**1. Sockets**

Sockets is one the inter process mechanism that allows processes on same or different machines to communicate with each other. It is similar to a file descriptor.

*1.2 Applications*

A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

*1.3 Types of socket*

There are two types of Internet sockets available to users, Stream socket and Datagram socket.

1. Stream Socket(SOCK\_STREAM)

A stream socket uses the Transmission Control Protocol (TCP) for sending messages. TCP provides an ordered and reliable connection between two hosts. This means that for every message sent, TCP guarantees that the message will arrive at the host in the correct order. This is achieved at the transport layer so that the application need not bother about it. Stream sockets are also called as connection oriented sockets.

2. Datagram Socket(SOCK\_DGRAM)

A datagram socket uses the User Datagram Protocol (UDP) for sending messages. UDP is a much simpler protocol as it does not provide any of the delivery guarantees that TCP does. Messages, called datagrams, can be sent to another host without requiring any prior communication or a connection having been established. As such, using UDP can lead to lost messages or messages being received out of order. It is assumed that the application can tolerate an occasional lost message or that the application will handle the issue of retransmission. Datagram sockets are also called as connectionless sockets

*1.4 Addressing*

Every host on the internet is identified by its address called as Internet Protocol (IP) Address and the process within the host is identified by the port numbers.

*1.4.1 IP Address*

An IP version 4 (IPV4) address is of 32 bits interpreted as four octets. IP version 6 (IPV6) is the recent version of IP protocol and the address is of 128 bits and represented as 8 hextets.

*1.4.2 Port Numbers*

This address is used to identify the communicating processes. It is 16 bit number and is local for the connection.

*1.4.3 Byte order*

There are two types of byte ordering - Network Byte order same as Big- Endian and Host Byte Order same as Little – Endian. To have portability between different machines across the internet there are four functions to convert the byte ordering. The conversion to network byte order is done when the data goes out on the wire and convert to host byte order as data come in off the wire.

htons() host to network short

htonl() host to network long

ntohs() network to host short

ntohl() network to host long

*1.5 Structures*

The structsockaddr holds the socket address information for many types of sockets defined in the header file <sys/socket.h>. The members of this structures are described below

structsockaddr

{

unsigned short sa\_family; // address family, AF\_xxx

charsa\_data[14]; // 14 bytes of protocol address

};

sa\_family can be a variety of things, AF\_INET (IPv4) or AF\_INET6 (IPv6). sa\_data contains a destination address and port number for the socket.

Usually to deal with structsockaddr, programmers created a parallel structure: structsockaddr\_in (“in” for “Internet”) to be used with IPv4.A pointer to a structsockaddr\_in can be cast to a pointer to a structsockaddr and vice-versa.

The structure sockaddr\_in is described below:

structsockaddr\_in

{ shortintsin\_family; // Address family, AF\_INET

unsigned short intsin\_port; // Port number

structin\_addrsin\_addr; // Internet address

unsigned char sin\_zero[8]; // Same size as structsockaddr

};

This structure makes it easy to reference elements of the socket address. Note that sin\_zero (which is included to pad the structure to the length of a structsockaddr) should be set to all zeros with the function memset().sin\_family corresponds to sa\_family in a structsockaddr and should be set to “AF\_INET”. The sin\_port must be in Network Byte Order.

The structure in\_addr is used to refer four byte IP address in Network Byte Order.

structin\_addr {

uint32\_t s\_addr; // 32-bit int (4 bytes)

};

*1.6 Address Conversion Functions*

There are couple of functions that help us in converting the dotted decimal form of IP addresses to the required structin\_addr format and vice versa.

1. inet\_pton()- converts an IP address in numbers-and-dots notation into structin\_addr. The pton stands for “presentation to network”.

2. inet\_ntop()- converts the address that is in structin\_addr format to numbers and dots notation. The ntop stands for “network to presentation”.

*1.7 Essential system calls for Inter Process communication using sockets*

In any process communication, one process acts as a client and other as the server. This section discusses about the system calls used by the client and servers to establish the connection and to transfer the data.

*1.7.1 Steps involved in establishing a socket on client side.*

* Create a socket with the **socket()** system call.
* Connect the socket to the address of the server using the **connect()** system call.
* Send and receive data. There are a number of ways to do this, but the simplest way is to use the **read()** and **write()** system calls.

