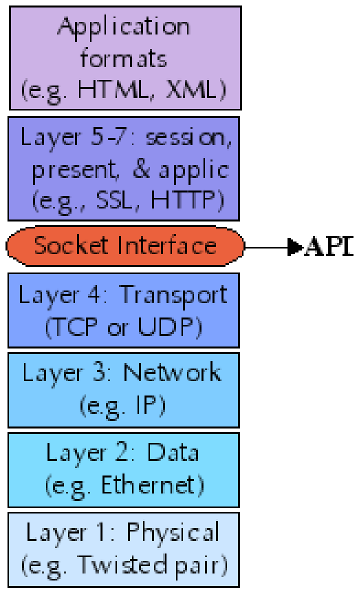
## I.INTRODUCTION

The Windows Sockets Specification has been built upon the Berkeley Sockets programming model which is the de facto standard for TCP/IP networking. It is intended to provide a high degree of familiarity for programmers who are used to programming with sockets in UNIX and other environments, and to simplify the task of porting existing sockets-based source code. The Windows Sockets API is consistent with release 4.3 of the Berkeley Software Distribution (4.3BSD).

**What is a network?**

What usually call a computer network is composed of a number of "network layers”, each providing a different restriction and/or guarantee about the data at that layer. The protocols at each network layer generally have their own packet formats, and headers.

The seven traditional layers of a network are divided into two groups: upper layers and lower layers. The sockets interface provides a uniform API to lower layers of a network, and allows implementing upper layers within r sockets application. Further, application data formats may themselves constitute further layers, e.g. SOAP is built on top of XML, and XML may itself utilize SOAP.


Application formats (e.g. HTML, XML)
Layer 5-7: session, present, & applic (e.g., SSL, HTTP)
Socket Interface
API
Layer 4: Transport (TCP or UDP)
Layer 3: Network (e.g. IP)
Layer 2: Data (e.g. Ethernet)
Layer 1: Physical (e.g. Twisted pair)

**What is a socket?**

While the sockets interface theoretically allows access to *protocol families* other than IP, in practice, every network layer use in r sockets application will use IP. At the transport layer, sockets support two specific protocols: TCP (transmission control protocol) and UDP (user datagram protocol). Sockets cannot be used to access lower (or higher) network layers; for example, a socket application does not know whether it is running over ethernet, token ring, or a dialup connection. Nor does the sockets pseudo-layer know anything about higher-level protocols like NFS, HTTP, FTP At times, the sockets interface is not r best choice for a network programming API. Specifically, many excellent libraries exist (in various languages) to use higher-level protocols directly, without having to worry about the details of sockets--the libraries handle those details. Lower-level layers than those sockets address fall pretty much in the domain of device driver programming.

- The basic building block for communication is the socket. A socket is an endpoint of communication to which a name may be bound. Each socket in use has a type and an associated process. Sockets exist within communication domains. A communication domain is an abstraction introduced to bundle common properties of threads communicating through sockets. Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The Windows Sockets facilities support a single communication domain: the Internet domain, which is used by processes which communicate using the Internet Protocol Suite. (Future versions of this specification may include additional domains.)

- Sockets are typed according to the communication properties visible to a user. Applications are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support.

- Two types of sockets currently are available to a user. A stream socket provides for the bi-directional, reliable, sequenced, and unduplicated flow of data without record boundaries.

- A datagram socket supports bi-directional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as Ethernet.

## IP, TCP and UDP

As the last panel indicated, when  program a sockets application,  have a choice to make between using TCP and using UDP. Each has its own benefits and disadvantages.

- TCP is a stream protocol, while UDP is a datagram protocol. TCP establishes a continuous open connection between a client and a server, over which bytes may be written--and correct order guaranteed--for the life of the connection. Hover, bytes written over TCP have no built-in structure, so higher-level protocols are required to delimit any data records and fields within the transmitted byte stream.

- UDP, on the other hand, does not require that any connection be established between client and server; it simply transmits a message between addresses. A nice feature of UDP is that its packets are self-delimiting--each datagram indicates exactly where it begins and ends. A possible disadvantage of UDP, hover, is that it provides no guarantee that packets will arrive in-order, or even at all. Higher-level protocols built on top of UDP may, of course, provide handshaking and acknowledgements.

- A useful analogy for understanding the difference between TCP and UDP is the difference between a telephone call and posted letters. The telephone call is not active until the caller "rings" the receiver and the receiver picks up. The telephone channel remains alive as long as the parties stay on the call--but they are free to say as much or as little as they wish to during the call. All remarks from either party occur in temporal order. On the other hand, when send a letter, the post office starts delivery without any assurance the recipient exists, nor any strong guarantee about how long delivery will take. The recipient may receive various letters in a different order than they re sent, and the sender may receive mail interspersed in time with those she sends. Unlike with the USPS, undeliverable mail always goes to the dead letter office, and is not returned to sender.

## Peers, ports, names, and addresses

Beyond the protocol--TCP or UDP--there are two things a peer (a client or server) needs to know about the machine it communicates with: An IP address and a port. An IP address is a 32-bit data value, usually represented for humans in "dotted quad" notation, e.g., 64.41.64.172. A port is a 16-bit data value, usually simply represented as a number less than 65536--most often one in the tens or hundreds range. An IP address gets a packet *to* a machine, a port lets the machine decide which process/service (if any) to direct it to. That is a slight simplification, but the idea is correct.

The above description is almost right, but it misses something. Most of the time when humans think about an internet host (peer), do not remember a number like 64.41.64.172, but instead a name like gnosis.cx.

## II.ABOUT SYNTAX

Most network applications can be divided into two pieces: a client and a server.

**Creating a socket**

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

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

When you create a socket there are three main parameters that you have to specify:

- the domain
- the type
- the protocol

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

**DESCRIPTION**

Socket creates an endpoint for communication and returns a descriptor. The domain parameter 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 |
|---|---|
| PF_UNIX,PF_LOCAL | Local communication |
| PF_INET | IPv4 Internet protocols |
| PF_INET6 | IPv6 Internet protocols |
| PF_IPX | IPX - Novell protocols |

The socket has the indicated type, which specifies the communication semantics. Currently defined types are:

SOCK_STREAM:

Provides sequenced, reliable, two-way, connection- based byte streams. An out-of-band data transmission mechanism may be supported.

SOCK_DGRAM:

Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

SOCK_SEQPACKET:

Provides a sequenced, reliable, two-way connection- based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each read system call.

SOCK_RAW:

Provides raw network protocol access.

SOCK_RDM:

Provides a reliable datagram layer that does not guarantee ordering.

SOCK_PACKET:

Obsolete and should not be used in new programs. Some socket types may not be implemented by all protocol families. For example, SOCK_SEQPACKET is not implemented for AF_INET.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the "communication domain" in which communication is to take place.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. They do not preserve record boundaries. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect call. Once connected, data may be transferred using read and write calls or some variant of the send and recv calls. When a session has been completed a close may be performed. Out-of-band data may also be transmitted as described in send and received as described in recv.

