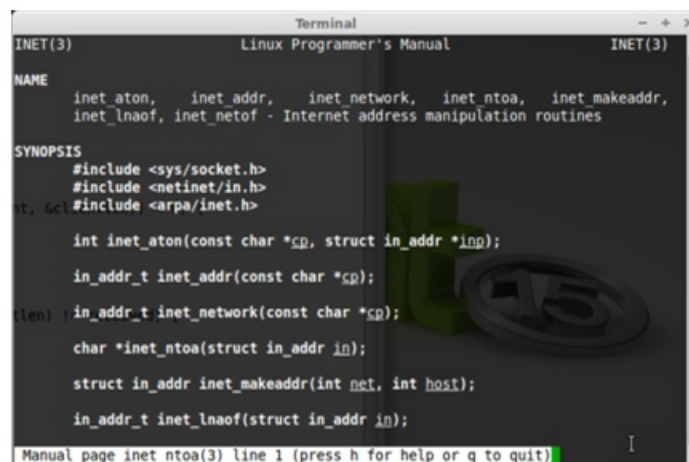
DESCRIPTION
    The htonl() function converts the unsigned integer hostlong from host
    byte order to network byte order.

    The htons() function converts the unsigned short integer hostshort from
    host byte order to network byte order.

    Manual page htons(3) line 1 (press h for help or q to quit)
```

1.17 Helper Functions: Address Manipulation

Helper Functions: Address Manipulation



```
Terminal
Linux Programmer's Manual
INET(3)

NAME
    inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr,
    inet_lnaof, inet_netof - Internet address manipulation routines

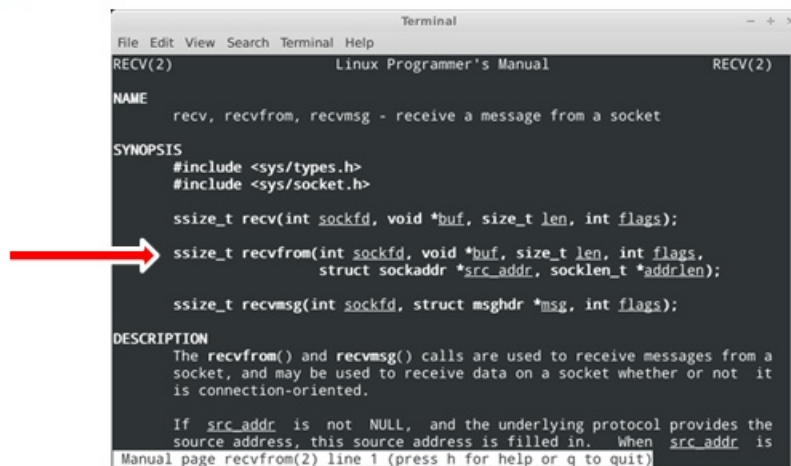
SYNOPSIS
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>

    int inet_aton(const char *cp, struct in_addr *inp);
    in_addr_t inet_addr(const char *cp);
    in_addr_t inet_network(const char *cp);
    char *inet_ntoa(struct in_addr in);
    struct in_addr inet_makeaddr(int net, int host);
    in_addr_t inet_lnaof(struct in_addr in);

    Manual page inet_ntoa(3) line 1 (press h for help or q to quit)
```

1.18 System call: recvfrom

System call: recvfrom



```
Terminal
File Edit View Search Terminal Help
RECV(2) Linux Programmer's Manual RECV(2)

NAME
recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);

DESCRIPTION
The recvfrom() and recvmsg() calls are used to receive messages from a
socket, and may be used to receive data on a socket whether or not it
is connection-oriented.

If src_addr is not NULL, and the underlying protocol provides the
source address, this source address is filled in. When src_addr is
Manual page recvfrom(2) line 1 (press h for help or q to quit)
```

1.19 System call: recvfrom

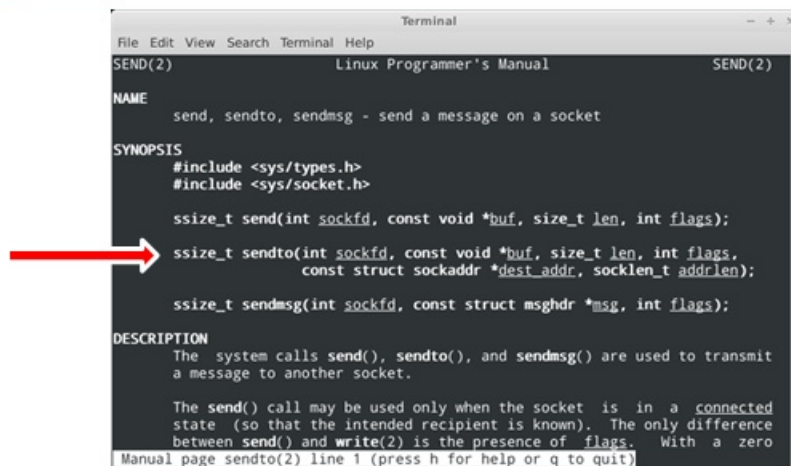
System call: recvfrom