1. Socket( )- to get the socket descriptor. This system call returns a

Synopsis

#include<sys/types.h>

#include<sys/socket.h>

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

Description

* domain is PF\_INET or PF\_INET6,
* type is SOCK\_STREAM or SOCK\_DGRAM,
* Protocol can be set to 0 to choose the proper protocol for the given type.

Return Value

This system call returns a socket descriptor(sockfd)

2. Connect( )- To establish the connection to server’s socket.

Synopsis

#include<sys/types.h>

#include<sys/socket.h>

int connect(intsockfd, structsockaddr \*serv\_addr, intaddrlen);

Description

* sockfd is socket descriptor, as returned by the socket() call,
* serv\_addr is a structsockaddr containing the destination port and IP address,
* addrlen is the length in bytes of the server address structure.

Return Value

The system call returns -1 on error and 0 on success.

3. send() and recv()

These two functions are for communicating over stream sockets or connected sockets. For unconnected datagram sockets, sendto() and recvfrom() are used.

Send()

Synopsis

#include<sys/types.h>

#include<sys/socket.h>

int send(intsockfd, const void \*msg, intlen, int flags);

Description

* sockfd is the socket descriptor on which data has to be sent (whether it's the one returned by socket() or the one got with accept().)
* msg is a pointer to the data that has to be sent,
* len is the length of that data in bytes and

Flag is set to 0.

Return value

The system call returns number of bytes actually sent out.

Recv()

Synopsis

#include<sys/types.h>

#include<sys/socket.h>

intrecv(intsockfd, void \*buf, intlen, int flags);

Description

* sockfd is the socket descriptor to read the data from.
* Buf is the buffer to read the information into,
* len is the length of the buffer and
* Flag is set to 0.

Return value

The system call returns number of bytes actually read into the buffer, or -1 on error.

*1.7.2 Steps involved in establishing a socket on server side.*

* Create a socket with the **socket()** system call.
* Bind the socket to an address using the **bind()** system call.
* Listen for connections with the **listen()** system call.
* Accept a connection with the **accept()** system call. This call typically blocks the connection until a client connects with the server.
* Send and receive data using the **send()** and recv**()** system calls.

1. bind()-binds the server process to the address

Once the socket is created at the server side, it should get associated with the port number on the local machine. The port number is used by the kernel to match an incoming packet to a certain processe’s socket descriptor.

Synopsis

#include<sys/types.h>

#include<sys/socket.h>

int bind(intsockfd, structsockaddr \*my\_addr, intaddrlen);

Description

* sockfd is the socket file descriptor returned by socket().
* my\_addr is a pointer to a structsockaddr that contains information about server’s address, namely, port and IP address.
* addrlen is the length in bytes of that address.

Return Value

bind() also returns -1 on error and sets errno to the error's value.

2. listen( )- Block until a connection arrives.

This system call makes the process wait for incoming connections.

Synopsis

#include<sys/types.h>

#include<sys/socket.h>

int listen(intsockfd, int backlog);

Description

* sockfd is the usual socket file descriptor from the socket() system call.
* backlog is the number of connections allowed on the incoming queue. The incoming connections are wait the queue until they are accepted by accept() call. It is the limit on how many can queue up. Most systems silently limit this number to about 20.

Return Value

listen() returns -1 and sets errno on error.

3. Accept( ) – To accept the incoming connections.

This system call creats a new file descriptor to use for this connection.

Synopsis

#include<sys/types.h>

#include<sys/socket.h>

int accept(intsockfd, structsockaddr \*addr, socklen\_t \*addrlen);

Description

* sockfd is the listen()ing socket descriptor.
* addr will usually be a pointer to a local structure.
* addrlen is a local integer variable that should be set to sizeof(structsockaddr\_storage) before its address is passed to accept().

Return Value

On success returns a new socket descriptor and returns -1 and sets errno if an error occurs.

*1.8 Flow diagram for client server interaction*

TCP Server

shutdown( )

close()

read()

close()

socket()

connect()

write()

read()

process request

write()

new socket

socket()

bind()

listen()

accept()

TCP Client

Blocks until connection from client

connectionestblishment

data request

**\**

data reply

connection termination

**2 Distance vector**

Routing algorithm is a part of network layer software which is responsible for deciding which outputline an incoming packet should be transmitted on. If the subnet uses datagram internally ,this decision must be made anew for every arriving data packet since the best route may have changed since lasttime. If the subnet uses virtual circuits internally, routing decisions are made only when a newestablished route is being set up. The latter case is sometimes called session routing, because a route remains in force for an entire user session.

