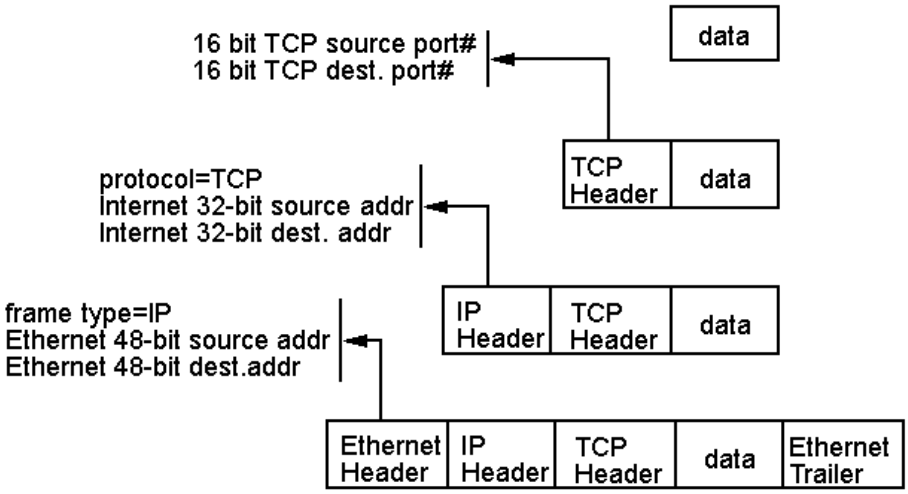
Internet Transport Layer Protocols

To date, we've compared the 7-layered OSI/ISO model to the 4-layered TCP/IP protocol suite.

We've recently focused on how TCP/IP delivers its packets - using 32-bit IPv4 addresses to deliver data, first to the *correct network*, and then to the *correct host* on that network. We've also focused on how the protocols are embedded, or *encapsulated*, within each other:



Pedantically, we could say that Ethernet delivers the IP packet to the *correct interface* on that host.

Port numbers

IP addresses, alone, are not enough as they only address hosts, and not individual operating system processes on those hosts.

From the perspective of any transport protocol, such as TCP (next), each arriving frame is further identified by a 16-bit positive *port number* that identifies the 'software end-point' to receive the payload.

One role of TCP is to *demultiplexed* each arriving segment to its corresponding communication end-point, using a port as an index.

Port numbers below 1024 are described as *reserved ports*, and on operating systems with distinct users and privilege levels, elevated privilege ('root' or 'administrator' access) is required to create a 'software end-point' *bound* to such ports.

The file */etc/services* on Linux and macOS, or *C:\Windows\System32\drivers\etc on Windows*, lists ports commonly used (worldwide), and ports in use for dedicated/local applications:

```
// Ports below 1024 are reserved

echo       7/tcp
ftp        21/tcp          # File transfer protocol
ssh        22/tcp          # SSH Remote Login Protocol
telnet     23/tcp          # Telnet
smtp       25/tcp          # Simple mail transfer protocol
finger     79/tcp
http       80/tcp          # Hypertext transfer protocol
pop3       110/tcp         # POP version 3
sunrpc     111/tcp         # RPC 4.0 portmapper TCP
nntp       119/tcp         # Network news transfer protocol
https      443/tcp         # Secured hypertext transfer protocol
exec       512/tcp         # Remote execution
login      513/tcp

// Ports above 1023 are not reserved, but may be pre-assigned:

ms-sql-s 1433/tcp          # Microsoft-SQL-Server
license  1702/tcp          # matlab/FLEXlm licence manager
nfs      2049/udp          # The network file system (NFS)
x11      6000/tcp          # the X Window System
```

The Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) transforms the 'raw' IP into a full-duplex reliable character stream [**Ref:**RFC 793].

