DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING
[INQ0091623] AA 2021-22

Lab 8

Distributed IPC with Sockets

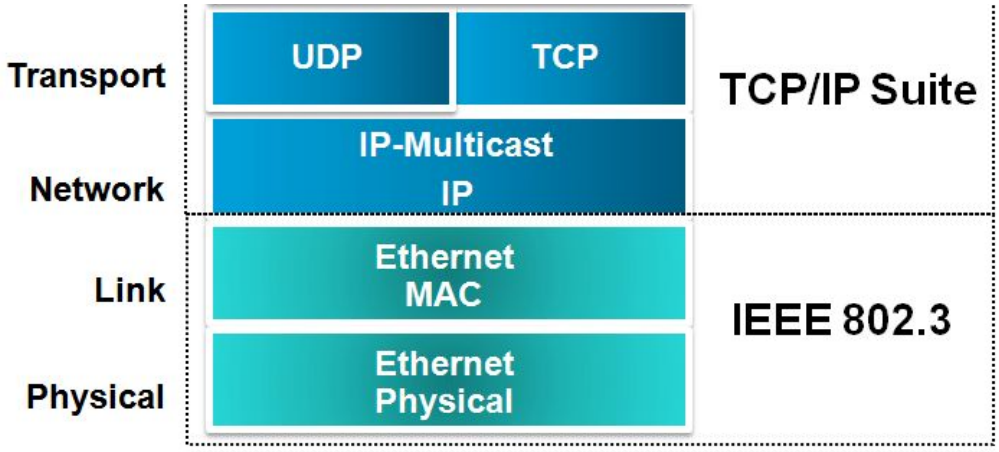
Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

REPLAY: Network and Transport

Many different network communication layers are defined for different network protocols, and every layer, normally built on top of one or more other layers, provides some added functionality in respect of that provided by the layers below.

Here we shall consider the **Internet Protocol (IP)**, which addresses the Network Layer, and the **Transmission Control Protocol (TCP)**, addressing the Transport Layer. IP and TCP together implement the well-known **TCP/IP protocol**. Both protocols are specified and discussed in detail in a number of Request for Comments (RFC), a series of informational and standardization documents about Internet.

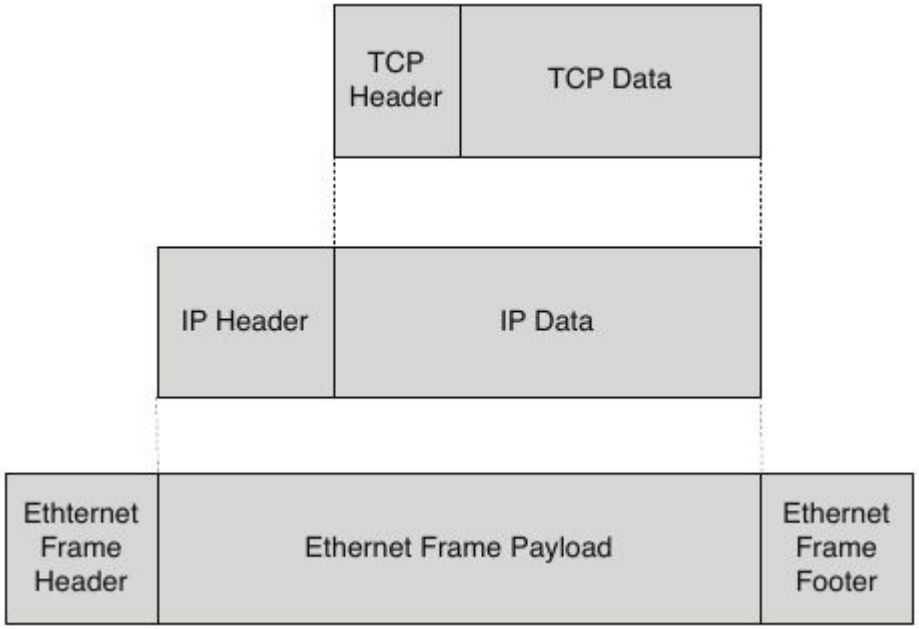


REPLAY: TCP/IP Stack

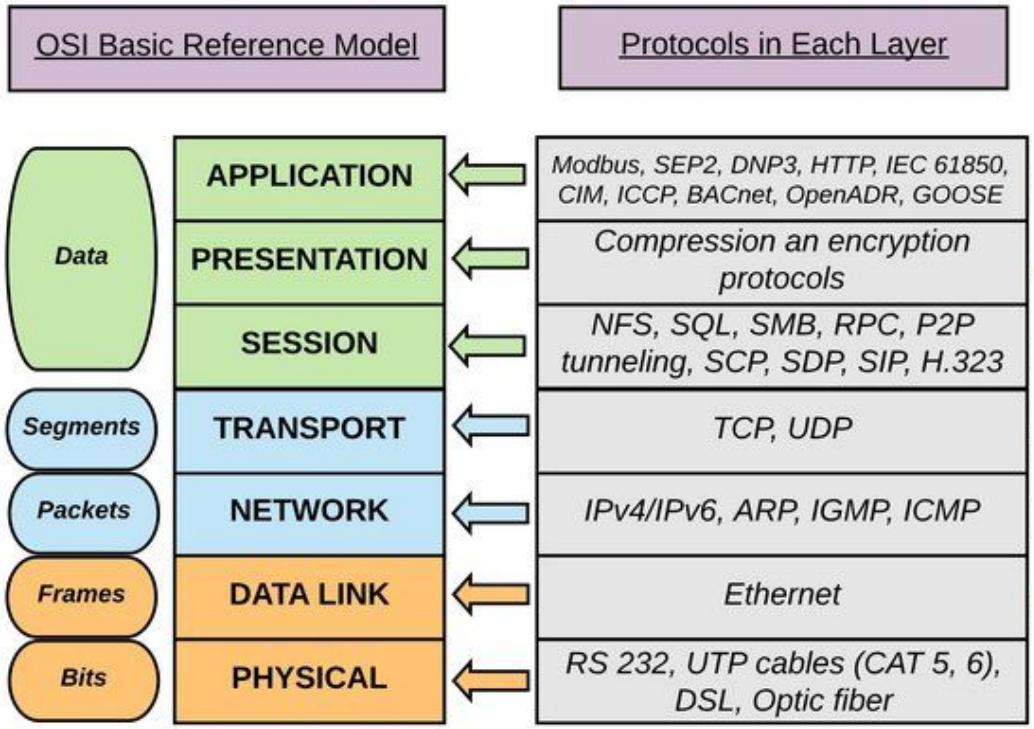
All protocols are organized that one layer knows nothing about the upper layers.