Routing algorithms can be grouped into two major classes: adaptive and nonadaptive.Non adaptive algorithms do not base their routing decisions on measurement or estimates ofcurrent traffic and topology. Instead, the choice of route to use to get from *I* to *J* (for all *I* and *J*) is compute in advance,offline, and downloaded to the routers when the network ids booted. This procedure is sometime called static routing.

Adaptive algorithms, in contrast, change their routing decisions to reflect changes in thetopology, andusually the traffic as well. Adaptive algorithms differ in where they get information(e.g., locally,from adjacent routers, or from all routers), when they change the routes (e.g., every*T* sec, when the load changes, or when the topology changes), and what metric is used foroptimization (e.g., distance,number of hops, or estimated transit time).

Two algorithms in particular, distance vector routing and link state routing are the mostpopular.Distance vector routing algorithms operate by having each router maintain a table (i.e.,vector) givingthe best known distance to each destination and which line to get there. Thesetables are updated byexchanging information with the neighbours.

The distance vector routing algorithm is sometimes called by other names, including thedistributedBellman-Ford routing algorithm and the Ford-Fulkerson algorithm, after theresearchers whodeveloped it (Bellman, 1957; and Ford and Fulkerson, 1962). It was the originalARPANET routingalgorithm and was also used in the Internet under the RIP and in earlyversions of DECnet andNovell’s IPX. AppleTalk and Cisco routers use improved distance vectorprotocols.

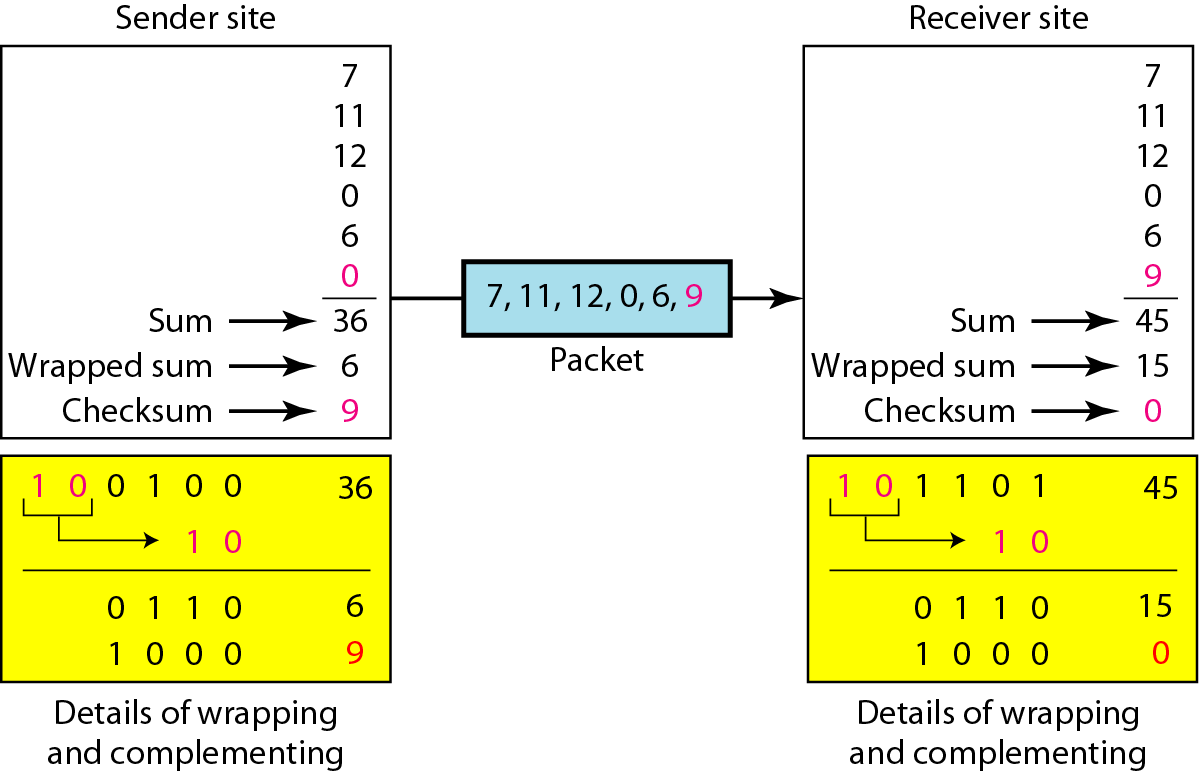
In distance vector routing, each router maintains a routing table indexed by, and containingone entryfor, each router in subnet. This entry contains two parts: the preferred out going line touse for thatdestination, and an estimate of the time or distance to that destination. The metricused might benumber of hops, time delay in milliseconds, total number of packets queued alongthe path, orsomething similar.

