Appendix M
Sockets: A Programmer's Introduction

William Stallings

Supplement to
Operating Systems, Eighth Edition
Pearson 2014
http://williamstallings.com/OperatingSystems/

The concept of sockets and sockets programming was developed in the 1980s in the Unix environment as the Berkeley Sockets Interface. In essence, a socket enables communications between a client and server process and may be either connection-oriented or connectionless. A socket can be considered an endpoint in a communication. A client socket in one computer uses an address to call a server socket on another computer. Once the appropriate sockets are engaged, the two computers can exchange data.

Typically, computers with server sockets keep a TCP or UDP port open, ready for unscheduled incoming calls. The client typically determines the socket identification of the desired server by finding it in a Domain Name System (DNS) database. Once a connection is made, the server switches the dialogue to a different port number to free up the main port number for additional incoming calls.

Internet applications, such as TELNET and remote login (rlogin) make use of sockets, with the details hidden from the user. However, sockets can be constructed from within a program (in a language such as C or Java), enabling the programmer to easily support networking functions and applications. The sockets programming mechanism includes sufficient semantics to permit unrelated processes on different hosts to communicate.

The Berkeley Sockets Interface is the de facto standard application programming interface (API) for developing networking applications, spanning a wide range of operating systems. The sockets API provides generic access to interprocess communications services. Thus, the sockets capability is ideally suited for students to learn the principles of protocols and distributed applications by hands-on program development.

The Sockets Application Program Interface (API) provides a library of functions that programmers can use to develop network aware applications. It has the functionality of identifying endpoints of the connection, establishing the communication, allowing messages to be sent, waiting for incoming messages, terminating the communication, and error handling. The operating system used and the programming language both determine the specific Sockets API.

We concentrate on only two of the most widely used interfaces – the Berkley Software Distribution Sockets (BSD) as introduced for UNIX, and its slight modification the Windows Sockets (WinSock) API from Microsoft.

This sockets material is intended for the C language programmer. (It provides external references for the C++, Visual Basic, and PASCAL languages.) The Windows operating system is in the center of our discussion. At the same time, topics from the original BSD UNIX specification are introduced in order to point out (usually minor) differences in the sockets specifications for the two operating systems. Basic knowledge of the TCP/IP and UDP network protocols is assumed. Most of the code would compile on both Windows and UNIX like systems.

We cover C language sockets exclusively, but most other programming languages, such as C++, Visual Basic, PASCAL, etc., can take advantage of the Winsock API, as well. The only requirement is that the language has to recognize dynamic link libraries (DLLs). In a 32-bit Windows environment you will need to import the `wsock32.lib` to take advantage of the WinSock API. This library has to be linked, so that at run time the dynamic link library `wsock32.dll` gets loaded. `wsock32.dll` runs over the TCP/IP stack. Windows NT, Windows 2000, and Windows 95 include the file `wsock32.dll` by default. When you create your executables, if you link with `wsock32.lib` library, you will implicitly link the `wsock32.dll` at run time, without adding lines of code to your source file.

The Web site for this book provides links to useful Sockets Web sites.

M.1   SOCKETS, SOCKET DESCRIPTORS, PORTS, AND CONNECTIONS

Sockets are endpoints of communication referred to by their corresponding socket descriptors, or natural language words describing the socket's association with a particular machine or application (e.g., we will refer to a server socket as `server_s`). A connection (or socket pair) consists of the pair of IP addresses that are communicating with each other, as well a pair of port numbers, where a port number is a 32-bit positive integer usually denoted in its decimal form. Some destination port numbers are well known and indicate the type of service being connected to.

For many applications, the TCP/IP environment expects that applications use well-known ports to communicate with each other. This is done so that client applications assume that the corresponding server application is listening on the well-known port associated with that application. For example, the port number for HTTP, the protocol used to transfer HTML pages across the World Wide Web, is TCP port 80. By default, a Web browser will attempt to open a connection on the destination host's TCP port 80 unless another port number is specified in the URL (such as 8000 or 8080).

A *port* identifies a connection point in the local stack (i.e., port number 80 is typically used by a Web server). A *socket* identifies an IP address and port number pair (i.e. , port 192.168.1.20:80 would be the Web server port on host 192.168.1.20. The two together are considered a socket.). A *socket pair* identifies all four components (source address and port, and destination address and port). Since well-known ports are unique, they are sometimes used to refer to a specific application on any host that might be running the application. Using the word socket, however, would imply a specific application on some specific host. Connection, or a *socket pair*, stands for the sockets connection between two specific systems that are communicating. TCP allows multiple simultaneous connections involving the same local port number as long as the remote IP addresses or port numbers are different for each connection.