The communications protocols which implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered to be dead. When SO_KEEPALIVE is enabled on the socket the protocol checks in a protocol-specific manner if the other end is still alive. A SIGPIPE signal is raised if a process sends or receives on a broken stream; this causes naive processes, which do not handle the signal, to exit. SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded. Also all message boundaries in incoming datagrams are preserved.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send calls. Datagrams are generally received with recvfrom, which returns the next datagram with its return address.

SOCK_PACKET is an obsolete socket type to receive raw packets directly from the device driver. Use packet instead. When the network signals an error condition to the protocol module (e.g. using a ICMP message for IP) the pending error flag is set for the socket. The next operation on this socket will return the error code of the pending error. For some protocols it is possible to enable a per-socket error queue to retrieve detailed information about the error; see IP_RECVERR in ip. The operation of sockets is controlled by socket level options. These options are defined in <sys/socket.h>.Setsockopt and getsockopt are used to set and getoptions, respectively.

RETURN VALUE

-1 is returned if an error occurs otherwise the return value is a descriptor referencing the socket.

EXAMPLE: The Domain parameter specifies a communications domain within which communication will take place, in our example the domain parameter was AF_INET, that specify the ARPA Internet Protocols The Type parameter specifies the semantics of communication, in our mini chat we used the Stream socket type(SOCK_STREAM), because it offers a bi-directional, reliable, two-way connection based byte stream(resource 2). Finally the protocol type, since we used a Stream Socket type we must use a protocol that provide a connection-oriented protocol, like IP, so we decide to use IP in our protocol Type, and we saw in /etc/protocols the number of ip, 0. So our function now is:

```
s = socket(AF_INET , SOCK_STREAM , 0)
```

where 's' is the file descriptor returned by the socket function.

**Binding a socket to a port and waiting for the connections**

Like all services in a Network TCP/IP based, the sockets are always associated with a port, like Telnet is associated to Port 23, FTP to 21... In our Server we have to do the same thing, bind some port to be prepared to listening for connections (that is the basic difference between Client and Server), Listing 2. Bind is used to specify for a socket the protocol port number where it will be waiting for messages. So there is a question, which port could we bind to our new service? Since the system pre-defined a lot of ports between 1 and 7000 ( /etc/services ) we choose the port number 15000.

**BIND:**

Bind will bind a name to socket, it gives the socket sockfd, the local address myaddr, myaddr is addrlen bytes long.Traditionally, this is called assigning a name to a socket. When a socket is created with sockect () it exists in a name space but has no name assigned. It is normally necessary to assign local address using bind before sock_stream socket may receive connection.

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

The struct necessary to make socket works is the struct sockaddr_in address; and then we have the follow lines to say to system the information about the socket.

The type of socket

address.sin_family = AF_INET /* use a internet domain */

The IP used

address.sin_addr.s_addr = INADDR_ANY /*use a specific IP of host*/

The port used

address.sin_port = htons(15000); /* use a specific port number */

And finally bind our port to the socket

bind(create_socket , (struct sockaddr *)&address,sizeof(address));

Now another important phase, prepare a socket to accept messages from clients, the listen function is used on the server in the case of connection oriented communication and also the maximum number of pending connections

RETURN VALUE:

On sucess0 is return, on error –1 is returned and errno is set to appropriately.

**LISTEN:**

listen - listen for connections on a socket

#include <sys/socket.h>

int listen(int s, int backlog);

**DESCRIPTION**

To accept connections, a socket is first created with socket, a willingness to accept incoming connections and a queue limit for incoming connections are specified with listen, and then the connections are accepted with accept. The listen call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET. The backlog parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that retries succeed.

RETURN VALUE On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

ERRORS

EADDRINUSE: Another socket is already listening on the same port.

EBADF: The argument s is not a valid descriptor.

ENOTSOCK: The argument s is not a socket.

EOPNOTSUPP: The socket is not of a type that supports the listen operation.

**CONNECT**: initiate a connection on a socket

The parameters are the socket descriptor of the master socket (create_socket), followed by a sockeaddr_in structure and the size of the structure.(resource 3) Maybe the biggest difference is that client needs a Connect() function. The connect operation is used on the client side to identify and, possibly, start the connection to the server. The connect syntax is

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

#include <sys/socket.h>

int  connect(int sockfd, const struct sockaddr *serv_addr, sizeof(address));
```

**DESCRIPTION**

The file descriptor sockfd must refer to a socket. If the socket is of type SOCK_DGRAM then the serv_addr address isthe address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type SOCK_STREAM or SOCK_SEQPACKET, this call attempts to make a connection to another socket. The other socket is specified by serv_addr, which is an address (of length addrlen) in the communications space of the socket. Each communications space interprets the serv_addr parameter in its own way.

Generally, connection-based protocol sockets may successfully connect only once; connectionless protocol sockets may use connect multiple times to change their association. Connectionless sockets may dissolve the association by connecting to an address with the sa_family member of sockaddr set to AF_UNSPEC.

RETURN VALUE

If the connection or binding succeeds, zero is returned.

On error, -1 is returned, and errno is set appropriately.

ERRORS

The following are general socket errors only. There may be other domain-specific error codes.

EBADF The file descriptor is not a valid index in the descriptor table.

EFAULT The socket structure address is outside the user's address space.

**ACCEPT** it accept a socket connection. And to finish we have to tell the server to accept a connection, using the accept ( ) function. Accept is used with connection based sockets such as streams.

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

#include <sys/socket.h>

int   accept(int   s,  struct  sockaddr  *addr,  socklen_t *addrlen);
```

**DESCRIPTION**

The accept function is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET and SOCK_RDM). It extracts the first connection request on the queue of pending connections, creates a new connected socket with mostly the same properties as s, and allocates a new file descriptor for the socket, which is returned. The newly created socket is no longer in the listening state. The original socket s is unaffected by this call. Note that any per file descriptor flags (everything that can be set with the F_SETFL fcnlab38tl, like non blocking or async state) are not inherited across an accept.

The argument addr is a pointer to a sockaddr structure. This structure is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the address passed in the addr parameters determined by the socket's family The addrlen argument is a value-result parameter: it should initially contain the size of the structure pointed to by addr, on return it will contain the actual length (in bytes) of the address returned. When addr is NULL nothing is filled in.

If no pending connections are present on the queue, and the socket is not marked as non-blocking, accept blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, accept returns EAGAIN.

In order to be notified of incoming connections on a socket, you can use select. A readable event will be delivered when a new connection is attempted and you may then call accept to get a socket for that connection. Alternatively, you can set the socket to deliver SIGIO when activity occurs on a socket;

There may not always be a connection waiting after a SIGIO is delivered or select or poll return a readability event because the connection might have been removed by an asynchronous network error or another thread before accept is called. If this happens then the call will block waiting for the next connection to arrive. To ensure that accept never blocks, the passed socket s needs to have the O_NONBLOCK flag set .

RETURN VALUE

The call returns -1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket

**RECV, RECVFROM:** receive a message from a socket

#include <sys/types.h>

#include <sys/socket.h>

int recv(int s, void *buf, size_t len, int flags);

int recvfrom(int s, void *buf, size_t len, int flags

struct sockaddr *from, socklen_t *fromlen);

**DESCRIPTION**

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

If from is not NULL, and the socket is not connection-oriented, the source address of the message is filled in the argument fromlen is a value-result parameter, initialized to the size of the buffer associated with from, and modified on return to indicate the actual size of the address stored there.