TCP uses a 'well-understood' sliding window with selective-repeat protocol, and conveys a number of important fields in its TCP frame header:

What TCP/IP Provides to Applications

TCP/IP provides 6 major features:

1. *Connection orientation* - for two programs to employ TCP/IP, one program must first request a connection to the destination before communication may proceed.
2. *Reliable connection startup* - when two applications create a connection, both must agree to the new connection. No packets from previous, or ongoing, connections will interfere.
3. *Point-to-point communication* - each communication session has exactly 2 endpoints.
4. *Full-duplex communication* - once established, a single connection may be used for messages in both directions.

   This requires buffering at both each input and output 'end', and enables each application to continue with computation while data is being communicated.
5. *Stream interface* - from the application's viewpoint, data is sent and received as a continuous sequence of bytes.

   Applications need not (but may) communicate using fixed-sized records. Data may be written and read in blocks of arbritary size.
6. *Graceful connection shutdown* - TCP/IP guarantees to reliably deliver all 'pending' data once a connection is closed by one of the endpoints.

TCP/IP 3-way connection establishment and sequence numbers

A *three-way handshake* is employed in TCP's internal *open* sequence.

If machine A wishes to establish a connection with machine B, A transmits the following message:

```
A->B : SYN, ISN_a
```

This initial packet request has the *synchronize sequence number* bit (SSN) set in its header, and an initial 32-bit unsigned sequence number ISN_a.

B replies with:

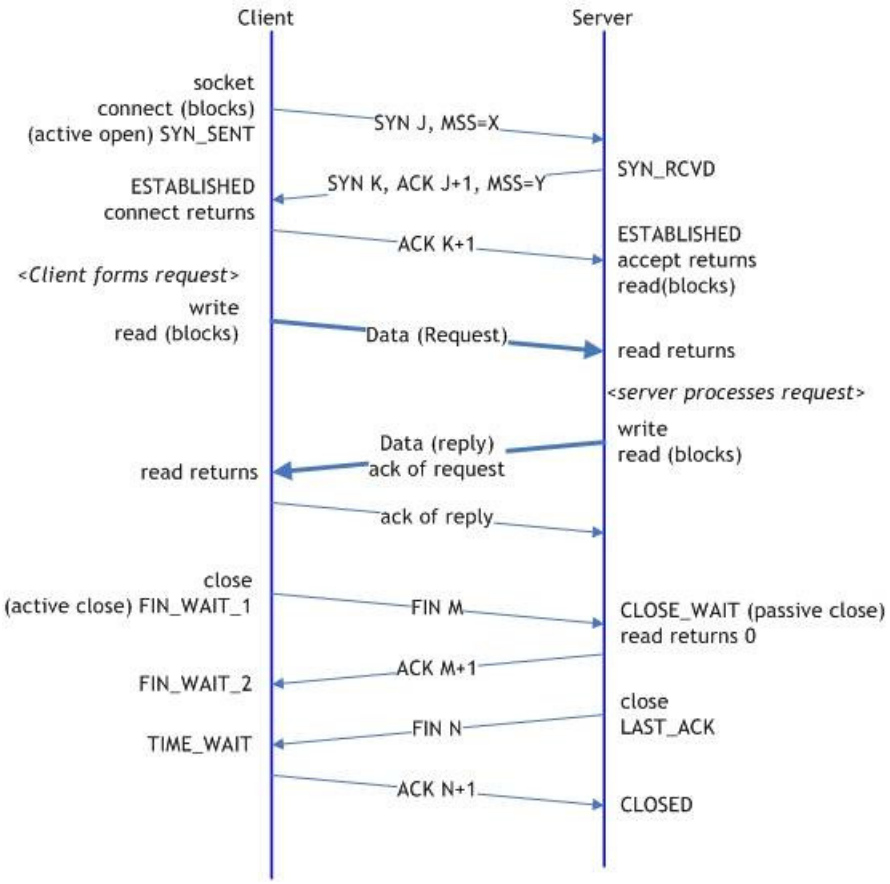
```
B->A : SYN, ISN_b, ACK(ISN_a)
```

to provide its own initial sequence number, ISN_b, and to acknowledge ISN_a.

A will finally acknowledge ISN_b with

```
A->B : ACK(ISN_b)
```

and the connection is established. May be of interest: the *Byzantine Generals' Problem*.