The router is assumed to know the “distance” to each of its neighbor. If the metric is hops, thedistance is just one hop. If the metric is queue length, the router simply examines each queue. Ifthemetric is delay, the router can measure it directly with special ECHO packets has the receiverjusttime stamps and sends back as fast as possible.

**3. Internet checksum**

## A checksum is a count of the number of bits in a transmission unit that is included with the unit so that the receiver can check to see whether the same number of bits arrived.

The checksum is used in the Internet by several protocols although not at the data link layer.



**Sender side:**

1. The message is divided into 16-bit words.

2. The value of the checksum word is set to 0.

3. All words including the checksum are added using one’s complement addition.

4. The sum is complemented and becomes the checksum.

5. The checksum is sent with the data.

**Receiver side:**

1. The message (including checksum) is divided into 16-bit words.

2. All words are added using one’s complement addition.

3. The sum is complemented and becomes the new checksum.

4. If the value of checksum is 0, the message is accepted; otherwise, it is rejected.

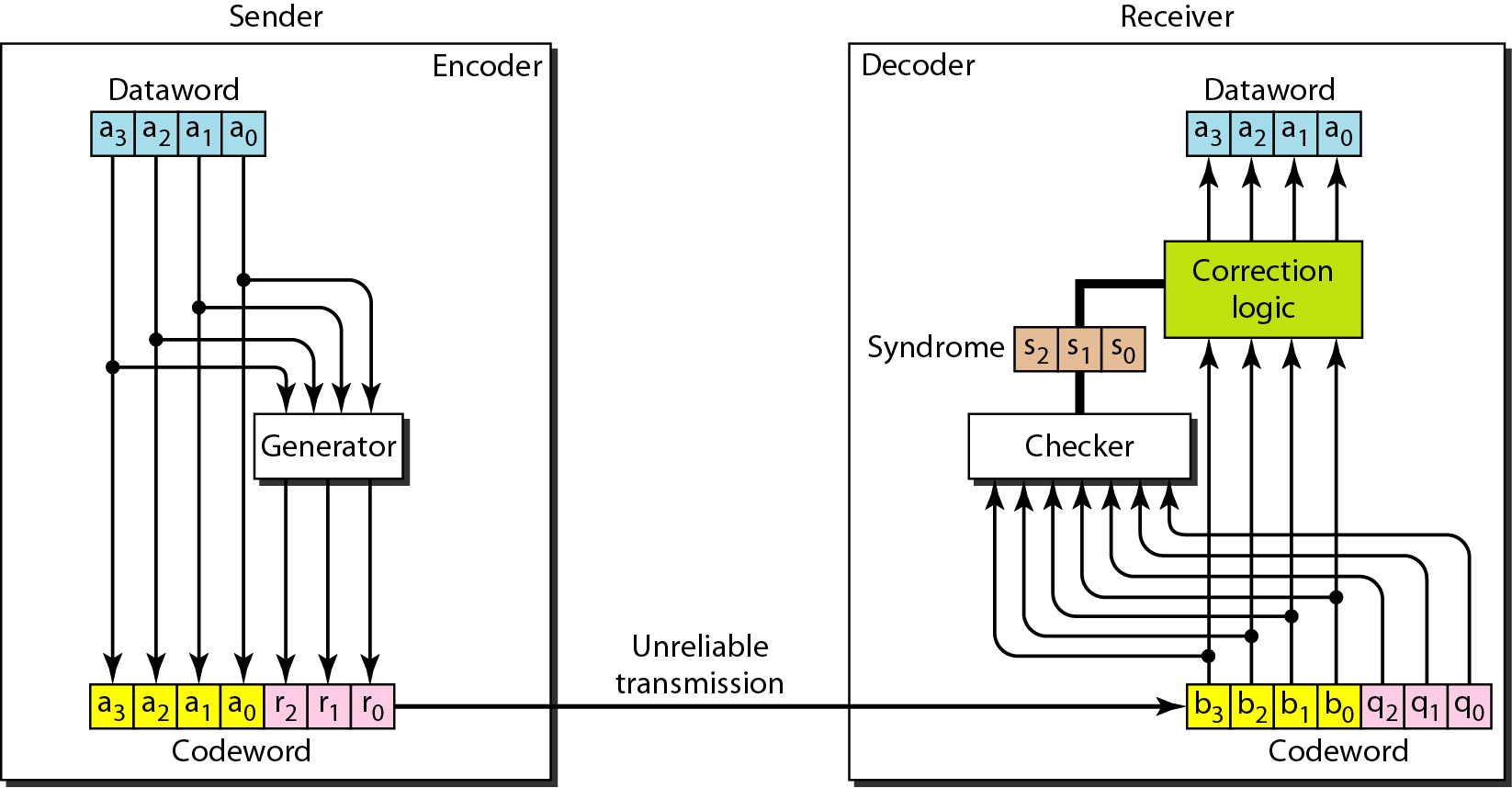
**4. Hamming code**

Hamming code is a set of error-correction codes that are used to **detect and correct the errors** that occurs during data transmission.

*4.1 Redundant bits*

Redundant bits are extra binary bits that are generated and added to the information-carrying bits of data transfer to ensure that no bits were lost during the data transfer.  
The number of redundant bits can be calculated using the formula: 2r>= m+r+1, where r represents redundant bits, m represents data bits. For example if m=4 then according to formula the number of redundant bits to be added to data bits is 3.

*4.2 Structure of encoder and decoder for hamming code*



In the encoding process the data bits are appended with the redundant bits(ro,r1 and r2). The value of redundant bits are determined and transmitted. At the receiver side the codeword is passes through the checker to determine the syndrome bits(s0,s1 and s2). If the bits are zero it indicates there is no error in the received codeword, otherwise the binary value of the syndrome bits represents the position of error in the codeword.

*4.3 Determining the redundant bits*