The recv call is normally used only on a connected socket and is identical to recvfrom with a NULL from parameter.

All three routines return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from socket when more data arrives.

The flags argument to a recv call is formed by OR'ing one or more of the following values:

RETURN VALUE

These calls return the number of bytes received, or -1 if an error occurred.

**SENDTO, SEND:** send a message from a socket

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

#include <sys/socket.h>

int send(int s, const void *msg, size_t len, int flags);

int  sendto(int s, const void *msg, size_t len, int flags,

const struct sockaddr *to, socklen_t tolen);
```

**DESCRIPTION**

Send, sendto are used to transmit a message to another socket. Send may be used only when the socket is in a connected state, while sendto may be used at any time.The address of the target is given by to with tolen specifying its size. The length of the message is given by len. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted. No indication of failure to deliver is implicit in a send. Locally detected errors are indicated by a return value of-1.

When the message does not fit into the send buffer of the socket, send normally blocks, unless the socket has been placed in non-blocking I/O mode. In non-blocking mode it would return EAGAIN in this case. The select call may be used to determine when it is possible to send more data.**RETURN VALUE**

The calls return the number of characters sent, or -1 if an error occurred.

## III. Design of Server Software

Servers can be classified based on

1) their servicing discipline (iterative or concurrent);
2) their communication methods (connection-oriented or connectionless);
3) their information that they keep (stateless or stateful).

**Iterative Server**

An iterative server serves the requests one after the other.

It is easier to design and implement.

Iterative design is suitable when the service time for eachrequest is small (because the mean response time is stillacceptably small).

It is suitable for simple services such as the TIMEservice. Iterative design is not suitable when the service time for arequest may be large.

Example

Two clients are using a file transfer service:

• the 1st client requests to get a file of size 200 Mbytes,

• the 2nd client requests to get a file of size 20 bytes.

If iterative design is used, the 2nd client has to wait a long time.

**Concurrent Server**

A concurrent server serves multiple clients simultaneously.

Concurrency refers to either

1) real simultaneous computing (using multiple processors)

2) apparent simultaneous computing (by time sharing).Concurrent design is more difficult to implement, but it can give a smaller response time when

3) Different clients require very different service time, or the service requires significant I/O, or the server is executed on a computer with multiple processors.

Process and Thread

We can use processes and threads to realize concurrency on a processor via time sharing.

Process

A process is a fundamental unit of computation. The OS has to store various information about each process.

Thread

• Windows provides a second form of concurrent execution known as threads of execution or threads.

• Each thread must be associated with a single process.

• A thread is executed independently of other threads.

• All threads in a process share: (i) global variables and (ii) resources that the OS allocates to the process.

• Each thread in a process has its own local variables.

For example, if multiple threads execute the following
piece of code,

```
for ( i=1 ; i<=10 ; i++ )
printf ( "%d\n" , i ) ;
```

then each thread has its own index variable i.

**Connection-Oriented Servers**

A connection-oriented server uses TCP for connection-oriented communication.

TCP provides reliable transport. The servers need not do so. The servers are simpler.

A server uses a separate socket for each connection.

**Connectionless Servers**

A connectionless server uses UDP for connectionlesscommunication. UDP does not guarantee reliable transport. A connectionless server may need to realize reliabilitythrough timeout and retransmission. The server and client are relatively complicated.

**Stateless Server**

A stateless server does not keep information about the
ongoing interactions with each client.

Example

• Consider the service that a client can read a file storedon the server's computer.

• Tomaintain stateless, each client's request must specifya file name, a position in the file, and the number ofbytes to read.

• The server handles each request independently:

• open the specified file,

• seek to the specified position,

• read the specified number of bytes,

• send these bytes to the client,

• close the file.

Stateless servers are less efficient but they are more reliable they may be a good choice for UDP transport.

**Stateful Servers**

A stateful server keeps information about the status ongoing interactions with each client.

Example

• Consider the file-reading service.

• The server maintains the following information:

• File reading can be more efficient.

Four possible types of server designs:

1. Iterative, connection-oriented server.
2. Iterative, connectionless server.
3. Concurrent, connection-oriented server.
4. Concurrent, connectionless server.

Some iterative server use tcp because to send data reliable

## 1 (ii) Understanding of arp, telnet, ftp, finger, traceroute, ifconfig, netstat, ping Commands

### ARP

arp - manipulate the system ARP cache

```
arp [-evn] [-H type] [-i if] -a [hostname]

arp [-v] [-i if] -d hostname [pub]

arp [-v] [-H type] [-i if] -s hostname hw_addr [temp]

arp  [-v]  [-H  type] [-i if] -s hostname hw_addr [netmask nm] pub

arp [-v] [-H type] [-i if] -Ds hostname ifa  [netmask  nm]  pub

arp [-vnD] [-H type] [-i if] -f [filename]
```

### DESCRIPTION

Arp manipulates the kernel's ARP cache in various ways. The primary options are clearing an address mapping entry and manually setting up one. For debugging purposes, the arp program also allows a complete dump of the ARP cache.

OPTIONS

-v, --verbose

Tell the user what is going on by being verbose.

-n, --numeric

shows numerical addresses instead of trying to determine symbolic host, port or user names.

-H type, --hw-type type, -t type

## TELNET

NAME

telnet - user interface to the TELNET protocol