Being the TCP layer built on top of the Internet layer, the latter cannot know anything about the structure of the TCP data packet, which is contained in the data part of the Internet packet and considered payload.

For this reason this organization is called the TCP Stack.



REPLAY: OSI model vs Protocols



User **Datagram** Protocol

Stream vs Datagram

A network socket connection can be of two main kinds:

- **Stream:** pipe of ordered data in a full duplex point to point connection
- **Datagram:** connectionless communication of packets

STREAM (TCP)

A stream socket is like a phone call -- one side places the call, the other answers, you **say hello to each other** (SYN/ACK in TCP), and then you exchange information. Once you are done, **you say goodbye** (FIN/ACK in TCP). If one side doesn't hear a goodbye, they will usually call the other back since this is an unexpected event; usually the client will reconnect to the server. There is a guarantee that **data will arrive in the order you sent** it, and there is a reasonable guarantee that **data will not be damaged**.

DATAGRAM (UDP)

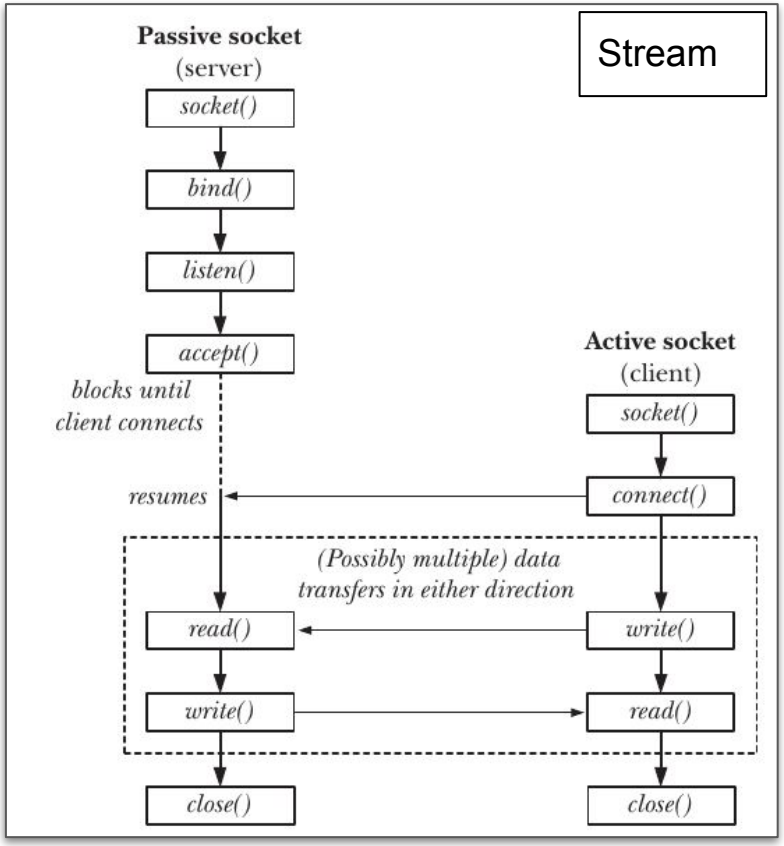
A datagram is a simpler form of communication… like a text message you sent to a friend or a group. The note may not reach its destination, and it may be modified by the time it gets there. If you pass two notes to the same person, they may arrive in an order you didn't intend.

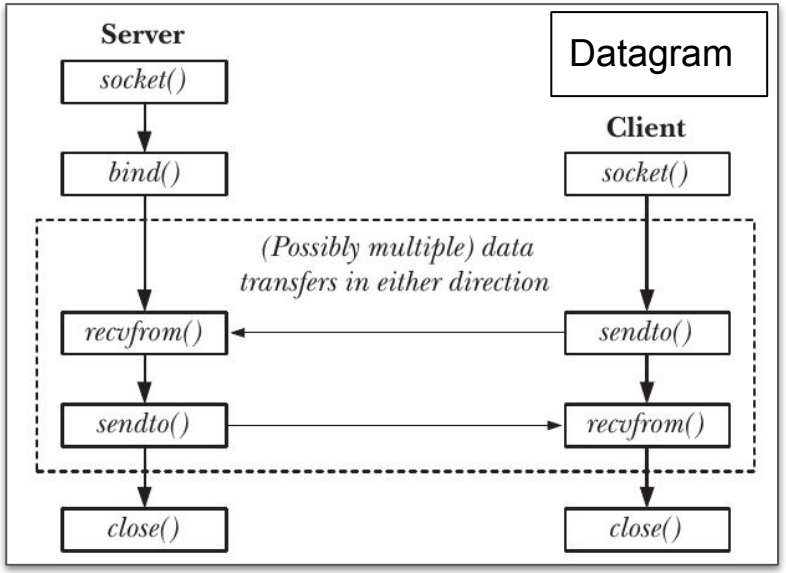
Socket stream vs datagram

The presence of connection in stream sockets leads to a different application programming pattern.

Stream: On one side we have a *listen() accept()* couple that identifies the server (holding connection and passively waiting for incoming requests). On the other side the client perform the *connect()* that actively instantiate the connection.

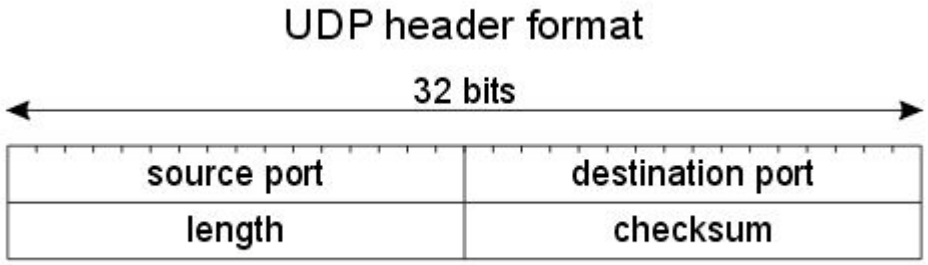
Datagram: The recvfrom() and sendto() system calls simply receive and send datagrams on a datagram socket. There is no need for an explicit connection.