```
ssize_t recvfrom(int sockfd, void *buf, size_t len,
                int flags, struct sockaddr *src_addr,
                socklen_t *addrlen)
```

- receives a datagram **buf** of size **len** from socket **sockfd**
 - will wait until a datagram is available
 - **flags** specifies wait behavior, e.g.: 0 for default
- **src_addr** will hold address information of sender
 - **struct sockaddr** defines address structure
 - **addrlen** specifies length of **src_addr** structure
- returns the number of bytes received, i.e. size of datagram

1.20 System call: sendto

System call: sendto



```
Terminal
File Edit View Search Terminal Help
SEND(2) Linux Programmer's Manual SEND(2)

NAME
    send, sendto, sendmsg - send a message on a socket

SYNOPSIS
    #include <sys/types.h>
    #include <sys/socket.h>

    ssize_t send(int sockfd, const void *buf, size_t len, int flags);
    ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                   const struct sockaddr *dest_addr, socklen_t addrlen);
    ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);

DESCRIPTION
    The system calls send(), sendto(), and sendmsg() are used to transmit
    a message to another socket.

    The send() call may be used only when the socket is in a connected
    state (so that the intended recipient is known). The only difference
    between send() and write(2) is the presence of flags. With a zero
    Manual page sendto(2) line 1 (press h for help or q to quit)
```

1.21 System call: sendto

System call: sendto

```
ssize_t sendto(int sockfd,
               const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen)
```

- sends datagram **buf** of size **len** to socket **sockfd**
 - will wait if there is no ready receiver
 - flags specifies wait behavior, e.g.: 0 for default
- **dest_addr** holds address information of receiver
 - **struct sockaddr** defines address structure
 - **addrlen** specifies length of **dest_addr** structure
- returns the number of bytes sent, i.e. size of datagram

1.22 System Call: close

System Call: close

```
int close(int fd)
```

- closes ~~file~~ specified by ~~fd~~ ^{socket} ~~file~~ descriptor
- returns zero on success

1.23 Example: UDP Programming

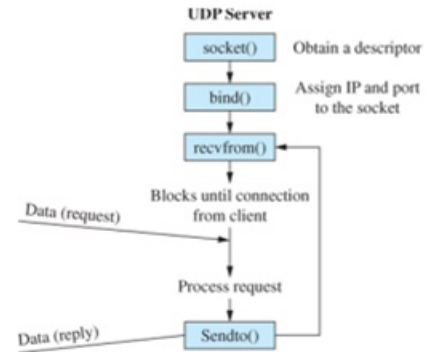
Example: UDP Programming

- simple server: echo
 - receives datagrams, sends them back to sender
- simple client
 - sends datagram to server, receives response

1.24 Illustration: echoServer.cxx

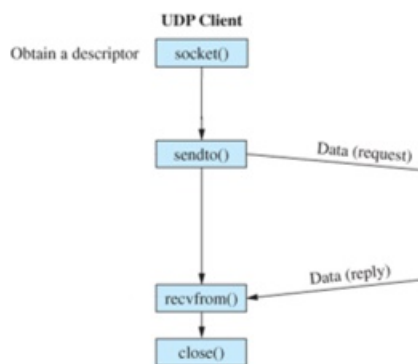
Illustration: echoServer.cxx