```
telnet [-8EFKLacdfrx] [-X authtype] [-b hostalias] [-e escapechar]
       [-k realm] [-l user] [-n tracefile] [host [port]]
```

**DESCRIPTION**

The telnet command is used to communicate with another host using the TELNET protocol. If telnet is invoked without the host argument, it enters command mode, indicated by its prompt (telnet>). In this mode, it accepts and executes the commands listed below. If it is invoked with arguments, it performs an open command with those arguments.

The options are as follows:

-8 Specifies an 8-bit data path. This causes an attempt to negotiate the TELNET BINARY option on both input and output.

-E Stops any character from being recognized as an escape character.

-F If Kerberos V5 authentication is being used, the -F option allows the local credentials to be forwarded to the remote system including any credentials that have already been forwarded into the local environment.

-K Specifies no automatic login to the remote system.

-L Specifies an 8-bit data path on output. This causes the BINARY option to be negotiated on output.

-X atype Disables the atype type of authentication.

-a Attempt automatic login. Currently, this sends the user name via the USER variable of the ENVIRON option if supported by the remote system. The name used is that of the current user as returned by getlogin(2) if it agrees with the current user ID, otherwise it is the name associated with the user ID.

-b hostalias

Uses bind(2) on the local socket to bind it to an aliased address (see ifconfig(8) and the ``alias'' specifier) or to the address of another interface than the one naturally chosen by connect(2).

This can be useful when connecting to services which use IP addresses for authentication and reconfiguration of the server is undesirable (or impossible).

-c Disables the reading of the user's .telnetrc file.

-d Sets the initial value of the debug toggle to TRUE.

-e escapechar

Sets the initial telnet escape character to escapechar. If escapechar is omitted, then there will be no escape character.

-f If Kerberos V5 authentication is being used, the -f option allows the local credentials to be forwarded to the remote system.

-k realm

If Kerberos authentication is being used, the -k option requests that telnet obtain tickets for the remote host in realm realm instead of the remote host's realm, as determined by krb_realmofhost(3).

-l user

When connecting to the remote system, if the remote system understands the ENVIRON option, then user will be sent to the remote system as the value for the variable USER. This option implies the -a option. This option may also be used with the open command.

-n tracefile

Opens tracefile for recording trace information.

-r Specifies a user interface similar to rlogin(1). In this mode, the escape character is set to the tilde (~) character, unless modified by the -e option.

-x Turns on encryption of the data stream if possible.

host Indicates the official name, an alias, or the Internet address of a remote host.

port Indicates a port number (address of an application). If a number is not specified, the default telnet port is used. When in rlogin mode, a line of the form ~. disconnects from the remote prompt.

**FINGER**

finger - user information lookup program

finger [-lmsp] [user ...] [user@host ...]

**DESCRIPTION**

The finger displays information about the system users.

Options are:

-s Finger displays the user's login name, real name, terminal name and write status (as a ``*'' after the terminal name if write permission is denied), idle time, login time, office location and office phone number. Login time is displayed as month, day, hours and minutes, unless more than six months ago, in which case the year is displayed rather than the hours and minutes. Unknown devices as well as nonexistent idle and login times are displayed as single asterisks.

-l Produces a multi-line format displaying all of the information

Described for the -s option as well as the user's home directory, home phone number, login shell, mail status, and the contents of he files ``.plan", ``.project", ``.pgpkey" and ``.forward" from the user's home directory. If write permission is denied to the device, the phrase ``(messages off)" is appended to the line containing the device name. One entry per user is displayed with the -l option; if a user is logged on multiple times, terminal information is repeated once per login.

Mail status is shown as ``No Mail." if there is no mail at all,

``Mail last read DDD MMM ## HH:MM YYYY (TZ)" if the person has looked at their mailbox since new mail arriving, or ``New mail received ...", `` Unread since ..." if they have new mail. -p Prevents the -l option of finger from displaying the contents of the ``.plan", ``.project" and ``.pgpkey" files.

-m Prevent matching of user names. User is usually a login name; how ever, matching will also be done on the users' real names, unless the -m option is supplied. All name matching performed by finger is case insensitive.

If no options are specified, finger defaults to the -l style output if operands are provided, otherwise to the -s style. Note that some fields may be missing, in either format, if information is not available for them. If no arguments are specified, finger will print an entry for each user currently logged into the system. Finger may be used to look up users on a remote machine. The format is to specify a user as ``user@host", or ``@host", where the default output format for the former is the -l style, and the default output format for the latter is the -s style. The -l option is the only option that may be passed to a remote machine.

If standard output is a socket, finger will emit a carriage return (^M) before every linefeed (^J). This is for processing remote finger requests when invoked by fingered(8).

**FTP**

NAME ftp - Internet file transfer program

```
ftp [-pinegvd] [host]

pftp [-inegvd] [host]
```

**DESCRIPTION**

Ftp is the user interface to the Internet standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site Options may be specified at the command line, or to the command interpreter.

-p Use passive mode for data transfers. Allows use of ftp in environ¡ments where a firewall prevents connections from the outside world back to the client machine. Requires that the ftp server support the PASV command. This is the default now for all clients (ftp and pftp) due to security concerns using the PORT transfer mode. The flag is kept for compatibility only and has no effect anymore.

-i Turns off interactive prompting during multiple file transfers.

-n Restrains ftp from attempting ``auto-login" upon initial connection. If auto-login is enabled, ftp will check the .netrc (see netrc(5)) file in the user's home directory for an entry describing an account on the remote machine. If no entry exists, ftp will prompt for the remote machine login name (default is the user identity on the local machine), and, if necessary, prompt for a password and an account with which to login.

-e Disables command editing and history support, if it was compiled into the ftp executable. Otherwise, does nothing.

-g Disables file name globbing.

-v Verbose option forces ftp to show all responses from the remote

server, as well as report on data transfer statistics.

-d Enables debugging. The client host with which ftp is to communicate may be specified on the command line. If this is done, ftp will immediately attempt to establish a connection to an FTP server on that host; otherwise, ftp will enter its command interpreter and await instructions from the user. When ftp is awaiting commands from the user the prompt `ftp>' is provided to the user. The following commands are recognized by ftp:

! [command [args]]

Invoke an interactive shell on the local machine. If there are arguments, the first is taken to be command to execute directly, with the rest of the arguments as its arguments.

$ macro-name [args]

Execute the macro macro-name that was defined with the macdef command. Arguments are passed to the macro unglobbed.

account [passwd] Supply a supplemental password required by a remote system for access to resources once a login has been successfully completed. If no argument is included, the user will be prompted for an account password in a non-echoing input mode.

append local-file [remote-file]

Append a local file to a file on the remote machine.If remote-file is left unspecified, the local file name is used in naming the remote file after being altered by any ntrans or nmap setting. File transfer uses the current settings for type, format, mode, and structure.

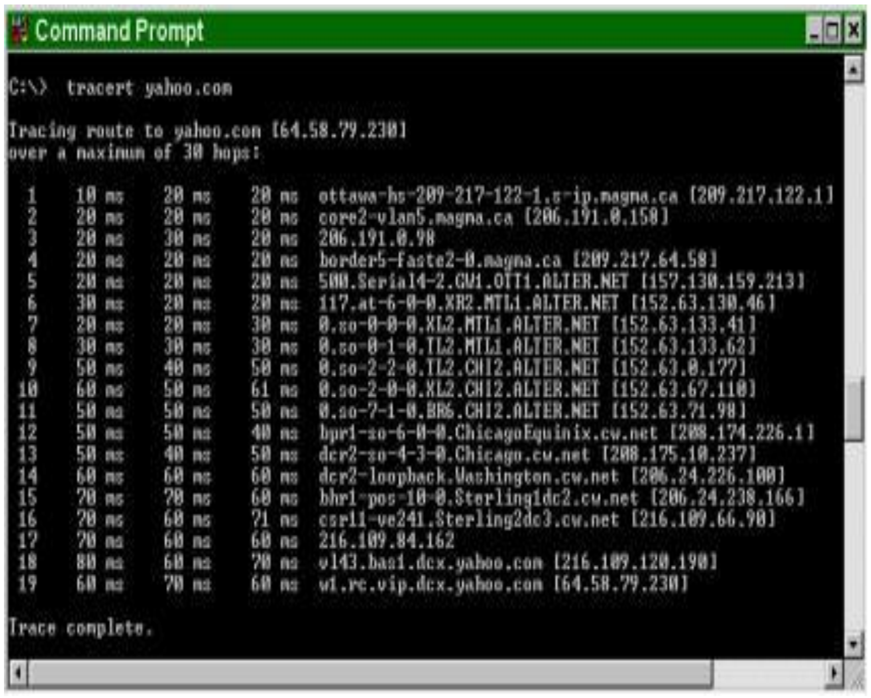
**TRACEROUTE**

The traceroute command traces the network path of Internet routers that packets take as they are forwarded from your computer to a destination address. The "length" of the network connection is indicated by the number of Internet routers in the traceroute path. Traceroutes can be useful to diagnose slow network connections. For example, if you can usually reach an Internet site but it is slow today, then a traceroute to that site should show you one or more hops with either long times or marked with "*" indicating the time was really long. If so, the blockage could be anywhere from your Internet service provider to a backbone provider, and there is likely little you can do except wait with the infinite patience of the mighty oak.

**UNDERSTANDING OF FTP COMMAND**

The subsections below describe operating system traceroute versions, traceroute web sites, lists of traceroute information sites, and the original version of the traceroute program.

Traceroute operating systems. On a Windows computer, you can run a traceroute in an MSDOS or Command window by typing "tracert" followed by the domain name, for example as in "tracert www.yahoo.com"