TCP/IP Retransmissions

TCP/IP is employed between processes physically tens of centimetres (i.e. nano-seconds) apart, as well as processes tens of thousands of kilometres (near a second) apart.

As TCP/IP uses a sliding window protocol, timeouts are employed to force re-transmissions. As there are so many destination hosts, what should be the timeout value?

To cope with the widely varying network delays, TCP maintains a dynamic estimate of the current *round trip time* (RTT) for each connection. Because the RTTs vary tremendously, TCP averages RTTs into a *smoothed* round trip time (SRTT) that minimizes the effects of unusually short or long RTTs.

$$SRTT = (\alpha \times SRTT) + ((1-\alpha) \times RTT)$$

where α is a smoothing factor that determines how much weight the new values are given. When α=1, the new value of RTT is ignored, when α=0 all previous values are ignored. Typically α is between 0.8 and 0.9.

The SRTT estimates the average round trip time. To also allow for queuing and transmission delays, TCP also calculates the *mean deviation* (MDEV) of the RTT from the measured value.

This is also smoothed:

$$SMDEV = (\alpha \times SMDEV) + ((1-\alpha) \times MDEV)$$
$$RTO = SRTT + 2 \times SMDEV$$

TCP/IP Congestion Control

Perhaps the most important, and certainly the most studied and 'tinkered with' aspect of TCP/IP is its congestion control.

TCP attempts to avoid *congestion collapse* by using end-to-end packet loss as the metric of congestion.

- The TCP receiver normally fills the *Window* field of an acknowledgment header to report how much additional buffer space (the receiver's *window size*) is available for further data.
- When a message is lost, the TCP sender could naively retransmit enough data to fill the receiver's buffers. Instead, TCP commences by sending a single packet. If an acknowledgment for this single packet returns, the sender next transmits two packets; if all of their acknowledgments return, up to four, and so on.

  In effect, the protocol grows (doubles) the sender's sliding window until packets are lost; it then restarts at 1.
- TCP/IP responds to congestion by backing-off quickly, and avoids further congestion by slowly increasing offered traffic.

In combination with its closely related *slow-start* algorithm for new connections, TCP is capable of both avoiding and recovering from most congestion.

Ref: RFC 2001.

Network Application Program Interfaces (APIs)

Dating back to early operating system implementations, applications attempted to provide a common framework to access both files and devices.

Calls to Unix `open()` return a *file descriptor* which is then used in calls to `read()` and `write()`.

It is preferable if the *application program interfaces* (API) to network I/O exhibit the same semantics as file, or stream, I/O, but this is difficult for a number of reasons:

- The typical *client-server* relationship is not symmetrical - each program must know what role it has to play.
- Network connections may be connection-oriented or connectionless. With a connectionless protocol there is nothing akin to `open()` since every network I/O operation *could* be with a different process on a different host.
- Identification is more important to networking than for file operations. Networking applications need to *verify peer processes* when accepting new connections.
- There are more associations to be made for network I/O than for file I/O:

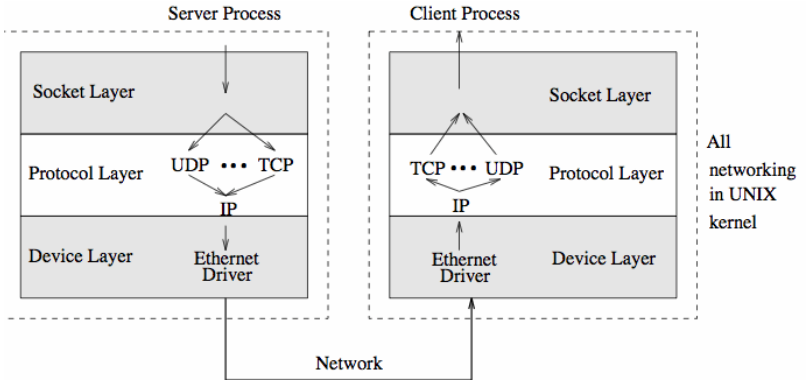
  { protocol, local-address, local-process, remote-address, remote-process }

- Many file I/O models presume all data is in a continuous data stream; this precludes networking applications working with variable length datagrams.