```
echoServer.cxx
23 int main(int argc, char *argv[]) {
24     if (argc != 2) {
25         cerr << "Usage: echoServer port\n";
26         exit(EXIT_FAILURE);
27     }
28     char buffer[256];
29     int received = 0;
30
31     // Create the UDP socket
32     struct sockaddr_in server_address; // structure for address of server
33     struct sockaddr_in client_address; // structure for address of client
34     unsigned int addrlen = sizeof(client_address);
35
36     // Create the UDP socket
37     if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
38         perror("Failed to create socket");
39         exit(EXIT_FAILURE);
40     }
41
42     // Construct the server sockaddr in structure
43     memset(&server_address, 0, sizeof(server_address)); // Clear struct */
44     server_address.sin_family = AF_INET; // Internet/IPv4 */
45     server_address.sin_addr.s_addr = INADDR_ANY; // Any IP address */
46     server_address.sin_port = htons(atoi(argv[1])); // server port */
47
48     // Bind the socket
49     if (bind(sock, (struct sockaddr *) &server_address, sizeof(server_address)) < 0) {
50         perror("Failed to bind server socket");
51         exit(EXIT_FAILURE);
52     }
53
54     cout << "echoServer listening on port: " << argv[1] << endl;
55
56     // Run until cancelled
57     while (true) {
58         // Receive a message from the client
59         if ((received = recvfrom(sock, buffer, 256, 0, (struct sockaddr *) &client_address, &addrlen)) < 0) {
60             perror("recvfrom");
61             exit(EXIT_FAILURE);
62         }
63         // Send the message back to client
64         if (sendto(sock, buffer, received, 0, (struct sockaddr *) &client_address, addrlen) < 0) {
65             perror("sendto");
66             exit(EXIT_FAILURE);
67         }
68     }
69     close(sock);
70     return 0;
71 }
```



1.25 Illustration: echoClient.cxx

Illustration: echoClient.cxx



```
echoClient.cxx
25 int main(int argc, char *argv[]) {
26     if (argc != 4) {
27         cerr << "Usage: echoClient server_ip port message\n";
28         exit(EXIT_FAILURE);
29     }
30     char buffer[256];
31     int echolen, received = 0;
32
33     // Create the UDP socket
34     struct sockaddr_in server_address; // structure for address of server
35     unsigned int addrlen = sizeof(server_address);
36
37     // Create the UDP socket
38     if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
39         perror("Failed to create socket");
40         exit(EXIT_FAILURE);
41     }
42
43     // Construct the server sockaddr in structure
44     memset(&server_address, 0, sizeof(server_address)); // Clear struct */
45     server_address.sin_family = AF_INET; // Internet/IPv4 */
46     server_address.sin_addr.s_addr = inet_addr(argv[1]); // IP address */
47     server_address.sin_port = htons(atoi(argv[2])); // server port */
48
49     // Send the message to the server (don't forget to count the terminating null)
50     echolen = strlen(argv[3]) + 1;
51     if (sendto(sock, argv[3], echolen, 0, (struct sockaddr *) &server_address, sizeof(server_address)) != echolen) {
52         perror("sendto");
53         exit(EXIT_FAILURE);
54     }
55
56     // Receive the message back from the server
57     if ((received = recvfrom(sock, buffer, 256, 0, (struct sockaddr *) &server_address, &addrlen)) != echolen) {
58         perror("recvfrom");
59         exit(EXIT_FAILURE);
60     }
61
62     cout << "Server (" << inet_ntoa(server_address.sin_addr) << ") echoed: " << received << " bytes: " << buffer << endl;
63     close(sock);
64     return 0;
65 }
```


1.26 Server detail: socket setup

Server detail: socket setup

```
int sock;
struct sockaddr_in server_address; // structure for address of server
struct sockaddr_in client_address; // structure for address of client
unsigned int addrlen = sizeof(client_address);

// Create the UDP socket
if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("Failed to create socket");
    exit(EXIT_FAILURE);
}

// Construct the server sockaddr_in structure
memset(&server_address, 0, sizeof(server_address)); /* Clear struct */
server_address.sin_family = AF_INET; /* Internet/IP */
server_address.sin_addr.s_addr = INADDR_ANY; /* Any IP address */
server_address.sin_port = htons(atoi(argv[1])); /* server port */

// Bind the socket
if (bind(sock, (struct sockaddr *) &server_address, sizeof(server_address)) < 0) {
    perror("Failed to bind server socket");
    exit(EXIT_FAILURE);
}
```

1.27 Server detail: receive/send loop

Server detail: receive/send loop