To establish the relationship between the redundant bits and the data bits, the position of the redundant bits must be determined. The redundant bits are placed at the positions corresponding to power of 2(20,21,22…). The value of these redundant bits is determined by performing addition modulo 2 of various position bits.

Consider an example: m=4 (1011) hence r=3(r2, r1,r0). The resulting codeword is of 7 bits.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit Positions | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| Power of 2 positions |  |  |  | 22 |  | 21 | 20 |
| Positions of redundant bits |  |  |  | r2 |  | r1 | r0 |
| Position of data bits | d3 | d2 | d1 |  | d0 |  |  |
| Codeword | d3 | d2 | d1 | r2 | d0 | r1 | r0 |
| For example | 1 | 0 | 1 | r2 | 1 | r1 | r0 |

The value of r1 is determined by performing arithmetic addition of the bits whose binary representation of the position has least significant bit 1(bits at position 3,5, and 7). Similarly r2 is determined with bit position having 1 at second position from least significant bit(3,6,7) and r3 is determined with numbers having bit position having 1 at third position from least significant bit(5,6,7 )

r3= d1+d2+d3 mod 2

= 1+0+1

=0

r2= d0+d2+d3 mod 2

= 1+0+1

=0

r1= d0+d1+d3 mod 2

= 1+1+1

=1

The transmitted codeword is 1011 001

Let us assume sixth bit is in error. The received codeword is 1111 001. To detect error, the codeword is sent to the checker, the checker computes the syndrome bits same as redundant bits at the sender side.

s3= d1+d2+d3+r3 mod 2

= 1+1+1+0

=1

s2= d0+d2+d3+r2 mod 2

= 1+1+1+0

=1

s1= d0+d1+d3+r1 mod 2

= 1+1+1+1

=0

Since the syndrome is 110, the error is in sixth bit is in error. If syndrome bits are 000 then the codeword is error free.

**5 Congestion control algorithms**

The congesting control algorithms are basically divided into two groups: open loop and closed loop. Open loop solutions attempt to solve the problem by good design, in essence, to make sure it does not occur in the first place. Once the system is up and running, midcourse corrections are not made. Open loop algorithms are further divided into ones that act at source versus ones that act at the destination.

In contrast, closed loop solutions are based on the concept of a feedback loop if there is any congestion. Closed loop algorithms are also divided into two sub categories: explicit feedback and implicit feedback. In explicit feedback algorithms, packets are sent back from the point of congestion to warn the source. In implicit algorithm, the source deduces the existence of congestion by making local observation, such as the time needed for acknowledgment to come back.