```
C:\>  tracert yahoo.com

Tracing route to yahoo.com [64.58.79.230]
over a maximum of 30 hops:

  1    10 ms    20 ms    20 ms  ottawa-hs-209-217-122-1.s-ip.magma.ca [209.217.122.1]
  2    20 ms    20 ms    20 ms  core2-vlan5.magma.ca [206.191.0.150]
  3    20 ms    30 ms    20 ms  206.191.0.98
  4    20 ms    20 ms    20 ms  border5-faste2-0.magma.ca [209.217.64.58]
  5    20 ms    20 ms    20 ms  500.Serial4-2.GW1.OTT1.ALTER.NET [157.130.159.213]
  6    30 ms    20 ms    20 ms  117.at-6-0-0.XR2.MTL1.ALTER.NET [152.63.130.46]
  7    20 ms    20 ms    30 ms  0.so-0-0-0.XL2.MTL1.ALTER.NET [152.63.133.41]
  8    30 ms    30 ms    30 ms  0.so-0-1-0.TL2.MTL1.ALTER.NET [152.63.133.62]
  9    50 ms    40 ms    50 ms  0.so-2-2-0.TL2.CHI2.ALTER.NET [152.63.0.177]
 10    60 ms    50 ms    61 ms  0.so-2-0-0.XL2.CHI2.ALTER.NET [152.63.67.110]
 11    50 ms    50 ms    50 ms  0.so-7-1-0.BR6.CHI2.ALTER.NET [152.63.71.98]
 12    50 ms    50 ms    40 ms  bpr1-so-6-0-0.ChicagoEquinix.cw.net [208.174.226.1]
 13    50 ms    40 ms    50 ms  dcr2-so-4-3-0.Chicago.cw.net [208.175.10.237]
 14    60 ms    60 ms    60 ms  dcr2-loopback.Washington.cw.net [206.24.226.100]
 15    70 ms    70 ms    60 ms  bhr1-pos-10-0.Sterling1dc2.cw.net [206.24.238.166]
 16    70 ms    60 ms    71 ms  csr11-ve241.Sterling2dc3.cw.net [216.109.66.90]
 17    70 ms    60 ms    60 ms  216.109.84.162
 18    80 ms    60 ms    70 ms  v143.bas1.dcx.yahoo.com [216.109.120.190]
 19    60 ms    70 ms    60 ms  w1.rc.vip.dcx.yahoo.com [64.58.79.230]

Trace complete.
```

Like most Internet utilities, the traceroute command was originally developed for Unix computers. The options for the original Unix traceroute command line version are shown below:

traceroute [-m #] [-q #] [-w #] [-p #] {IP_address|host_name}

Print the route packets take to network host.

SYNTAX

traceroute [-d] [-F] [-I] [-n] [-v] [-x] [-f first_ttl] [-g gateway [-g gateway] | -r] [-i iface] [-m max_ttl] [-p port] [-q nqueries] [-s src_addr] [-t tos] [-w waittime ] host [packetlen]

EXAMPLES

Traceroute computerhope.com - would display results similar to the following:

```
traceroute to computerhope.com (166.70.10.23), 30 hops max, 40 byte packets
1 198.60.22.1 (198.60.22.1) 2.303 ms 1.424 ms 2.346 ms
2 krunk3.xmission.com (198.60.22.6) 0.742 ms * 1.521 ms
```

Note: in this example because we are local to the address we are tracerouting the amount of hops is very minimal. However, when you traceroute computerhope.com you may hop more than we do. This command is very useful for distinguishing network / router issues. If the domain does not work or is not available you can traceroute an IP.

**NETSTAT**

Netstat command netstat displays the contents of various network-related data structures in various formats, depending on the options you select. The first form of the command displays a list of active sockets for each protocol. The second form selects one from among various other network data structures. The third form shows the state of the interfaces. The fourth form displays the routing table, the fifth form displays the multicast routing table, and the sixth form displays the state of DHCP on one or all interfaces.

## Implementation of time services using connection oriented sockets systems calls.

**DESCRIPTION**

The TIME service is an Internet protocol. Its purpose is to provide a site-independent, machine readable date and time. Time can operate over either TCP or UDP. When operating over TCP, a host connects to a server that supports the TIME protocol on TCP port 37. The server then sends the time as a 32-bit unsigned binary number in network byte order representing a number of seconds since 00:00 (midnight) 1 January, 1900 GMT and closes the connection. The host receives the time and closes the connection.

When operating over UDP, the client sends a (typically empty) datagram to UDP port 37. The server responds with a single datagram of length 4 containing the time. There is no connection setup or teardown. The TIME protocol is used by the Linux rdate command. The TIME protocol has been superseded by the Network Time Protocol (NTP).

**Time Protocol**

This RFC specifies a standard for the ARPA Internet community. Hosts on the ARPA Internet that choose to implement a Time Protocol are expected to adopt and implement this standard.

This protocol provides a site-independent, machine readable date and time. The Time service sends back to the originating source the time in seconds since midnight on January first 1900.

One motivation arises from the fact that not all systems have a date/time clock, and all are subject to occasional human or machine error. The use of time-servers makes it possible to quickly confirm or correct a system's idea of the time, by making a brief poll of several

independent sites on the network.

This protocol may be used either above the Transmission Control Protocol (TCP) or above the User Datagram Protocol (UDP). When used via TCP the time service works as follows:

S: Listen on port 37 (45 octal).

U: Connect to port 37.

S: Send the time as a 32 bit binary number.

U: Receive the time.

U: Close the connection.

S: Close the connection.

The server listens for a connection on port 37. When the connection is established, the server returns a 32-bit time value and closes the connection. If the server is unable to determine the time at its site, it should either refuse the connection or close it without sending anything. When used via UDP the time service works as follows:

S: Listen on port 37 (45 octal).

U: Send an empty datagram to port 37.

S: Receive the empty datagram.

S: Send a datagram containing the time as a 32 bit binary number.

U: Receive the time datagram.

The server listens for a datagram on port 37. When a datagram arrives, the server returns a datagram containing the 32-bit time value. If the server is unable to determine the time at its site, it should discard the arriving datagram and make no reply.

The time is the number of seconds since 00:00 (midnight) 1 January 1900

GMT, such that the time 1 is 12:00:01 am on 1 January 1900 GMT; this base will serve until the year 2036.

For example: the time 2,208,988,800 corresponds to 00:00 1 Jan 1970 GMT,

2,398,291,200 corresponds to 00:00 1 Jan 1976 GMT, 2,524,521,600 corresponds to 00:00 1 Jan 1980 GMT,

2,629,584,000 corresponds to 00:00 1 May 1983 GMT,

and -1,297,728,000 corresponds to 00:00 17 Nov 1858 GMT.

**ALGORITHM**

1. This program implements the time using TCP protocol in the integer formatted way such that it gives the total time in seconds from some reference point.
2. It is shown in the example below:

   ```
   Tme is -> 3475219317
   ```

3. In this program the standard port number used is “37”.
4. The type of the family used is AF_INET and the port no used is”13”and Internet address used is “192.168.2.150” .
5. The socket is created in the program uses the SOCK_STREAM method which corresponds to the TCP protocol
6. The headers implemented are “stdio.h” ,”sys/socket.h”,netinet/in.h”.
7. The first one is the standard input output file,the second is the type of the connection that is implementation in the program ,the third is the socket creation file, the last one is the internet protocol

## Prog. To Implement Time Service

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<string.h>

#define ERROR -1

main()

{

        unsigned long timeval,tempval;

        int sfd;

        struct sockaddr_in serv_addr;

        sfd=socket(AF_INET,SOCK_STREAM,0);

        if(sfd==ERROR)

        {

                perror("socket failed");

                exit(1);

        }

        serv_addr.sin_family=AF_INET;

        serv_addr.sin_addr.s_addr=inet_addr("192.168.2.150");

        serv_addr.sin_port=htons(37);
```

```
	if(connect(sfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr))==ERROR)
	{
		perror("connection failed");
		exit(1);
	}
	read(sfd,&tempval,80);
	timeval=htonl(tempval);
	printf("\n TIME BY USING TOP \n");
	printf("\n\tTIME is ----> %u\n\n",timeval);
	close(sfd);
}
```