```
// Run until cancelled
while (true) {
    // Receive a message from the client
    if ((received = recvfrom(sock, buffer, 256, 0, (struct sockaddr *) &client_address, &addrlen)) < 0) {
        perror("recvfrom");
        exit(EXIT_FAILURE);
    }
    cerr << "Client (" << inet_ntoa(client_address.sin_addr) << ") sent " << received << " bytes: " << buffer << endl;
    // Send the message back to client
    if (sendto(sock, buffer, received, 0, (struct sockaddr *) &client_address, addrlen) < 0) {
        perror("sendto");
        exit(EXIT_FAILURE);
    }
}
```

1.28 Client detail: socket setup

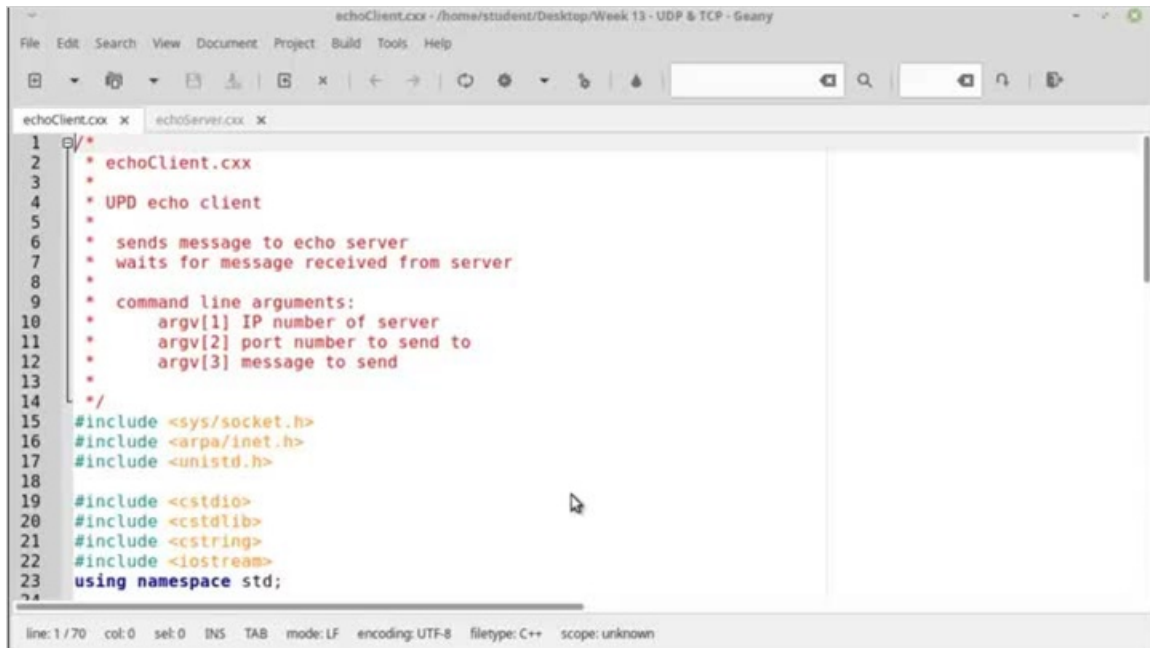
Client detail: socket setup

```
int sock;
struct sockaddr_in server_address; // structure for address of server
unsigned int addrlen = sizeof(server_address);

// Create the UDP socket
if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("Failed to create socket");
    exit(EXIT_FAILURE);
}

// Construct the server sockaddr_in structure
memset(&server_address, 0, sizeof(server_address)); /* Clear struct */
server_address.sin_family = AF_INET;                /* Internet/IP */
server_address.sin_addr.s_addr = inet_addr(argv[1]); /* IP address */
server_address.sin_port = htons(atoi(argv[2]));    /* server port */
```

1.29 Echo Example



The screenshot shows a code editor window titled "echoClient.cxx - /home/student/Desktop/Week 13 - UDP & TCP - Geany". The editor displays the header and include sections of the "echoClient.cxx" file. The code is as follows:

```
1  /*
2  * echoClient.cxx
3  *
4  * UDP echo client
5  *
6  * sends message to echo server
7  * waits for message received from server
8  *
9  * command line arguments:
10 *   argv[1] IP number of server
11 *   argv[2] port number to send to
12 *   argv[3] message to send
13 *
14 */
15 #include <sys/socket.h>
16 #include <arpa/inet.h>
17 #include <unistd.h>
18
19 #include <cstdio>
20 #include <cstdlib>
21 #include <cstring>
22 #include <iostream>
23 using namespace std;
```

The status bar at the bottom indicates: line: 1 / 70, col: 0, sel: 0, INS, TAB, mode: LF, encoding: UTF-8, filetype: C++, scope: unknown.

1.30 Summary

Summary

- Transport layer
- User datagram protocol
- UDP programming
- Example UDP client/server programs