The presence of congestion means that the load is (temporarily) greater than the resources (in part of the system) can handle. For subnets that use virtual circuits internally, these methods can be used at the network layer.

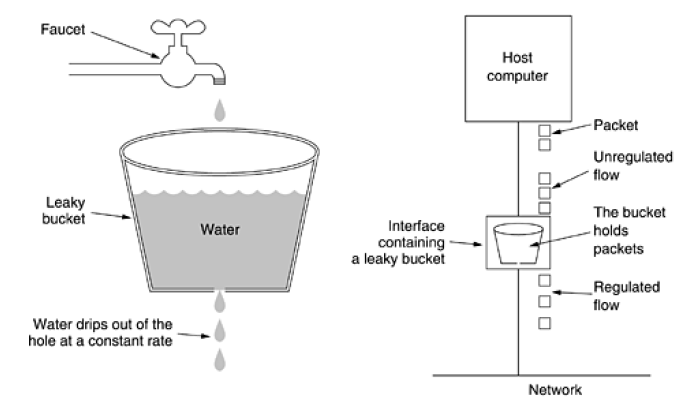
Another open loop method to help manage congestion is forcing the packet to be transmitted at a more predictable rate. This approach to congestion management is widely used in ATM networks and is called traffic shaping.

*5.1 Leaky bucket algorithm.*

Each host is connected to the network by an interface containing a leaky bucket, that is, a finite internal queue. If a packet arrives at the queue when it is full, the packet is discarded. In other words, if one or more process are already queued, the new packet is unceremoniously discarded. This arrangement can be built into the hardware interface or simulate d by the host operating system. In fact it is nothing other than a single server queuing system with constant service time.

The host is allowed to put one packet per clock tick onto the network. This mechanism turns an uneven flow of packet from the user process inside the host into an even flow of packet onto the network, smoothing out bursts and greatly reducing the chances of congestion.

The leaky-bucket implementation is used to control the rate at which traffic is sent to the network. A leaky bucket provides a mechanism by which bursty traffic can be shaped to present a steady stream of traffic to the network, as opposed to traffic with erratic bursts of low-volume and high-volume flows.The algorithm can be conceptually understood as follows:

****

* Consider a bucket with a hole in the bottom.
* If packets arrive, they are placed into the bucket. If the bucket is full, packets are discarded.
* Packets in the bucket are sent at a **constant rate**, equivalent to the size of the hole in the

**6. Multicast routing mechanism**

Multicast communications refers to one-to-many or many-to many communications. If there is lot of information, that should be transmitted to various hosts over an internet, then Multicast is the solution.

*6.1 Multicast Address*

In multicasting, "Class D Address" is used. Every IP datagram whose destination address starts with "1110" is an IP Multicast datagram.The remaining 28 bits identify the multicast "group" the datagram is sent to.

*6.2 Sending multicast datagram*

Multicast traffic is handled at the transport layer with UDP, as TCP provides point−to−point connections and is not feasible for multicast traffic**.**In principle, an application just needs to open a UDP socket and fill with a class D multicast address the destination address where it wants to send data to. However, there are some operations that a sending process must be able to control.

*6.3 Multicast Programming*

Several extensions to the programming API are needed in order to support multicast. All of them are handled via two system calls:

setsockopt() (used to pass information to the kernel) and

getsockopt() (to retrieve information regarded multicast behavior).

The addition consists on a new set of options (multicast options) that are passed to these system calls, that the kernel must understand. The following are the setsockopt()/getsockopt() function prototypes:

intgetsockopt(int s, int level, intoptname, void\* optval, int\* optlen);

intsetsockopt(int s, int level, intoptname, const void\* optval, intoptlen);

* The first parameter, s, is the socket the system call applies to. For multicasting, it must be a socket of the family AF\_INET and its type may be either SOCK\_DGRAM or SOCK\_RAW. The most common use is with SOCK\_DGRAM sockets.
* The second one, level, identifies the layer that is to handle the option, message or query. So, SOL\_SOCKET is for the socket layer, IPPROTO\_IP for the IP layer, etc... For multicast programming, level will always be IPPROTO\_IP.optname identifies the option we are setting/getting. Its value (either supplied by the program or returned by the kernel) is optval. The optnames involved in multicast programming are the following:

setsockopt() getsockopt()

IP\_MULTICAST\_LOOP yes yes

IP\_MULTICAST\_TTL yes yes

IP\_MULTICAST\_IF yes yes

IP\_ADD\_MEMBERSHIP yes no

IP\_DROP\_MEMBERSHIP yes no

IP\_ADD\_MEMBERSHIP.