An Example Network API - Berkeley Sockets

Sockets are a generalization of the original Unix file system model. The most important difference is that the operating system binds file descriptors, *once*, to files and devices when they are opened. With sockets, applications can specify the destination *each time* they use the socket.

When sockets were first proposed (1982 in 4.1cBSD), it was unclear how significant TCP/IP would become. As a (beneficial) consequence, sockets have been designed to use many different protocols.



The current (kernel) socket implementation consists of three parts :

1. The socket layer provides the interface between user programs and the networking (via operating system system calls).
2. The protocol layer supports different protocols in use, such as TCP/IP, X.25, etc.
3. The device driver supports the physical devices such as Ethernet controllers.

Essential reading:

A Guide to Network Programming using Internet sockets, *by Brian "Beej" Hall*.

Domain Addressing

Legal combinations of protocols and drivers are specified when the kernel is configured.

For example, sockets that share common communication properties, such as naming conventions and protocol address formats, are grouped into *address families*.

The Linux file */usr/include/bits/socket.h* lists all supported *address families*.

```
#define AF_UNSPEC     0    // unspecified
#define AF_UNIX       1    // local to host (pipes, portals)
#define AF_INET       2    // internetwork: UDP, TCP, etc.
#define AF_IMPLINK    3    // arpanet imp addresses
#define AF_CCITT      10   // CCITT protocols, X.25 etc
#define AF_SNA        11   // IBM SNA
#define AF_DECnet     12   // DECnet
#define AF_APPLETALK 16    // Apple Talk

#define AF_NIT        17   // Network Interface Tap
#define AF_802        18   // IEEE 802.2, also ISO 8802
#define AF_OSI        19   // umbrella for all families used by OSI
#define AF_X25        20   // CCITT X.25 in particular
#define AF_GOSIP      22   // U.S. Government OSI
 .....
```

Processes communicate using the *client-server* paradigm.

A server process *listens* to a *socket*, one end of a bidirectional communication path and the client processes communicate with the server over another socket, the other end of the communication path.

The kernel maintains internal connections and routes data from client to server.

Establishing Sockets With OS System Calls

The socket mechanism requires several Unix system calls. The `socket()` call establishes an end point of a communications link.

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

    int family, type, protocol;
    int sd, socket(int, int, int);
     ...
    sd = socket(addr_family, type, protocol);
```

protocol is usually `0` to indicate the default for the family/type combination. The`socket()` system call returns a small integer, termed a *socket descriptor*, (akin to a file descriptor). The call may fail due to a request for an unknown protocol or when a request is made for a type without a supporting protocol.

	protocol	*local-addr, local-process*	*remote-addr, remote-process*
connection-orient server	`socket()`	`bind()`	`listen(), accept()`
connection-orient client	`socket()`	`connect()`	
connectionless server	`socket()`	`bind()`	`recvfrom()`
connectionless client	`socket()`	`bind()`	`sendto()`

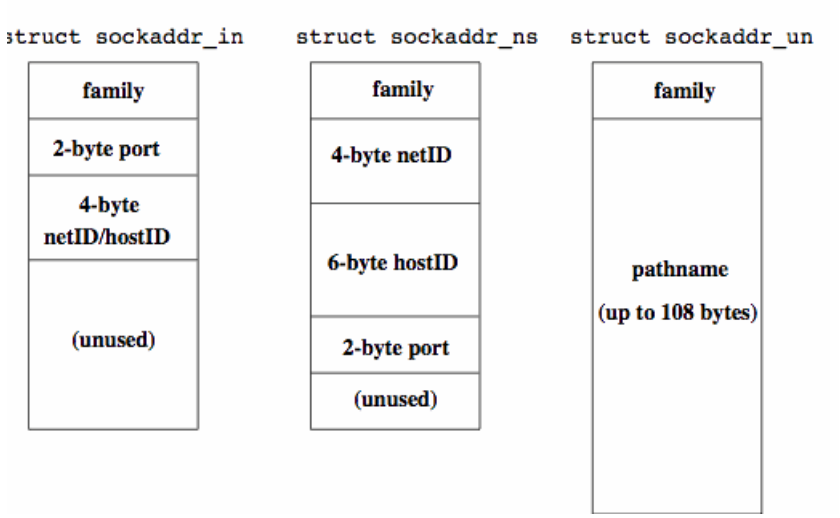
The `socket()` system call only instantiates *protocol* from the 5-tuple association.

Depending on whether the socket is being used in the client or server of either a connection-oriented or connectionless communication, different programs do different things next:

Naming Sockets

When initially created a socket is *unbound* (it has no addresses associated with it).

Communication cannot occur on an unbound socket - without a name for the process owning the socket, the kernel cannot demultiplex packets to the correct socket. The bind() routine provides an address (a name) to the local end of the socket.