**OUTPUT:**

```
TIME BY USING TCP
TIME is ----> 3281892359
```

## Implementation of Daytime services using connection oriented sockets systems calls.

**DESCRIPTION**

**TCP Based Daytime Service**

One daytime service is defined as a connection based application on TCP. A server listens for TCP connections on TCP port 13. Once a connection is established the current date and time is sent out the connection as a ascii character string (and any data received is thrown away). The service closes the connection after sending the quote.

**UDP Based Daytime Service**

Another daytime service service is defined as a datagram based application on UDP. A server listens for UDP datagrams on UDP port13. When a datagram is received, an answering datagram is sent containing the current date and time as a ASCII character string (the data in the received datagram is ignored).

**Daytime Syntax**

There is no specific syntax for the daytime. It is recommended that it be limited to the ASCII printing characters, space, carriage return, and line feed. The daytime should be just one line. One popular syntax is: Weekday, Month Day, Year Time-Zone

Example:

Tuesday, February 22, 1982 17:37:43-PST

Daytime Protocol

Another popular syntax is that used in SMTP:

dd mmm yy hh:mm:ss zzz

Example: 02 FEB 82 07:59:01 PST

**ALGORITHM:**

1. This is the implementation of the daytime using TCP method. In this program the time is denoted in some formatted type as below

   DAY TIME IS---->15 FEB 2010 16:23:24 IST

2. In this program the standard port number used is “13”

3. The type of the family used is AF_INET and the port no used is"13"and internet address used is "192.168.2.150" .

4. The socket is created in the program uses the SOCK_STREAM method which corresponds to the TCP protocol .

5. The headers implemented are "stdio.h" ,"sys/socket.h",netinet/in.h".

6. The first one is the standard input output file,the second is the type of the connection that is implementation in the program ,the third is the socket creation file, the last one is the internet protocol

## Prog.To Implement Day-Time Service

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#define ERROR -1

main()

{

    char str[80];

    int sfd,cn;

    struct sockaddr_in serv_addr;

    sfd=socket(AF_INET,SOCK_STREAM,0);

    if(sfd==ERROR)

    {

        perror("socket failed");
```

```
                exit(1);
        }
        serv_addr.sin_family=AF_INET;
        serv_addr.sin_addr.s_addr=inet_addr("192.168.2.150");
        serv_addr.sin_port=htons(13);
        cn=connect(sfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr));
        if(cn==ERROR)
        {
                perror("connection failed");
                exit(1);
        }
        read(sfd,str,30);
        printf("\nDAY TIME USING TCP\n");
        printf("\n\t DAY TIME IS ---->%s\n\n",str);
        close(sfd);
}
```

**OUTPUT:**

```
[staff]$ cc daytime.c
[staff]$ ./a.out
DAY TIME USING TCP
         DAY TIME IS ---->01 jul 2018 01:01:55 IST
```

## Demonstrate the use of Advanced socket system call (readv,writev)

READV, WRITEV: read or write a vector

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

int readv(int fd, const struct iovec * vector, int count);

int writev(int  fd,  const  struct  iovec  *  vector,  int count);

struct iovec {

  __ptr_t iov_base; /* Starting address.  */

  size_t iov_len; /* Length in bytes.  */

};
```

**DESCRIPTION**

readv reads data from file descriptor fd, and puts the result in the buffers described by vector. The number of buffers is specified by count. The buffers are filled in the order specified. Operates just like read except that data is put in vector instead of a contiguous buffer.

writev writes data to file descriptor fd, and from the buffers described by vector. The number of buffers is specified by count. The buffers are used in the order specified. Operates just like write except that data is taken from vector instead of a contiguous buffer.

RETURN VALUE

On success readv returns the number of bytes read. On success writev returns the number of bytes written. On error, -1 is returned, and errno is set appropriately.

ERRORS

EINVAL An invalid argument was given. For instance count might be greater than MAX_IOVEC, or zero. fd could also be attached to an object which is unsuit able for reading (for readv) or writing (for writev).

EFAULT "Segmentation fault." Most likely vector or some of the iov_base pointers points to memory that is not properly allocated.

**ALGORITHM:**

1. Set-up the base-address using the iovector() function.
2. Creating a connection-Oriented server socket
3. Binding the structure to the server socket.
4. Listen to indicate that it is ready to receive connections
5. Accept takes the first connection request.
6. At the client's side , connect system call to establish a connection with a server.
7. For multiple reads and multiple writes , multiple buffers are used.