It is necessary to tell the kernel which multicast groups the receivers are interested in. If no process is interested in a group, packets destined to it that arrive to the host are discarded. In order to inform the kernel of the interests and, thus, become a member of that group, it is required to fill a ip\_mreq structure which is passed later to the kernel in the optval field of the setsockopt() system call.

The ip\_mreq structure (taken from /usr/include/linux/in.h) has the following members:

structip\_mreq { structin\_addrimr\_multiaddr; /\* IP multicast address of group \*/

structin\_addrimr\_interface; /\* local IP address of interface \*/

};

The first member, imr\_multiaddr, holds the group address that receiver want to join. The memberships are also associated with interfaces, not just groups. This is the reason to provide a value for the second member: imr\_interface. This way, if it is a multihomed host, then it can join the same group in several interfaces. It is always possible to fill this last member with the wildcard address (INADDR\_ANY) and then the kernel will deal with the task of choosing the interface. With this structure filled (say you defined it as: structip\_mreqmreq;) a call tosetsockopt() is:

setsockopt (socket, IPPROTO\_IP, IP\_ADD\_MEMBERSHIP, &mreq, sizeof(mreq));

Notice that a host can join several groups to the same socket, not just one. The limit to this is IP\_MAX\_MEMBERSHIPS and, as of version 2.0.33, it has the value of 20.

IP\_DROP\_MEMBERSHIP.

The process is quite similar to joining a group:

structip\_mreqmreq;

setsockopt (socket, IPPROTO\_IP, IP\_DROP\_MEMBERSHIP, &mreq, sizeof(mreq));

wheremreq is the same structure with the same data used when joining the group.

If the imr\_interface member is filled with INADDR\_ANY, the first matching group is dropped. When a socket is closed, all memberships associated with it are dropped by the kernel. The same occurs if the process that opened the socket is killed. Both ADD\_MEMBERSHIP and DROP\_MEMBERSHIP are nonblocking operations. They should return immediately indicating either success or failure.

**7. Encryption and Decryption**

The message to be sent through an unreliable medium is known as plaintext. For security purpose it is encrypted before sending over the medium. The encrypted message is known as ciphertext, which is received at the other end of the medium and decrypted to get back the original plaintext message. There are various cryptography algorithms and can be divided into two broad categorize - Symmetric key cryptography and Public key cryptography. The following figure shows a simple cryptography model

ENCRYPTION

ALGORITHM

DECRYPTION

ALGORITHM

ENCRYPTION KEY

DECRYPTION KEY

*7.1 Public key Cryptography*

In public key cryptography, there are two keys: a private key and a public key. The public key is announced to the public, where as the private key is kept by the receiver. The sender uses the public key of the receiver for encryption and the receiver uses his private key for decryption.

*7.2 RSA*

The most popular public-key algorithm is the RSA (named after their inventors Rivest, Shamir and Adleman)

Key features of the RSA algorithm are given below:

1. Public key algorithm that performs encryption as well as decryption based on number theory
2. Variable key length; long for enhanced security and short for efficiency (typical 512 bytes)
3. Variable block size, smaller than the key length
4. The private key is a pair of numbers (d, n) and the public key is also a pair of numbers (e, n)

**8. Key exchange algorithms**

To preserve data confidentiality during transmission, secure file transfer protocols like FTPS, HTTPS, and SFTP have to encrypt the data through what is known as symmetric encryption. This kind of encryption requires the two communicating parties to have a shared key in order for them to encrypt and decrypt messages. In the real world, the two communicating parties would likely be geographically separated by long distances. The key can't just be sent through ordinary methods because anyone who gets hold of it would then be able to decrypt all the files that the two parties would be sending to one another. A key exchange method should be easy to use, secure, and highly scalable.

*8.1 Diffiehallman key exchange algorithm*

Diffie-Hellman is a way of establishing a shared secret between two endpoints (parties). Consider following example to understand the algorithm. Let Alice and Bob wants to communicate with each other without John knowing their communication. To start, Alice and Bob decide publicly (John will also get a copy) on two prime numbers, ***g*** and ***n***. Generally ***g*** is a small prime number and ***n*** is quite large, usually 2000 or more commonly 4000 bits long. So now Alice, Bob and John all know these numbers.