UDP

Unlike the TCP (stream) connection the datagram does not need much information to be exchanged between the sender and the receiver. The protocol header can be much shorter. The **User Datagram Protocol (UDP)** is used in this case:



Datagrams the sockets (SOCK_DGRAM) allow data to be exchanged in the form of messages called datagrams (or datagram packets).

Unlike a stream socket, a datagram socket doesn't need to be connected to another socket in order to be used.

Send and Receive data to/from socket

Whether we have a connection (stream) or not (datagram) we can always use the following receive and send data functions:

```
#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);

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);
```

FLAGS:

MSG_OOB
This flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols.

MSG_PEEK
This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.

MSG_WAITALL (since Linux 2.2)
This flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned.

MSG_DONTWAIT (since Linux 2.2)
Enables non-blocking operation; if the operation would block, the call fails with the error **EAGAIN** or **EWOULDBLOCK** (this can also be enabled using the **O_NONBLOCK** flag).

Read and Write to socket

As we said that the socket is just like a file descriptor we can also decide to use the standard file operations like read and write buffers:

```
#include <unistd.h>

 ssize_t read(int fd, void *buf, size_t count);
 ssize_t write(int fd, const void *buf, size_t count);
```

NOTE on DATAGRAMS:

On datagrams we usually don't connect as the "connectionless link" but a connect() fake call can be performed even for datagrams that does not actually do any handshake with target but simply register the destination target within the socket.

- Using read() on a UDP socket is perfectly fine if you don't need the source address.
- You can use write() only if you connect() the UDP socket to the destination.

Out Of Band data on streams

Streams with connections (TCP) permit **out-of-band data** that is delivered with **higher priority** than ordinary data.

Typically the reason for sending out-of-band data is to send notice of an exceptional condition. To send out-of-band data use send, specifying the flag MSG_OOB

Out-of-band data are received with higher priority because the receiving process need not read it in sequence; to read the next available out-of-band data, use recv with the MSG_OOB flag.

Ordinary read operations do not read out-of-band data; they read only ordinary data.

- You can test for pending out-of-band data, or wait until there is out-of-band data, using the **select( )** function; it can wait also for an exceptional condition on the socket.

- Alternatively, when a socket finds that out-of-band data are on their way, it sends a **SIGURG signal** to the owner process or process group of the socket. You must also establish a handler for this signal, (We have not seen signals yet Signal Handling … we will see them in the next lab).

UDP SERVER simple example

CODING EXAMPLE

CODING EXAMPLE

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT    8080
#define MAXLINE 1024

int main() {
   int sockfd;
   char buffer[MAXLINE];
   char *hello = "Hello from server";
   struct sockaddr_in servaddr, cliaddr;

   // Creating socket file descriptor
   if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
       perror("socket creation failed");
       exit(EXIT_FAILURE);
   }

   memset(&servaddr, 0, sizeof(servaddr));
   memset(&cliaddr, 0, sizeof(cliaddr));

   // Filling server information
   servaddr.sin_family    = AF_INET; // IPv4
   servaddr.sin_addr.s_addr = INADDR_ANY;
   servaddr.sin_port = htons(PORT);

   // Bind the socket with the server address
   if ( bind(sockfd, (const struct sockaddr *)&servaddr,
           sizeof(servaddr)) < 0 ){
       perror("bind failed");
       exit(EXIT_FAILURE);
   }
```

```c
   int len, n;

   len = sizeof(cliaddr);  //len is value/resuslt

   n = recvfrom(sockfd, (char *)buffer, MAXLINE,
               MSG_WAITALL, ( struct sockaddr *) &cliaddr,
               &len);
   buffer[n] = '\0';
   printf("Client : %s\n", buffer);
   sendto(sockfd, (const char *)hello, strlen(hello),
       MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
           len);
   printf("Hello message sent.\n");