Port numbers are divided into three ranges:

- Ports 0 through 1023 are well known. They are associated with services in a static manner. For example, HTTP servers would always accept requests at port 80.
- Port numbers from1024 through 49151 are registered. They are used for multiple purposes.
- Dynamic and private ports are those from 49152 through 65535 and services should not be associated with them.

In reality, machines start assigning dynamic ports starting at 1024. If you are developing a protocol or application that will require the use of a link, socket, port, protocol, etc., please contact the Internet Assigned Numbers Authority (IANA) to receive a port number assignment. The IANA is located at and operated by the Information Sciences Institute (ISI) of the University of Southern California. The Assigned Numbers request for comments (RFC) published by IANA is the official specification that lists port assignments. You can access it at URL http://www.iana.org/assignments/port-numbers.

On both UNIX and Windows, the `netstat` command can be used to check the status of all active local sockets. Figure M.1 is a sample netstat output.

M.2 THE CLIENT/SERVER MODEL OF COMMUNICATION

A socket application consists of code, executed on both communication ends. The program initiating transmission is often referred to as the client. The server, on the other hand, is a program that passively awaits incoming connections from remote clients. Server applications typically load during system startup and actively listen for incoming connections on their well-known port. Client applications will then attempt to connect to the server, and a TCP exchange will then take place. When the session is complete, usually the client will be the one to terminate the connection. Figure 2 depicts the basic model of stream-based (or TCP/IP sockets) communication.

Running a Sockets Program on a Windows Machine Not Connected to a Network

As long as TCP/IP is installed on one machine, you can execute both the server and client code on it. (If you do not have the TCP/IP protocol stack installed, you can expect socket operations to throw exceptions such as `BindException`, `ConnectException`, `ProtocolException`, `SocketException`, etc.) You will have to use `localhost` as the hostname or `127.0.0.1` as the IP address.

Running a Sockets Program on a Windows Machine Connected to a Network, When Both Server and Client Reside on the Same Machine

In such a case you will be communicating with yourself. It is important to know whether your machine is attached to an Ethernet or communicates with the network through a telephone modem. In the first case you will have an IP address assigned to your machine, without efforts on your part. When communicating via a modem, you need to dial in, grab an IP address, and then be able to "talk to yourself." In both cases you can find out the IP address of the machine you are using with the `winipcfg` command for Win9X, and `ipconfig` for WinNT/2K and UNIX.

M.3 SOCKETS ELEMENTS

Socket Creation

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

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

- *domain* is AF_UNIX, AF_INET, AF_OSI, etc. AF_INET is for communication on the internet to IP addresses. We will only use AF_INET.
- *type* is either SOCK_STREAM (TCP, connection oriented, reliable), or SOCK_DGRAM (UDP, datagram, unreliable), or SOCK_RAW (IP level).
- *protocol* specifies the protocol used. It is usually 0 to say we want to use the default protocol for the chosen domain and type. We always use 0.

If successful, socket() returns a socket descriptor, which is an integer, and –1 in the case of a failure. An example call:

```
if ((sd = socket(AF_INET, SOCK_DGRAM, 0) < 0)
 {
      printf(socket() failed.);
    exit(1);
  }
```

The Socket Address

The structures to store socket addresses as used in the domain AF_INET:

```
struct in_addr {
   unsigned long s_addr;
};
```

in_addr just provides a name (s_addr) for the C language type to be associated with IP addresses.

```
struct sockaddr_in {
    unsigned short       sin_family; // AF_INET identifiers
    unsigned short       sin_port;   // port number,
                                    // if 0 then kernel chosen
    struct in_addr sin_addr;   // IP address
       // INADDR_ANY refers to the IP
       // addresses of the current host

    char           sin_zero[8]; // Unused, always zero
};
```

Both local and remote addresses will be declared as a sockaddr_in structure. Depending on this declaration, sin_addr will represent a local or remote IP address. (On a UNIX like system, you need to include the file <netinet/in.h> for both structures.)

Bind to a local port

```
#define  WIN      // WIN for Winsock and BSD for BSD sockets

#ifdef WIN
   …
#include <windows.h>    // for all Winsock functions
…
#endif

#ifdef BSD
```

```
…
#include <sys/types.h>
#include <sys/socket.h> // for struct sockaddr
…
#endif

int bind(int local_s, const struct sockaddr *addr, int addrlen);
```

- *local_s* is a socket descriptor of the local socket, as created by the socket()function;
- *addr* is a pointer to the (local) address structure of this socket;
- *addrlen* is the length (in bytes) of the structure referenced by *addr*.

`bind()` returns the integer 0 on success, and –1 on failure. After a call to `bind()`, a local port number is associated with the socket, but no remote destination is yet specified.