Alice now decides secretly on another number,***a***. and Bob decides secretly on a number,***b***. Neither Alice nor Bob send these numbers, they are kept to themselves. Alice performs a calculation, ***g^a mod n***, let this be A, since it comes from ***a***. Bob then performs***g^b mod n*** let this be ***B***.

Alice sends Bob,***A***, and Bob sends Alice, ***B***. Note John now has 4 numbers,***A, B, g*** and ***n*** but not ***a*** or***b***. Alice takes Bob’s ***B*** and performs***B^a mod n***. Similarly, Bob takes Alice’s ***A*** and performs ***A^b mod n***. This results in the same number i.e. ***B^a mod n = A^b mod n***. They now have a shared number. John can’t figure out what these numbers are from the numbers he’s got.

**1. Implement a client and server communication using sockets programming.**

Algorithm (Client Side)

1. Start.
2. Create a socket using socket() system call.
3. Connect the socket to the address of the server using connect() system call.
4. Send the filename of required file using send() system call.
5. Read the contents of the file sent by server by recv() system call.
6. Stop.

Algorithm (Server Side)

1. Start.
2. Create a socket using socket() system call.
3. Bind the socket to an address using bind() system call.
4. Listen to the connection using listen() system call.
5. accept connection using accept()
6. Receive filename and transfer contents of file with client.
7. Stop.

**PROGRAM**

/\*Server\*/

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<sys/stat.h>

#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include<fcntl.h>

#include <arpa/inet.h>

int main()

{

intcont,create\_socket,new\_socket,addrlen,fd;

intbufsize = 1024;

char \*buffer = malloc(bufsize);

charfname[256];

structsockaddr\_in address;

if ((create\_socket = socket(AF\_INET,SOCK\_STREAM,0)) > 0)

printf("The socket was created\n");

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(15001);

if (bind(create\_socket,(structsockaddr \*)&address,sizeof(address)) == 0)

printf("Binding Socket\n");

listen(create\_socket,3);

addrlen = sizeof(structsockaddr\_in);

new\_socket = accept(create\_socket,(structsockaddr \*)&address,&addrlen);

if (new\_socket> 0)

{

printf("The Client %s is Connected...\n",

inet\_ntoa(address.sin\_addr) );

}

recv(new\_socket,fname, 255,0);

printf("A request for filename %s Received..\n", fname);

if ((fd=open(fname, O\_RDONLY))<0)

{perror("File Open Failed"); exit(0);}

while((cont=read(fd, buffer, bufsize))>0) {

send(new\_socket,buffer,cont,0);

}

printf("Request Completed\n");

close(new\_socket);

return close(create\_socket);

}

/\*Client\*/

#include<sys/socket.h>

#include<sys/types.h>

#include<netinet/in.h>

#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>

int main(intargc,char \*argv[])

{

intcreate\_socket;

intbufsize = 1024, cont;

char \*buffer = malloc(bufsize);

charfname[256];

structsockaddr\_in address;

if ((create\_socket = socket(AF\_INET,SOCK\_STREAM,0)) > 0)

printf("The Socket was created\n");

address.sin\_family = AF\_INET;

address.sin\_port = htons(15001);

inet\_pton(AF\_INET,argv[1],&address.sin\_addr);

if (connect(create\_socket,(structsockaddr \*) &address, sizeof(address)) == 0)

printf("The connection was accepted with the server %s...\n",argv[1]);

printf("Enter The Filename to Request : "); scanf("%s",fname);

send(create\_socket, fname, sizeof(fname), 0);

printf("Request Accepted... Receiving File...\n\n");

printf("The contents of file are...\n\n");

while((cont=recv(create\_socket, buffer, bufsize, 0))>0) {

write(1, buffer, cont);

}

printf("\nEOF\n");

return close(create\_socket);

}

OUTPUT

SERVER

exam@dell:~$ gcc -o server server.c

exam@dell:~$ ./server

The socket was created

Binding Socket

The Client 127.0.0.1 is Connected...

A request for filename test.txt Received..

Request Completed

CLIENT

exam@dell:~$ gcc -o client client.c

exam@dell:~$ ./client 127.0.0.1

The Socket was created

The connection was accepted with the server 127.0.0.1...

Enter The Filename to Request : test.txt

Request Accepted... Receiving File...

The contents of file are...

hello

EOF