## Prog. To Implement writev()

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<arpa/inet.h>

#include<netinet/in.h>

#include<sys/uio.h>

#include<fcntl.h>
```

```
main()

{

        char s1[20],s2[20],s3[20],s4[20];

        struct iovec iov[4];

        int fd;

        fd=open("file.txt",O_WRONLY);

        printf("Enter 1st string\n");

        scanf("%s",&s1);

        printf("Enter 2nd string\n");

        scanf("%s",&s2);

        printf("Enter 3rd string\n");

        scanf("%s",&s3);

        printf("Enter 4th string\n");

        scanf("%s",&s4);

        iov[0].iov_base=s1;

        iov[0].iov_len=sizeof(s1);

        iov[1].iov_base=s2;

        iov[1].iov_len=sizeof(s2);

        iov[2].iov_base=s3;

        iov[2].iov_len=sizeof(s3);

        iov[3].iov_base=s4;

        iov[3].iov_len=sizeof(s4);
        writev(fd,(struct iovec *)&iov,4);
}
```

**OUTPUT:**

```
[staff]$ cc writev.c
[staff]$ ./a.out
Enter 1st string: hello
Enter 2nd string: world
Enter 3rd string: welcome
Enter 4th string: bye!
```

## Prog. To Implement readv()

```
#include<sys/types.h>
#include<sys/socket.h>
#include<sys/uio.h>
#include<arpa/inet.h>
#include<netinet/in.h>
#include<fcntl.h>
main()
{
        char s1[20],s2[20],s3[20],s4[20];
        struct iovec iov[4];
        int fd;
        iov[0].iov_base=s1;
        iov[0].iov_len=sizeof(s1);

        iov[1].iov_base=s2;

        iov[1].iov_len=sizeof(s2);

        iov[2].iov_base=s3;

        iov[2].iov_len=sizeof(s3);

        iov[3].iov_base=s4;

        iov[3].iov_len=sizeof(s4);

        fd=open("file.txt",O_RDONLY);

        readv(fd,(struct iovec *)&iov,4);

        printf("first string is:%s\n",s1);

        printf("second string is:%s\n",s2);

        printf("third string is:%s\n",s3);

        printf("fourth string is:%s",s4);

}
```

**OUTPUT:**

```
[staff]$ cc readv.c

[staff]$ ./a.out

first string is:hello

second string is:world

third string is:welcome

fourth string is:bye!
```

## Demonstrate the use of Advanced socket system call (setsock(),getsock())

SETSOCKOPT,GETSOCKOPT:get and set options on sockets

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

    #include <sys/socket.h>

    int  getsockopt(int  s, int level, int optname, void *optval, socklen_t *optlen);

    int setsockopt(int s, int level, int optname,  const  void *optval, socklen_t optlen);
```

**DESCRIPTION**

Getsockopt and setsockopt manipulate the options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost socket level. When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, level is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, toindicate that an option is to be interpreted by the TCP protocol, level should be set to the protocol number of TCP. The parameters optval and optlen are used to access option values for setsockopt. If no option value is to be supplied or returned, optval may be NULL. Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file <sys/socket.h> contains definitions for socket level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section 4 of the manual. Most socket-level options utilize an int parameter for optval. For setsockopt, the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled.

RETURN VALUE On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

GETSOCKNAME: - get socket name

#include <sys/socket.h>

int getsockname(int s , struct sockaddr * name , socklen_t * namelen )

**DESCRIPTION**

Getsockname returns the current name for the specified socket. The namelen parameter should be initialized to indicate the amount of space pointed to by name. On return it contains the actual size of the name returned(in bytes).

RETURN VALUE On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

ERRORS EBADF The argument s is not a valid descriptor.

ALGORITHM:

1. int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
2. int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
3. When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, level is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, level should be set to the protocol number of TCP; see getprotoent(3).
4. For getsockopt, optlen is a value-result parameter, initially containing the size of the buffer pointed to by optval, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, optval may be NULL. Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation.

## Prog.To Implement Advanced Socket System Calls(getsock(),setsock())

```
#include<stdio.h>

#include<arpa/inet.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<fcntl.h>

#include<netinet/tcp.h>

#include<sys/types.h>

main(int argc,char *argv[])

{

    int sockfd,a,b,c;

    sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

    int len=sizeof(a);

    char s[20];

    getsockopt(sockfd,IPPROTO_TCP,TCP_MAXSEG,&a,&len);

    {

        printf("PROTOCOL LEVEL>Da Max Segment Size is:%d\n",a);

    }

    getsockopt(sockfd,IPPROTO_TCP,TCP_NODELAY,s,len);

    getsockopt(sockfd,IPPROTO_TCP,TCP_NODELAY,&s,&len);

    {
        printf("PROTOCOL LEVEL>Da Option Value is:%d\n",b);

    }

    setsockopt(sockfd,SOL_SOCKET,SO_KEEPALIVE,s,len);

    getsockopt(sockfd,SOL_SOCKET,SO_KEEPALIVE,&s,&len);

    {

        printf("SOCKET LEVEL>Da Option Values is:%d\n",c);

    }

}
```

**OUTPUT:**

```
[staff]$ ./a.out

PROTOCOL LEVEL>Da Max Segment Size is:536

PROTOCOL LEVEL>Da Option Value is:0

SOCKET LEVEL>Da Option Values is:1
```

## Demonstrate the use of Advanced socket system call (setsock(),getsock())

GETPEERNAME **:** get name of connected peer

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

  int  getpeername(int  s,  struct sockaddr *name, socklen_t *namelen);
```

**DESCRIPTION**

Getpeername returns the name of the peer connected to socket s. The namelen parameter should be initialized to indicate the amount of space pointed to by name. On return it contains the actual size of the name returned(in bytes). The name is truncated if the buffer provided is too small.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

ALGORITHM:

1. Create a TCP socket and fill the internet socket address structure with server IP

   addresses and port number

2. Bind the server with known part to the socket created by calling ‘bind’ command.

3. Convert the socket created to listening socket or by calling ‘listen’ command.

4. Then by calling ‘getsockname’ function the local address details of socket are obtained.

5. Then by using the displaying function the details of IP

6. Address , port number are displayed at the standard output.

## Prog. To Implement Advanced Socket System Call(getpeername())

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#define ERROR -1

main()

{

        int s,k;

        struct sockaddr_in server,addr;

        socklen_t len;
        s=socket(AF_INET,SOCK_STREAM,0);

        server.sin_family=AF_INET;

        inet_aton("192.168.2.150",&server.sin_addr);

        server.sin_port=htons(80);

        k=connect(s,(struct sockaddr*)&server,sizeof(server));

        if(k<0)

        {

                perror("connected");

                exit(0);

        }

        len=sizeof(addr);

        getpeername(s,(struct sockaddr*)&addr,&len);

        printf("Peer IP Address:%s\n",inet_ntoa(addr.sin_addr));

        printf("Peer Port:%d\n",ntohs(addr.sin_port));

}
```

**OUTPUT:**

```
[staff]$ cc getpeername.c

[staff]$ ./a.out

Peer IP Address:192.168.2.150

Peer Port:37
```

## Implementation of Remote Procedure call using Socket system calls.

**DESCRIPTION** : RPC is a network programming model for point-to-point communication within or between software applications.

In RPC, the sender makes a request in the form of a procedure, function, or method call. RPC translates these calls into requests sent over the network to the intended destination. The RPC recipient then processes the request based on the procedure name and argument list, sending a response to the sender when complete. RPC applications generally implement software modules called "proxies" and "stubs" that broker the remote calls and make them appear to the programmer the same as local procedure calls (LPC). RPC calling applications usually operate synchronously, waiting for the remote procedure to return a result. RPC incorporates timeout logic to handle network failures or other situations where RPCs do not return. RPC has been a common programming technique in the Unix world since the 1990s. The Open Systems Foundation (OSF) Distributed Computing Environment (DCE) and Sun Microsystems Open Network Computing (ONC) libraries both were widely deployed. More recent examples of RPC technologies include Microsoft DCOM, Java RMI, and XML-RPC and SOAP.Remote procedure call (RPC) is an Inter-process communication technology that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer would write essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question is written using object-oriented principles, RPC may be referred to as remote invocation or remote method invocation.