An example call:

```
struct sockaddr_in name;
...
name.sin_family = AF_INET;          // use the internet domain
name.sin_port = htons(0);           // kernel provides a port
name.sin_addr.s_addr = htonl(INADDR_ANY); // use all IPs of host

if ( bind(local_socket, (struct sockaddr *)&name, sizeof(name)) != 0)
   // print error and exit
```

A call to bind is optional on the client side, but it is required on the server side. After `bind()` is called on a socket, we can retrieve its address structure, given the socket file descriptor, by using the function `getsockname()`.

Data representation and byte ordering

Some computers are big endian. This refers to the representation of objects such as integers within a word. A big endian machine stores them in the expected way: the high byte of an integer is stored in the leftmost byte, while the low byte of an integer is stored in the rightmost byte. So the number $5 \times 2^{16} + 6 \times 2^{8} + 4$ would be stored as:

Big endian representation		5	6	4
Little endian representation	4	6	5	
Memory (byte) address	0	1	2	3

As you can see, reading a value of the wrong word size will result in an incorrect value; when done on big endian architecture, on a little endian machine it can sometimes return the correct result. The big endian ordering is somewhat more natural to humans, because we are used to reading numbers from left to right.

A Sun Sparc is a big endian machine. When it communicates with an i-386 PC (which is a little endian), the following discrepancy will exist: The i-386 will interpret $5 \times 2^{16} + 6 \times 2^{8} + 4$ as $4 \times 2^{16} + 6 \times 2^{8} + 5$. To avoid this situation from occurring, the TCP/IP protocol defines a machine independent standard for byte order – network byte ordering. In a TCP/IP packet, the first transmitted data is the most significant byte. Because big endian refers to storing the most significant byte in the lowest memory address, which is the address of the data, TCP/IP defines network byte order as big endian.

Winsock uses network byte order for various values. The functions `htonl()`, `htons()`, `ntohl()`, `ntohs()` ensure that the proper byte order is being used in Winsock calls, regardless of whether the computer normally uses little endian or big endian ordering.

The following functions are used to convert from host to network ordering before transmission, and from network to host form after reception:

- unsigned long *htonl*(unsigned long *n*) - host to network conversion of a 32-bit value;
- unsigned short *htons*(unsigned short *n*) - host to network conversion of a 16-bit value;
- unsigned long *ntohl*(unsigned long *n*) - network to host conversion of a 32-bit value;
- unsigned short *ntohs*(unsigned short *n*) - network to host conversion of a 16-bit value.

Connecting a socket

A remote process is identified by an IP address and a port number. The `connect()` call evoked on the local site attempts to establish the connection to the remote destination. It is required in the case of connection oriented communication such as stream-based sockets (TCP/IP). Sometimes we call `connect()` on datagram sockets, as well. The reason is that this stores the destination address locally, so that we do not need to specify the destination address every time when we send datagram message and thus can use the `send()` and `recv()` system calls instead of `sendto()` and `recvfrom()`. Such sockets, however, cannot be used to accept datagrams from other addresses.

```
#define  WIN      // WIN for Winsock and BSD for BSD sockets

#ifdef WIN
```

```
  #include <windows.h>      // Needed for all Winsock functions
#endif

#ifdef BSD
  #include <sys/types.h>    // Needed for system defined identifiers
  #include <netinet/in.h>   // Needed for internet address structure
#include <sys/socket.h>   // Needed for socket(), bind(), etc...
#endif

int connect(int local_s, const struct  sockaddr *remote_addr, int
rmtaddr_len)
```

- *local_s* is a local socket descriptor;
- *remote_addr* is a pointer to protocol address of other socket;
- *rmtaddr_len* is the length in bytes of the address structure.

Returned is an integer 0 (on success). The Windows connect function returns a non-zero value to indicate an error, while the UNIX connection function returns a negative value in such case.

An example call:

```
#define  PORT_NUM    1050         // Arbitrary port number

struct sockaddr_in   serv_addr;    // Server Internet address
int                 rmt_s;        // Remote socket descriptor

// Fill-in the server (remote) socket's address information and connect
// with the listening server.
server_addr.sin_family      = AF_INET;           // Address family to use
server_addr.sin_port        = htons(PORT_NUM);    // Port num to use
server_addr.sin_addr.s_addr = inet_addr(inet_ntoa(address)); // IP
address

if (connect(rmt_s,(struct sockaddr *)&serv_addr, sizeof(serv_addr)) !=
0)
  // print error and exit
```