```
// Socket address, UNIX style.
struct sockaddr {
     u_short sa_family;         // address family
     char    sa_data[108];      // up to 108 bytes of addr
};

// Socket address, internet style.
struct sockaddr_in {
     short   sin_family;        // AF_INET
     u_short sin_port;          // 16-bit port number
     struct  in_addr sin_addr;  // 32-bit netid/hostid
     char    sin_zero[8];       // unused
};

    ....
    bind(sd, socket_addr, sizeof(socket_addr));
    ....
```

Naming Sockets, *continued*

The related call `connect()` takes the same arguments but binds an address to the *remote end* of the socket.

For connectionless protocols, such as UDP/IP, the kernel caches the destination address associated with the socket.

Server processes *bind* address to sockets and 'advertise' their names to identify themselves to clients.

Connection Establishment

Servers accept connections from remote clients and cannot use `connect()` because they do not (usually) know the address of the remote client until the client has initiated a connection.

Applications use `listen()` and `accept()` to perform *passive* opens.

When a server arranges to accept data over a virtual circuit, the kernel must arrange to queue requests until they can be serviced.

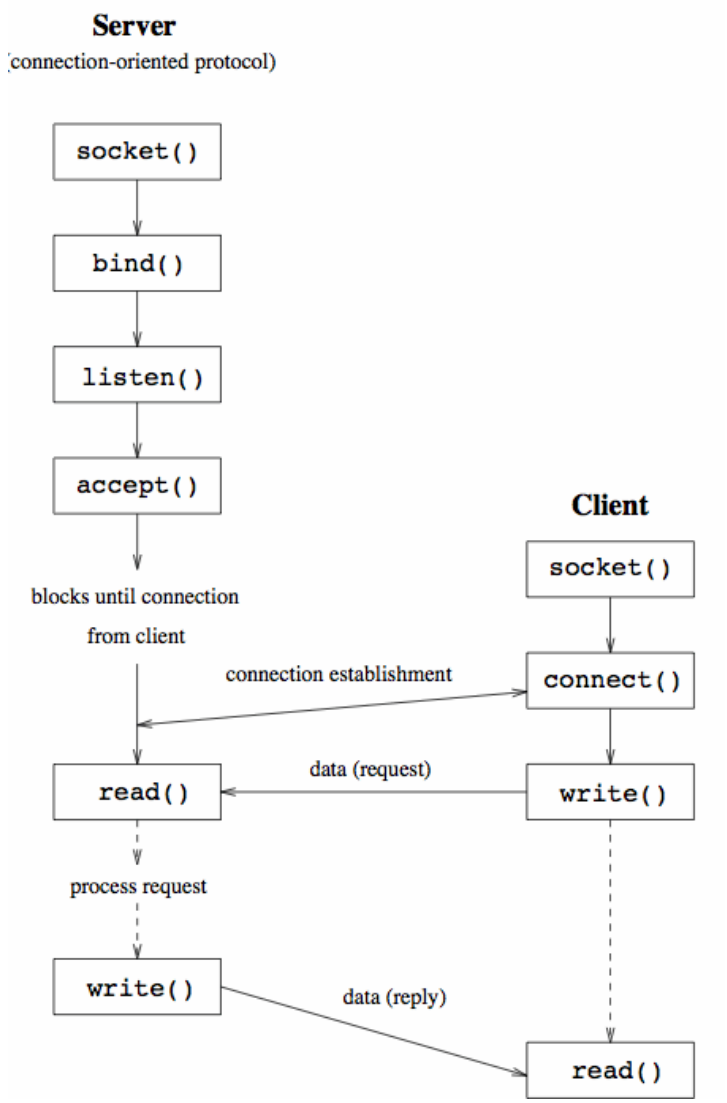
```
listen(sd, queue_length);
```

`listen()` only indicates that an application is *willing* to accept requests; applications call `accept()` to *accept* them.

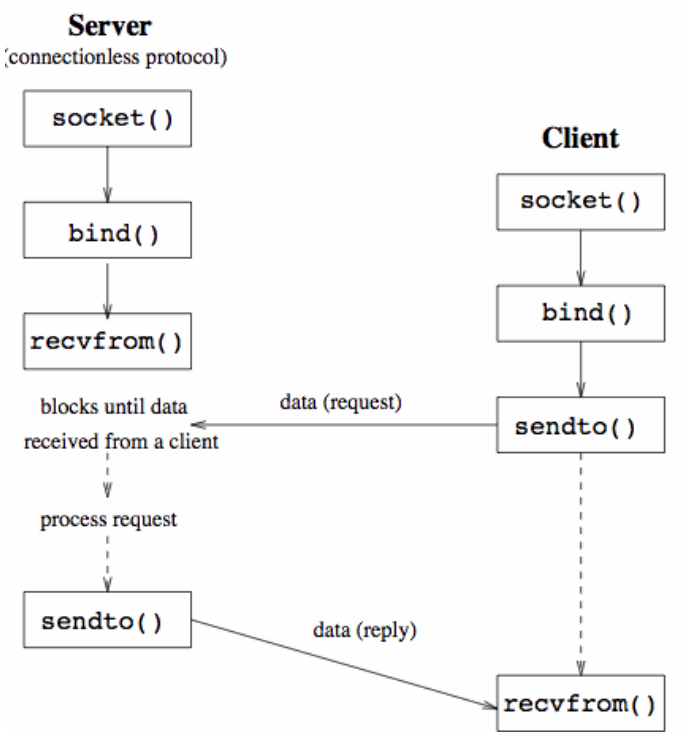
```
new_socket = accept(sd, from, fromlength);
```