Note that there are many different (often incompatible) technologies commonly used to accomplish is

| The Internet Protocol Suite |
|---|
| Application Layer |
| BGP · DHCP · DNS · FTP · GTP · HTTP · IMAP · IRC · Megaco · MGCP · NNTP · NTP · POP · RIP · RPC · RTP · RTSP · SDP · SIP · SMTP · SNMP · SOAP · SSH · Telnet · TLS/SSL · XMPP · (more) |
| Transport Layer |
| TCP · UDP · DCCP · SCTP · RSVP · ECNLAB38 · (more) |
| Internet Layer |
| IP (IPv4, IPv6) · ICMP · ICMPv6 · IGMP · IPsec · (more) |
| Link Layer |
| ARP/InARP · NDP · OSPF · Tunnels (L2TP) · PPP · Media Access Control (Ethernet, DSL, ISDN, FDDI) · (more) |

.

ALGORITHM (valid for rpctime and rpc echoserver)

1. Write your own specification file “sample.x”
2. Use rpcgen to generate the files you need(i.e.,sample.h,sample_svc.c,and sample_clnt.c)

   ```
   $rpcgen –a  sample.x
   ```

3. Use rpcgen to generate sample server program

   ```
   $rpcgen –Ss sample.x  > sample_server.c
   ```

4. Use rpcgen to generate sample client program

   ```
   $rpcgen –Sc sample.x  > sample_client.c
   ```

5. Modify your sample server and sample client programs
6. Use cc to compile your server program(sample_server.c)

   ```
   $cc sample_server.c sample_svc.c  -o sample_server
   ```

7. Use cc to compile your client program(sample_client.c)

   ```
   $cc  sample_client.c sample_clnt.c  -o sample_client
   ```

8. Run your server program

   ```
   $./sample_server
   ```

   Or to,put the server to run in the background

   ```
   $./sample_server &
   ```

9. Run your client program

   ```
   $./sample_client <ipaddress>
   ```

10. Use rpcinfo to manage your RPC resources,see the main page of rpcinfo for details.

(a)display rpc day time

Remote procedure call

List of files will be created from .x file

```
rpcdt_client    rpcdt_client.c  rpcdt_clnt.c    rpcdt.h

rpcdt_server    rpcdt_server.c  rpcdt_svc.c     rpcdt.x
```

step1:$ vi rpcdt.x

```
program TIME_PROG{

version TIME_VERSION{

string timeofday(void)=1;

}=1;

}=0x31234567;
```

```
[it338@cc5 cnlab38lab]$ rpcgen -a rpcdt.x
```

```
/*This is sample code generated by rpcgen. * These are only templates and you can use them

 * as a guideline for developing your own functions.*/
```

step2:

```
[it338@cc5 cnlab38lab]$ vi rpcdt_server.c

#include "rpcdt.h"

char **

timeofday_1_svc(void *argp, struct svc_req *rqstp)
```

```
{
        static char * result;
        long t= time(NULL);
        result=ctime(&t);
        return &result;
}
[staff]$ cc rpcdt_server.c rpcdt_svc.c -o rpcdt_server
rpcdt_server.c: In function `timeofday_1_svc':
rpcdt_server.c:15: warning: assignment makes pointer from integer without a cast
[staff]$ ./rpcdt_server&
[3] 8218
```

## Implementation of day time client using rpc

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */
#include "rpcdt.h"
void
rpcdt_prog_1(char *host)
{
	CLIENT *clnt;
	char * *result_1;
	char *timeofday_1_arg;
#ifndef DEBUG
	clnt = clnt_create (host, RPCDT_PROG, RPCDT_VERSION, "udp");
	if (clnt == NULL) {
		clnt_pcreateerror (host);
		exit (1);
	}
#endif  /* DEBUG */
```

```
        result_1 = timeofday_1((void*)&timeofday_1_arg, clnt);

        if (result_1 == (char **) NULL) {

                clnt_perror (clnt, "call failed");

        }

printf("\n time of day service");

printf("\n today date and time is %s \n", result_1[0]);

#ifndef DEBUG

        clnt_destroy (clnt);

#endif   /* DEBUG */

}

int

main (int argc, char *argv[])

{

        char *host;



        if (argc < 2) {

                printf ("usage: %s server_host\n", argv[0]);

                exit (1);

        }

        host = argv[1];
        rpcdt_prog_1 (host);

exit (0);

}
```

**OUTPUT:**

```
[staff]$ cc rpcdt_client.c rpcdt_clnt.c -o rpcdt_client

[staff]$ ./rpcdt_client 192.168.2.150

 time of day service

 today date and time is Sat Apr 17 12:59:48 2010
```

**(b)Display rpc echo server**

```
$ vi echo_prog.x

program ECGI_PROG{

version ECHO_VERSION{

string echo(string)=1;

}=1;

}=0x31234567;

$rpcgen –a echo_prog
```

**$vi echo_prog_server.c**

```
/*

 * This is sample code generated by rpcgen. * These are only templates and you can use them

 * as a guideline for developing your own functions. */

#include "echo_prog.h"

#include<rpc/rpc.h>

char **

echo_1_svc(char **argp, struct svc_req *rqstp)

{

        static char * result;

   result=argp;

printf("\n server:%s\n",argp[0]);
// return ar;

        return &result;

}

[staff]$ cc echo_prog_server.c echo_prog_svc.c -o echo_prog_server

 [staff]$ ./echo_prog_server

 server:hello
```

**$vi echo_prog_client.c**

```
/* sample code generated by rpcgen.

 * These are only templates and you can use them

 * as a guideline for developing your own functions.*/

#include <stdio.h>

#include "echo_prog.h"

#include <rpc/rpc.h>

void

ecgi_prog_1(char *host)

{

        CLIENT *clnt;

        //char * *result_1;

        char * echo_1_arg;
```

```
      char temp[100];

      char *snd;

char *result;

#ifndef DEBUG

        clnt = clnt_create (host, ECGI_PROG, ECHO_VERSION, "udp");

if (clnt == NULL) {

        clnt_pcreateerror (host);

                exit (0);

        }

#endif  /* DEBUG */

/*      result_1 = echo_1(&echo_1_arg, clnt);

        if (result_1 == (char **) NULL) {

                clnt_perror (clnt, "call failed");

        }*/

printf("enter a string");

scanf("%s",temp);

snd=&temp[0];

result=echo_1(&snd,clnt);

printf("the message was %s\n",*result);

#ifndef DEBUG
```

```
        clnt_destroy (clnt);
#endif   /* DEBUG */
}
int
main (int argc, char *argv[])
{
        char *host;


        if (argc!= 2) {
                printf ("usage: %s server_host\n", argv[0]);
                exit (0);
        }
        host = argv[1];
        ecgi_prog_1 (host);
exit (1);
}
```

**OUTPUT:**

```
[staff]$ cc echo_prog_client.c echo_prog_clnt.c -o echo_prog_client
echo_prog_client.c: In function `ecgi_prog_1':
echo_prog_client.c:34: warning: assignment from incompatible pointer type
[staff]$ ./echo_prog_client 192.168.2.150
```

```
enter a string hello
```