   return 0;
}
```

UDP CLIENT simple example

CODING EXAMPLE

CODING EXAMPLE

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT    8080
#define MAXLINE 1024

// Driver code
int main() {
   int sockfd;
   char buffer[MAXLINE];
   char *hello = "Hello from client";
   struct sockaddr_in   servaddr;

   // Creating socket file descriptor
   if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
       perror("socket creation failed");
       exit(EXIT_FAILURE);
   }

   memset(&servaddr, 0, sizeof(servaddr));

   // Filling server information
   servaddr.sin_family = AF_INET;
   servaddr.sin_port = htons(PORT);
   servaddr.sin_addr.s_addr = INADDR_ANY;

   int n, len;
   sendto(sockfd, (const char *)hello, strlen(hello),
       MSG_CONFIRM, (const struct sockaddr *) &servaddr,
           sizeof(servaddr));
   printf("Hello message sent.\n");

   n = recvfrom(sockfd, (char *)buffer, MAXLINE,
               MSG_WAITALL, (struct sockaddr *) &servaddr,
               &len);
   buffer[n] = '\0';
   printf("Server : %s\n", buffer);

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

NOTE: MSG_CONFIRM

Did you note anything weird on the code for UDP connection?

Why is the MSG_CONFIRM flag there ?!?!

Essentially, all it does is to tell the underlying ARP network layer to NOT to periodically verify the MAC of the recipient IP (which would update the ARP hardware-MAC-address-to-IP-address mapping), because we are confident that the IP we are sending to is actually the device we think it is, since this message are are sending is in **direct response to** a message we just **received** from them!

In other words, if in doubt, leave OUT the MSG_CONFIRM flag. Put it in ONLY if the message we are sending is a direct response to a message received, and therefore we want to increase the network efficiency (a tiny bit) by NOT verifying the MAC again periodically, at the risk that the destination MAC could change and be wrong and it no longer matches the IP address for the device we think we are sending this message to!

Pros of using MSG_CONFIRM:

1. It increases the efficiency of the network by NOT having ARP reprobe for the MAC of the destination address.

Cons or risks of using MSG_CONFIRM:

1. It may mean that if the old device on the network drops off, and a new destination device pops up on the network with the same destination address as the old one, but with a different hardware MAC, we won't know, because MSG_CONFIRM tells ARP *not* to reprobe that IP address MAC.

When MSG_CONFIRM can be used:

1. When our message out is a direct reply to a message we just received in from that device, so we are sure its MAC is still correct.
2. MSG_CONFIRM in this case can be thought of as a "confirmation" message to the sender that we got their previous message.

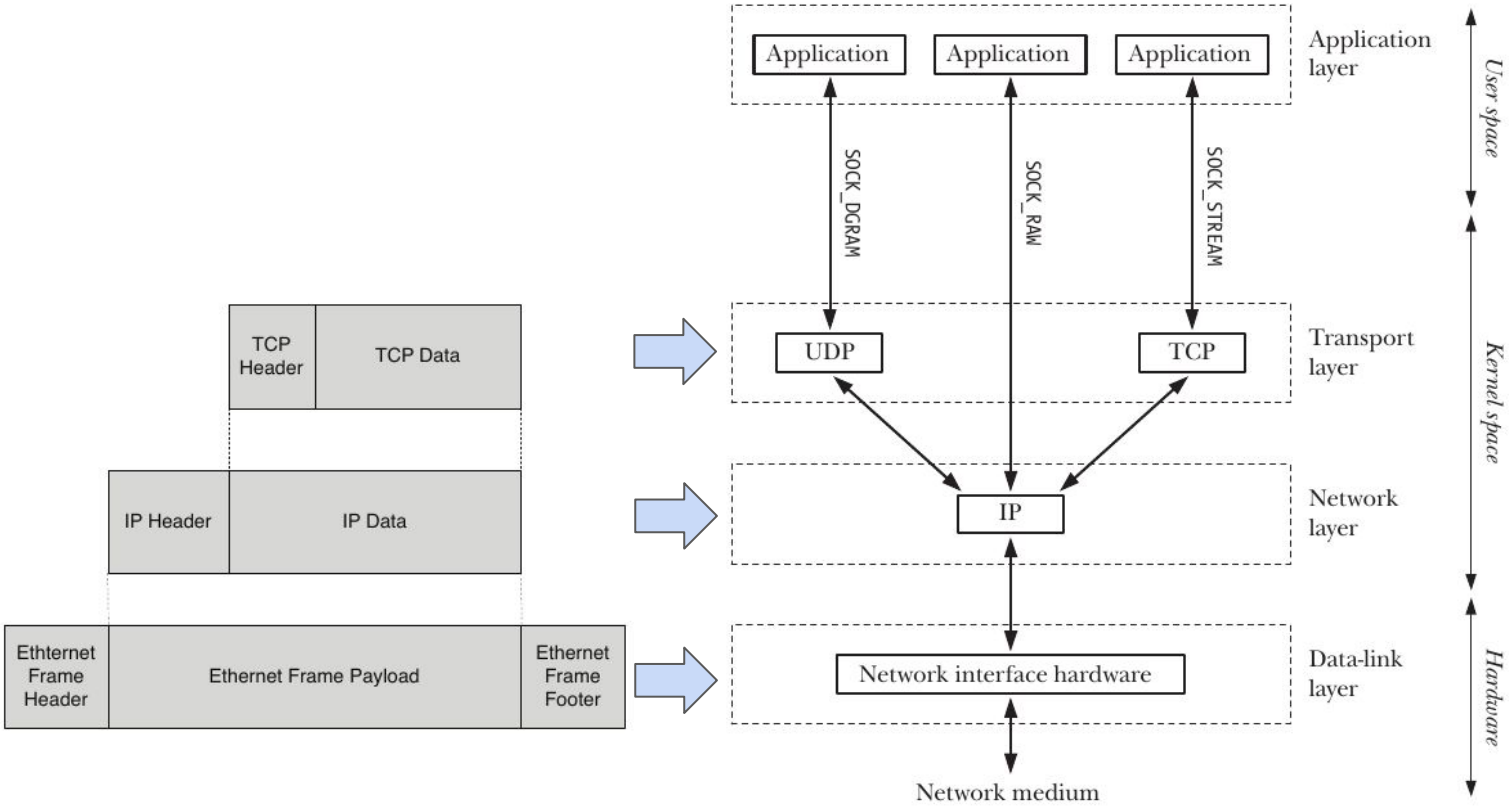
SUMMARY: TCP vs UDP

TCP provides reliable data, while UDP does not … “Why use UDP at all?”

1. A UDP server can receive (and reply to) datagrams from multiple clients, without needing to create and terminate a connection for each client
2. For simple request-response communications, UDP can be faster than TCP, since it doesn’t require connection establishment and termination.
3. **UDP sockets permit broadcasting and multicasting.**
4. In **real-time applications** the delay that may occur after TCP tries to recover from a lost segment may result in transmission delays that are unacceptably long.

ITEM	TCP/IP	UDP
Connection	Connection-oriented protocol - Established through a “handshake” before data can be sent	Connectionless protocol - Packets are sent individually and transported on top of IP.
Packet Entity	Segments	Datagram
Reliability	Reliable byte stream - messaging is managed and acknowledged. TCP does error checking and has recovery methods as well as traffic congestion control	Error checking through “checksum” but no recovery options.
Ordering	Transmissions are sent in sequence.	Time sensitive and time preferential. No sequencing or ordering of messages.
Recovery Methods	Lost or discarded packets are resent	No recovery
Applications	High-reliability and less critical transmission time - World Wide Web, file transfer, e-mail	Fast timing - Real time streaming protocol – Multicasting - Voice over IP, online gaming, video streaming, IPTV
Priority	Low priority since timing is not as important and delivery is confirmed	First priority due to application’s time sensitivity

SUMMARY:


TCP Header
TCP Data
IP Header
IP Data
Ethternet Frame Header
Ethernet Frame Payload
Ethernet Frame Footer
Application
Application
Application
Application layer
SOCK_DGRAM
SOCK_RAW
SOCK_STREAM
UDP
TCP
Transport layer
IP
Network layer
Network interface hardware
Data-link layer
Network medium
User space
Kernel space
Hardware

Set Socket Options

Socket Options

```
#include <sys/socket.h>

int getsockopt(int socket, int level, int option_name,
    void *restrict option_value, socklen_t *restrict option_len);

int setsockopt(int socket, int level, int option_name,
    const void *option_value, socklen_t option_len);
```

Socket options affect various features of the operation of a socket.

SO_REUSEADDR
SO_KEEPALIVE
SO_LINGER
SO_BROADCAST
SO_OOBINLINE
SO_SNDBUF
SO_RCVBUF
SO_TYPE
SO_ERROR

Socket Options SO_REUSEADDR

In some examples we used on lab there is a specific invocation of **setsockopt** setting option SO_REUSEADDR.

this option allows the address/port couple to be binded more than once in the system. Once a connection (or a datagram send) is requested for that port the system stack assign that request to the first active socket found.