When `accept()` returns, `from` contains the network address of the remote end of the socket, and `new_socket` is in a *connected* state.

System Call Sequences for Connection-oriented and Connectionless I/O



Connection-oriented I/O



Connectionless I/O

A Client Process in the Unix Domain (in C)

Consider a simple client process wishing to establish a connection with a server process in the Unix domain. When communicating within the Unix domain, the data frames never leave the single computer, and never get lost (other than on an extremely busy machine).

In this example, the client program sends commands to a 3D printer which is directly connected to the same computer. The client process simply connects to the server process and then writes the bytes to be printed to the socket *(note that this example is far from how print spooling works in practice!)*

```
#include <many-header-files.h>

int write_file_to_server(int sd, const char filenm[])
{
//  ENSURE THAT WE CAN OPEN PROVIDED FILE
    int fd  = open(filenm, O_RDWR, 0);

    if(fd >= 0) {
        char  buffer[1024];
        int   nbytes;

//  COPY BYTES FROM FILE-DESCRIPTOR TO SOCKET-DESCRIPTOR
        while( (nbytes = read(fd, buffer, sizeof(buffer)) ) ) {
            if(write(sd, buffer, nbytes) != nbytes) {
                close(fd);
                return 1;
            }
        }
        close(fd);
        return 0;
    }
    return 1;
}

int main(int argc, char *argv[])
{
//  ASK OUR OS KERNEL TO ALLOCATE RESOURCES FOR A SOCKET
    int sd = socket(AF_UNIX, SOCK_STREAM, 0);
    if(sd < 0) {
        perror(argv[0]);      // issue a standard error message
        exit(EXIT_FAILURE);
    }