```
int optval = 1; // true
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) == -1)
  errExit("socket");
```

In our code this was actually meant to overcome the possible error: EADDRINUSE **“Address already in use”**.

There are two scenarios in which this usually occurs:

- A previous invocation of the server that was connected to a client that performed a **close**, either by calling close(), or by crashing (e.g., it was killed by a signal). This leaves a TCP endpoint that remains in the TIME_WAIT state until the 2MSL timeout expires (usually 20 seconds).

- A previous invocation of the server created a child process to handle a connection to a client. Later, the server terminated, while **the child continues to serve** the client, and thus maintain a TCP endpoint using the server’s well-known port.

Socket Options SO_KEEPALIVE

Any server that thinks it has a connected client must dedicate some resources to it. If the server is of the forking type, then an entire Linux process with its associated memory is dedicated to that client. When things are going well, this scenario does not present any problem. The difficulty arises when a network disruption occurs, and all your clients become disconnected from your service.

After the network service is restored, all the clients will be attempting to connect to your server at the same time, as they need to re-establish connections. This is a real problem for you because your server has not yet realized that it lost the idle clients earlier so:

- it duplicates the needed resources for a while,
- it generates a lot of traffic to re-establish all connections.

If the SO_KEEPALIVE option is active when the socket connection is idle for long periods, a probe message is sent to the remote end (after two hours of inactivity). There are three possible responses to a keep-alive probe message:

1. The peer responds appropriately to indicate that all is well. No indication is returned to the application, because this is the application's assumption to begin with.

2. The peer can respond indicating that it knows nothing about the connection. This indicates that the peer has been rebooted since the last communication with that host. The error ECONNRESET will then be returned to the application with the next socket operation.

3. No response is received from the peer. In this case, the kernel might make several more attempts to make contact. TCP will usually give up in approximately 11 minutes if no response is solicited. The error ETIMEDOUT is returned with the next socket operation when this happens. Other errors such as EHOSTUNREACH can be returned if the network is unable to reach the host any longer, for example (this can happen because of bad routing tables or router failures).

Socket Options SO_LINGER (TCP)

The purpose of the SO_LINGER option is to control how the socket is shut down when the function close(2) is called. This option applies only to connection-oriented protocols such as TCP.

The default behavior of the kernel is to allow the close(2) function to return immediately to the caller. Any unsent TCP/IP data will be transmitted and delivered if possible, but no guarantee is made.

The SO_LINGER option can be enabled on the socket, to cause the application to block in the close(2) call until all final data is delivered to the remote end. Furthermore, this assures the caller that both ends have acknowledged a normal socket shutdown. Failing this, the indicated option timeout occurs and an error is returned to the calling application.

```
struct linger {
   int   l_onoff;
   int   l_linger;
};
```

example:

```
struct linger so_linger;
…
so_linger.l_onoff = TRUE;
so_linger.l_linger = 30;
z = setsockopt( s, SOL_SOCKET,  SO_LINGER,  &so_linger,  sizeof so_linger );
```

Nonblocking I/O

Connection lock

In the IPC scope the connection itself can be seen as a shared resource where we need to synchronize a send-receive channel.

- When an application issues one of the socket input APIs and there is no data to read, the API blocks and does not return until there is data to read. **(input lock)**

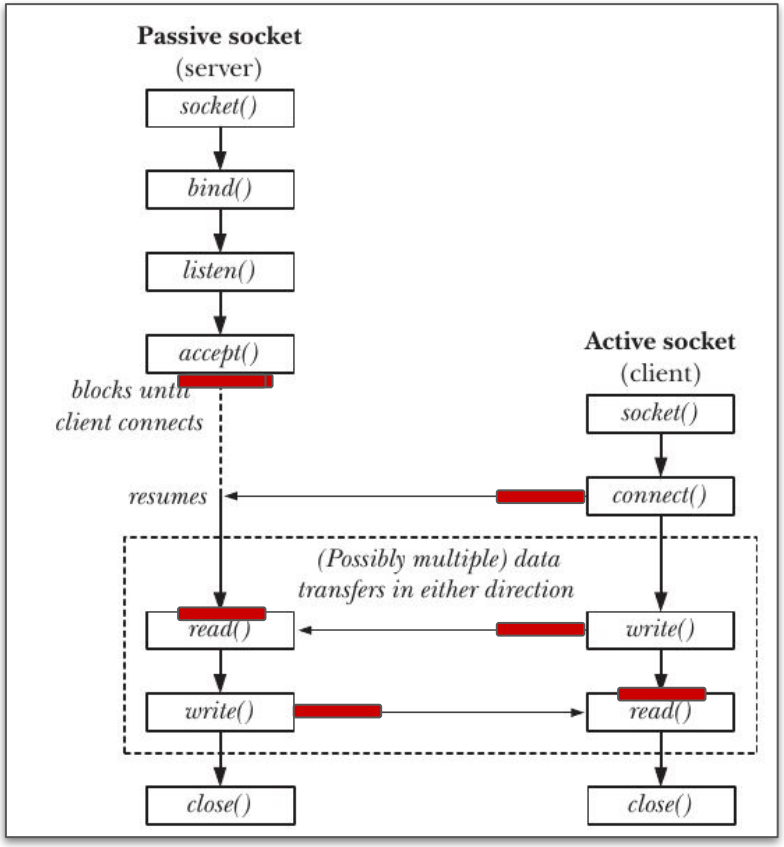
- Similarly, an application can block on a socket output API when data cannot be sent immediately. **(output lock)**

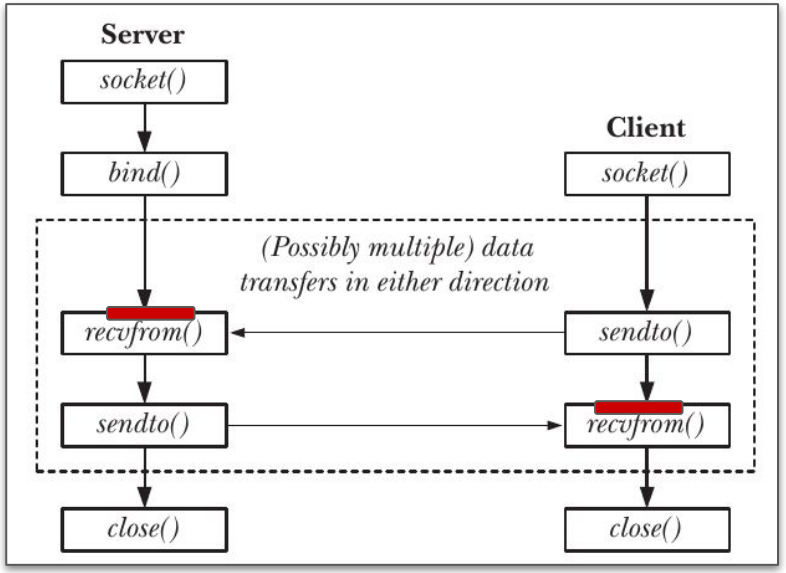
- Finally, in TCP also the **connect()** - **accept()** connection instance can block while waiting for connection establishment with the partner's programs.

Waiting for data or connection handling

The connection block several times, in TCP there are more waiting states: on the first connection handshaking, and on each read/write because the transaction need to be reliable so on each side we wait that all segments correctly passed through.

In UDP the situation is way simpler, we just need to wait for some input from the sender. The process still stops though !





Strategies to achieve a non-blocking connection

1) POLLING

- As we have seen in lab7 sockets are actually file descriptors and they also share the open options that are specific to a file. All sockets have the file descriptor option you can set: **O_NONBLOCK**
- Or you can use the **MSG_DONTWAIT** flag.

2) POLLING WITH DELAY

- You can use the SO_RCVTIMEO and SO_SNDTIMEO socket options to set timeouts for any socket operations, like so:

```
struct timeval timeout;
timeout.tv_sec = 10;
timeout.tv_usec = 0;

if (setsockopt (sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof timeout) < 0)
    error("setsockopt failed\n");

if (setsockopt (sockfd, SOL_SOCKET, SO_SNDTIMEO, &timeout, sizeof timeout) < 0)
    error("setsockopt failed\n");
```

Strategies to achieve a non-blocking connection

3) SELECT

Once the socket is open with **O_NONBLOCK**,

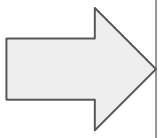
another option is to subscribe for file descriptor changes with a call to select( ), poll( ) or epoll( ) functions. The process will wait (with no CPU usage) until the kernel notice any file/socket change (i.e. data ready or any special conditions).

UDP Client Non Block

CODING EXAMPLE

CODING EXAMPLE

```c
// Driver code
int main() {
    int sockfd;
    char buffer[MAXLINE];
    char *hello = "Hello from client";
    struct sockaddr_in    servaddr;

    // Creating socket file descriptor
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    int flags = fcntl(sockfd,F_GETFL,0);
    assert(flags != -1);
    fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);

    memset(&servaddr, 0, sizeof(servaddr));

    // Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;

    int n, len;

    sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &servaddr,
            sizeof(servaddr));
    printf("Hello message sent.\n");
```



```c
// this should go through without waiting for reply
    do {

        n = recvfrom(sockfd, (char *)buffer, MAXLINE,
                0, (struct sockaddr *) &servaddr,
                &len);

        // n = recvfrom(sockfd, (char *)buffer, MAXLINE,
        //       MSG_DONTWAIT, (struct sockaddr *) &servaddr,
        //       &len);

    }
    while(n<0 && errno == EAGAIN);

    buffer[n] = '\0';
    printf("Server : %s\n", buffer);

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

1

2

Waiting on many connections in a single process

We presented examples using multi processing (fork) or multithreading (threads) methods to concurrently serve many connection to clients.

But sometimes a server that must share information between connected clients might find desirable to **keep the server contained within a single process**.

1. One process does not consume the same amount of system resources as many processes would.
2. The same process needs many connections (example: **Out-Of-Band data**)

For these reasons, it is necessary to consider a new server design philosophy.

Select usage:

The select function we saw permits you to block the execution of your server until there is something for the server to do on one or many file descriptors. It informs the caller that:

- ***read*** There is something to read from a file descriptor.
- ***write*** Writing to the file descriptor will not block the execution of the server program.
- ***except*** An exception has occurred on a file descriptor.

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

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

This function requires five input arguments:

1. The maximum number (n) of file descriptors to test. This value is at least the highest file descriptor number plus one, since descriptors start at zero.
2. The set of file descriptors (readfds) that are to be tested for read data.
3. The set of file descriptors (writefds) that are to be tested for writability.
4. The set of file descriptors (exceptfds) that are to be tested for exceptions (**Out-Of-Band data** for instance).
5. The pointer (timeout) to the timeout requirement, which is to be applied to this function call. This pointer may be NULL, indicating that there is no timeout (the function call may block forever).

The return results from the select function can be summarized as follows:

- A return value of -1 indicates that an error in the function call has occurred. The value of errno should be consulted for the nature of the error.
- A return value of zero indicates that a timeout has occurred without anything interesting happening.
- A return value greater than zero indicates the number of file descriptors where something of interest has occurred.

Manipulating File Descriptor Sets

The second, third, and fourth arguments of the select(2) function call require values of type fd_set, which might be new to you. This is an opaque data type, requiring that it be operated upon with macros provided for the purpose.

The following is a synopsis of the macros for your use:

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

Using the FD_ZERO Macro

This C macro is used to initialize a file descriptor set. Before you can register file descriptors (which includes sockets), you must initialize your set to all zero bits. To initialize a file descriptor set named read_socks and write_socks, you would write the following C language statements:

```
fd_set                                                                read_socks;
fd_set write_socks;

FD_ZERO(&read_socks);
FD_ZERO(&write_socks);
```

Manipulating File Descriptor Sets

Using the FD_SET Macro

After you have a file descriptor set initialized with the FD_ZERO macro, the next thing you want to accomplish is to register some file descriptors in it. This can be done with the FD_SET macro. The following example shows how a socket number c can be registered in a set named read_socks:

```
int     c;                                                  /*  Client  socket  */
fd_set          read_socks;                 /*      Read        set             */
…
FD_SET(c,&read_socks);
```

After calling FD_SET, a bit has registered interest in the corresponding file descriptor within the referenced set.