//  FIND AND CONNECT TO THE SERVICE ADVERTISED WITH "THREEDsocket"
    if(connect(sd,"THREEDsocket",strlen("THREEDsocket")) == -1) {
        perror(argv[0]);      // issue a standard error message
        exit(EXIT_FAILURE);
    }
    write_file_to_server(sd, FILENM_OF_COMMANDS);
    shutdown(sd, SHUT_RDWR);
    close(sd);
    exit(EXIT_SUCCESS);
}
```

A Server Process in the Unix Domain (in C)

Now consider our server process which accepts streams of bytes (commands and contents) to be printed on our 3D-printer.

To avoid contention for the printer, and to possibly screen the requests, a single server performs the printing.

```
#include <many-header-files.h>

int main(int argc, char *argv[])
{
//  ASK OUR OS KERNEL TO ALLOCATE RESOURCES FOR A SOCKET
    int  sd = socket(AF_UNIX, SOCK_STREAM, 0);

    if(sd < 0) {
        perror(argv[0]);     // issue a standard error message
        exit(EXIT_FAILURE);
    }

//  ADVERTISE THE STRING "THREEDsocket" TO THE NEW SOCKET
    if(bind(sd, "THREEDsocket", strlen("THREEDsocket")) != 0) {
        perror(argv[0]);     // issue a standard error message
        exit(EXIT_FAILURE);
    }

//  ENQUEUE 5 NEW CLIENTS BEFORE REFUSING NEW CONNECTIONS
    listen(sd, 5);

    while(true) {
        struct    sockaddr sockaddr;
        socklen_t          fromlen = sizeof(sockaddr);

//  USE THE 'ADVERTISING' DESCRIPTOR TO ACCEPT NEW CONNECTIONS
        int        newsd   = accept(sd, &sockaddr, &fromlen);

        if(newsd >= 0) {
            read_file_for_printing(newsd);
            shutdown(newsd, SHUT_RDWR);
            close(newsd);
        }
    }
    exit(EXIT_SUCCESS);
    return 0;
}
```

A Remote Login Client (Internet Domain, in C)

Most operating systems, supporting internetworking using the Berkeley sockets API, also provide many functions to facilitate access to many commonly required resources - such as hostnames, protocol numbers, service numbers, etc. In an environment where many computers require access to consistent data, these API functions, themselves, may be configured to seek their information via the Internet.

```c
#include <many-header-files.h>

int main(int argc, char *argv[])
{
//  LOCATE INFORMATION ABOUT THE REQUIRED SERVICE (PROTOCOL AND PORT)
    struct servent     *sp = getservbyname("login", "tcp");

    if(sp == NULL) {
        fprintf(stderr,"rlogin: tcp/login: unknown service\n");
        exit(1);
    }

//  LOCATE INFORMATION ABOUT THE REQUIRED HOST (ITS IP ADDRESS)
    struct hostent     *hp = gethostbyname(argv[1]);

    if(hp == NULL) {
        fprintf(stderr,"rlogin: %s: unknown host\n",argv[1]);
        exit(2);
    }

//  ASK OUR OS KERNEL TO ALLOCATE RESOURCES FOR A SOCKET
    int  sd  = socket(AF_INET, SOCK_STREAM, 0);
    if(sd < 0) {
        perror("rlogin: socket");
        exit(3);
    }

//  INITIALIZE FILEDS OF A STRUCTURE USED TO CONTACT SERVER
    struct sockaddr_in server;

    memset(&server, 0, sizeof(server));
    memcpy(&server.sin_addr, hp->h_addr, hp->h_length);
    server.sin_family  = hp->h_addrtype;
    server.sin_port    = sp->s_port;

//  CONNECT TO SERVER
    if(connect(sd, &server, sizeof(server)) < 0) {
        perror("rlogin: connect");
        exit(4);
    }

    communicate_with_rlogin_server(sd);
    shutdown(sd, SHUT_RDWR);
    close(sd);
    exit(EXIT_SUCCESS);
}
```