Using the FD_CLR Macro

This C macro undoes the effect of the FD_SET macro. Assuming a socket c once again, if the calling program wants to remove this descriptor from the set, it can perform the following:

```
int     c;                                                  /*  Client  socket  */
fd_set          read_socks;             /*          Read            set         */
…
FD_CLR(c,&read_socks);
```

The FD_CLR macro has the effect that it zeros the corresponding bit representing the file descriptor. Note that this macro differs from FD_ZERO in that it only clears one specific file descriptor within the set. The FD_ZERO macro, on the other hand, resets all bits to zero in the set.

Manipulating File Descriptor Sets

Testing File Descriptors with FD_ISSET Macro

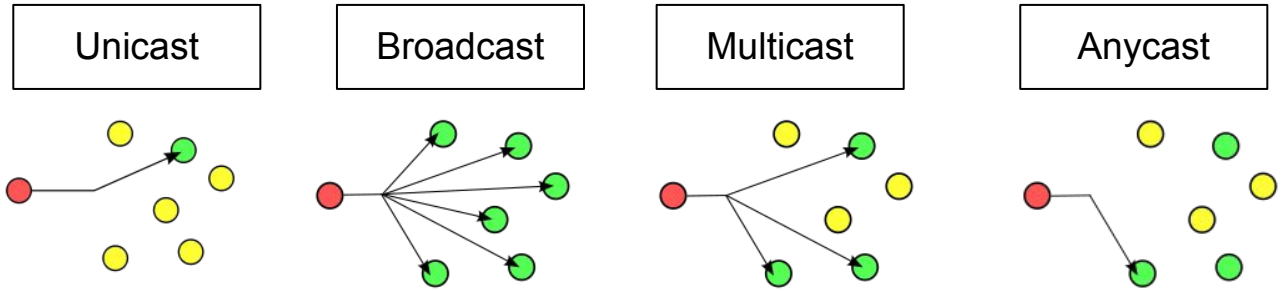
It is necessary, at times, to test to see if a particular file descriptor is present within a set (that is, to see if its corresponding bit has been set to one). To test whether socket c is set, you could write the following code:

```
int c;                                          /* Client socket */
fd_set read_socks;                              /* Read set */
…
if ( FD_ISSET(c,&read_socks) ) {
                /* Socket c is in the set */
                                                …
}                                               else {
                /* Socket c is not in the set */
                                                …
}
```

The if statement shown invokes the macro FD_ISSET to test if the socket c is present in the file descriptor set read_socks. If the test returns true, then the socket c does have its corresponding bit enabled within the set, and the first block of C code is executed. Otherwise, the socket c is not considered part of the set, and the else block of statements is executed instead.

ONE TO MANY CONNECTIONS

One to many



- Unicast delivers a message to a single specific node using a *one-to-one* association between a sender and destination: each destination address uniquely identifies a single receiver endpoint.

- Broadcast delivers a message to all nodes in the network using a *one-to-all* association; a single datagram from one sender is routed to all of the possibly multiple endpoints associated with the broadcast address. The network automatically replicates datagrams as needed to reach all the recipients within the scope of the broadcast, which is generally an entire network subnet.

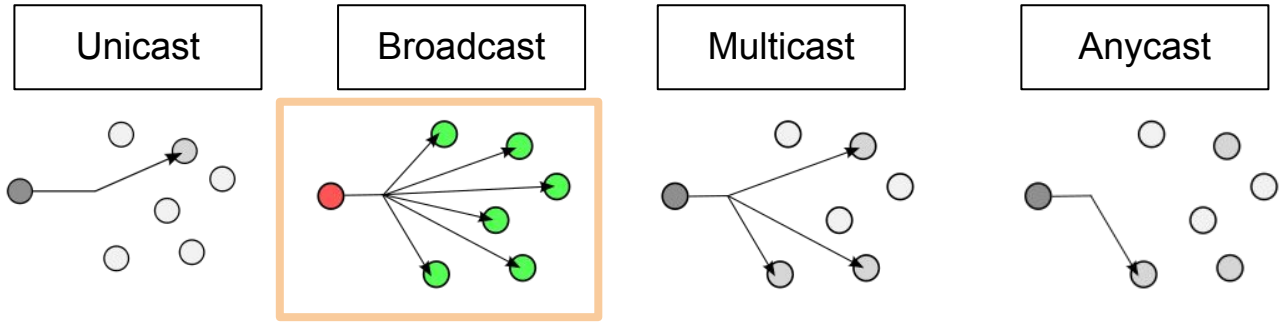
- Multicast delivers a message to a group of nodes that have expressed interest in receiving the message using a *one-to-many-of-many* or *many-to-many-of-many* association; datagrams are routed simultaneously in a single transmission to many recipients. Multicast differs from broadcast in that the destination address designates a subset, not necessarily all, of the accessible nodes.

- Anycast delivers a message to any one out of a group of nodes, typically the one nearest to the source using a *one-to-one-of-many* association where datagrams are routed to any single member of a group of potential receivers that are all identified by the same destination address. The routing algorithm selects the single receiver from the group based on which is the nearest according to some distance or cost measure.

IP BROADCAST

Communication would be inefficient if it always had to be accomplished between two individuals. Broadcasting, on the other hand, allows information to be disseminated to many recipients at once.

To use broadcasting, you must know about certain IP broadcast address conventions for IPv4. The convention used for a broadcast address in a subnet is that the **Host ID bits are all set to 1 bits**.



Getting the BROADCAST address:

```
#                                    ifconfig                                    eth0
eth0          Link      encap:Ethernet      HWaddr      00:A0:4B:06:F4:8D
              inet      addr:192.168.0.1    Bcast:192.168.0.255    Mask:255.255.255.0
          UP  BROADCAST RUNNING  PROMISC  MULTICAST  MTU:1500  Metric:1
              RX        packets:1955  errors:0  dropped:0  overruns:0  frame:31
              TX        packets:1064  errors:0  dropped:0  overruns:0  carrier:0
                                                collisions:0  txqueuelen:100
                                    Interrupt:9  Base      address:0xe400
#
```

IP BROADCAST

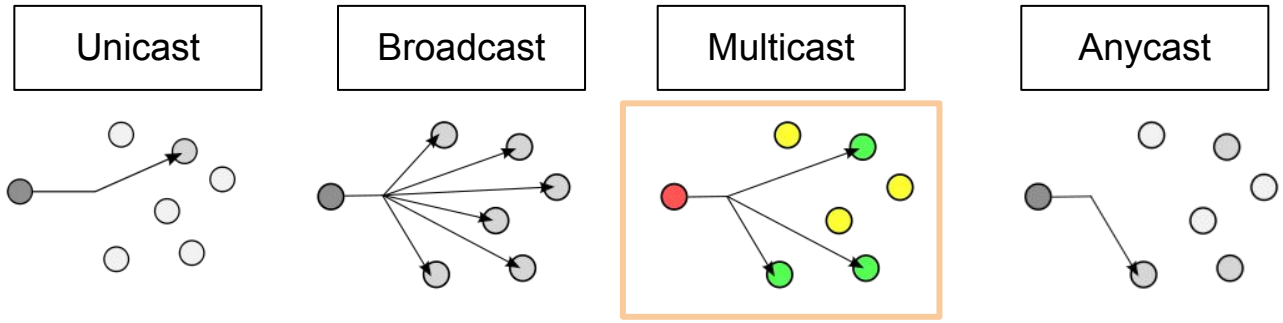
To send a broadcast UDP message just set the SO_BROADCAST option and send message to the broadcast address of the subnet you are connected to.

```
static int so_broadcast = TRUE;
static char *sv_addr = "127.0.0.1:*";
static char *bc_addr = "127.255.255.255:9097";

s = socket(AF_INET,SOCK_DGRAM,0);

setsockopt(s,  SOL_SOCKET,   SO_BROADCAST,   &so_broadcast,   sizeof so_broadcast);
```

IP MULTICAST

Multicast is known as a “**group communication**” where data transmission is addressed to a group of destination process simultaneously. It can be **one-to-many** or **many-to-many** distribution of messages.

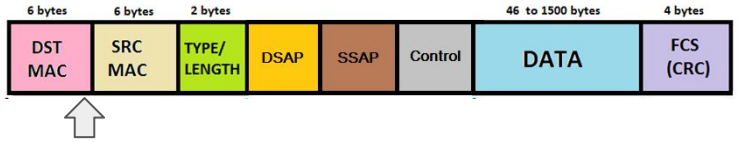
NOTE: Multicast should not be confused with physical layer point-to-multipoint communication that still holds the connection as we will see.



Multicast uses network infrastructure efficiently by requiring the source to send a packet only once, even if it needs to be delivered to a large number of receivers.

How it works

Data Link Layer:



On Ethernet Networks (IEEE 802.3) the ethernet frames with a value of 1 in the **least-significant bit of the first octet of the destination address** are treated as multicast frames and are flooded to all points on the network.

Modern Ethernet controllers filter received packets to reduce CPU load, by looking up the hash of a multicast destination address in a table, initialized by software, which controls whether a multicast packet is dropped or fully received.

IPv4 Layer:

Over an IP network the destination nodes send a Internet Group Management Protocol (IGMP) message to join and leave the group. The nodes in the network take care of replicating the packet to reach multiple receivers only when necessary.

The most common transport layer protocol to use multicast addressing is UDP. By its nature, UDP is not reliable and may be lost or delivered out of order, on the other hand **a TCP connection can not hold multicast.**

UDP/IP Multicast Receiver

CODING EXAMPLE

CODING EXAMPLE

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define PORT 4444
#define GROUP "225.0.0.37"

/* Maximum dimension of the receiver buffer */
#define BUFSIZE 256

/* Receiver main program. No arguments are passed in the command line. */
int main(int argc, char *argv[])
{
 struct sockaddr_in addr;
 int sd, nbytes,addrLen;
 struct ip_mreq mreq;
 char msgBuf[BUFSIZE];

/* Create a UDP socket */
 if ((sd=socket(AF_INET, SOCK_DGRAM, 0)) < 0)
 {
   perror("socket");
   exit(0);
 }

 int optval = 1; // true
 if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &optval,
     sizeof(optval)) == -1)
  {
     perror("options");
     exit(1);
  }
/* Set up receiver address. Same as in the TCP/IP example. */
 memset(&addr, 0,sizeof(addr));
 addr.sin_family = AF_INET;
 addr.sin_addr.s_addr = INADDR_ANY;
 addr.sin_port = htons(PORT);
```

```c
/* Bind to receiver address */
 if (bind(sd,(struct sockaddr *) &addr, sizeof(addr)) < 0)
 {
   perror("bind");
   exit(0);
 }
/* Use setsockopt() to request that the receiver join a multicast group */
 mreq.imr_multiaddr.s_addr = inet_addr(GROUP);
 mreq.imr_interface.s_addr = INADDR_ANY;
 if (setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq)) < 0)
 {
   perror("setsockopt");
   exit(0);
 }
/* Now the receiver belongs to the multicast group:
  start accepting datagrams in a loop */
 for(;;)
 {
   addrLen = sizeof(addr);
/* Receive the datagram. The sender address is returned in addr */
   if ((nbytes = recvfrom(sd, msgBuf, BUFSIZE, 0,
    (struct sockaddr *) &addr,&addrLen)) < 0)
   {
     perror("recvfrom");
     exit(0);
   }
/* Insert terminator */
   msgBuf[nbytes] = 0;
   printf("%s\n", msgBuf);
 }
  return 0;
}
```

UDP/IP Multicast Sender

CODING EXAMPLE CODING EXAMPLE

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define PORT 4444
#define GROUP "225.0.0.37"

/* Sender main program: get the string from the command argument */
int main(int argc, char *argv[]) {
 struct sockaddr_in addr;
 int sd;
 char *message;
/* Get message string */
 if(argc < 2)
 {
   printf("Usage: sendUdp <message>\n");
   exit(0);
 }
 message = argv[1];
/* Create the socket. The second argument specifies that
  this is an UDP socket */
 if ((sd = socket(AF_INET,SOCK_DGRAM,0)) < 0)
 {
   perror("socket");
   exit(0);
 }
/* Set up destination address: same as TCP/IP example */
 memset(&addr,0,sizeof(addr));
 addr.sin_family = AF_INET;
 addr.sin_addr.s_addr = inet_addr(GROUP);
 addr.sin_port=htons(PORT);
```

```c
/* Send the message */
 if (sendto(sd,message,strlen(message),0,
      (struct sockaddr *) &addr, sizeof(addr)) < 0)
 {
   perror("sendto");
   exit(0);
 }
/* Close the socket */
 close(sd);
}
```