The `gethostbyname()` function call

The function gethostbyname() is supplied a host name argument and returns NULL in case of failure, or a pointer to a struct hostent instance – on success. It gives information about the host names, aliases, and IP addresses. This information is obtained from the DNS or a local configuration database. The getservbyname() will determine the port number associated with a named service. If a numeric value is supplied instead, it is converted directly to binary and used as a port number.

```
#define struct  hostent {
      char    *h_name;   // official name of host
      char    **h_aliases;       // null terminated list of alias names
                          // for this host
      int     h_addrtype;        // host address type, e.g. AF_INET
      int     h_length;   // length of address structure
      char    **h_addr_list;     // null terminated list of addresses
                          // in network byte order
};
```

Note that `h_addr_list` refers to the IP address associated with the host.

```
#define  WIN      // WIN for Winsock and BSD for BSD sockets

#ifdef WIN
  #include <windows.h>       // for all Winsock functions
#endif

#ifdef BSD
#include <netdb.h>        // for struct hostent
#endif

 struct hostent *gethostbyname(const char *hostname);
```

Other functions that can be used to find hosts, services, protocols, or networks are: `getpeername()`, `gethostbyaddr()`, `getprotobyname()`, `getprotobynumber()`, `getprotoent()`, `getservbyname()`, `getservbyport(), getservent(), getnetbyname(), getnetbynumber(), getnetent().`

An example call:

```
#ifdef BSD
…
#include <sys\types.h> // for caddr_t type
…
#endif

#define  SERV_NAME   somehost.somecompany.com
#define  PORT_NUM    1050           // Arbitrary port number
#define h_addr h_addr_list[0]        // To hold host Internet address

…

struct sockaddr_in   myhost_addr;    // This Internet address
struct hostent       *hp;           // buffer information about remote host
int                  rmt_s;        // Remote socket descriptor


// UNIX specific part
bzero( (char *)&myhost_addr, sizeof(myhost_addr) );

// Winsock specific
memset( &myhost_addr, 0, sizeof(myhost_addr) );

// Fill-in the server (remote) socket's address information and connect
// with the listening server.
myhost_addr.sin_family      = AF_INET;           // Address family to use
myhost_addr.sin_port        = htons(PORT_NUM);    // Port num to use


if (hp = gethostbyname(MY_NAME)== NULL)
   // print error and exit

// UNIX specific part
bcopy(hp->h_name, (char *)&myhost_addr.sin_addr, hp->h_length );

// Winsock specific
```

```
memcpy( &myhost_addr.sin_addr, hp->h_addr, hp->h_length );

if(connect(rmt_s,(struct sockaddr *)&myhost_addr,
sizeof(myhost_addr))!=0)
  // print error and exit
```

The UNIX function `bzero()` zeroes out a buffer of specified length. It is one of a group of functions for dealing with arrays of bytes. `bcopy()` copies a specified number of bytes from a source to a target buffer. `bcmp()` compares a specified number of bytes of two byte buffers. The UNIX `bzero()` and `bcopy()` functions are not available in Winsock, so the ANSI functions `memset()` and `memcpy()` have to be used instead.

An example sockets program to get a host IP address for a given host name:

```

#define  WIN      // WIN for Winsock and BSD for BSD sockets

#include <stdio.h>          // Needed for printf()
#include <stdlib.h>         // Needed for exit()
#include <string.h>         // Needed for memcpy() and strcpy()
#ifdef WIN
  #include <windows.h>      // Needed for all Winsock stuff
#endif
#ifdef BSD
  #include <sys/types.h>    // Needed for system defined identifiers.
  #include <netinet/in.h>   // Needed for internet address structure.
  #include <arpa/inet.h>    // Needed for inet_ntoa.
  #include <sys/socket.h>   // Needed for socket(), bind(), etc...
  #include <fcntl.h>
  #include <netdb.h>
#endif

void main(int argc, char *argv[])
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1);       // Stuff for WSA
functions
WSADATA wsaData;                              // Stuff for WSA functions
#endif

struct hostent *host;            // Structure for gethostbyname()
struct in_addr  address;          // Structure for Internet address
char            host_name[256];  // String for host name

if (argc != 2)
{
printf(*** ERROR - incorrect number of command line arguments \n);
printf(            usage is 'getaddr host_name' \n);
exit(1);
}

#ifdef WIN
// Initialize winsock
WSAStartup(wVersionRequested, &wsaData);
#endif

// Copy host name into host_name
strcpy(host_name, argv[1]);

// Do a gethostbyname()
printf(Looking for IP address for '%s'... \n, host_name);
host = gethostbyname(host_name);

// Output address if host found
if (host == NULL)
printf(  IP address for '%s' could not be found \n, host_name);
else
{
memcpy(&address, host->h_addr, 4);
printf(  IP address for '%s' is %s \n, host_name, inet_ntoa(address));
}

#ifdef WIN
// Clean up winsock
WSACleanup();
#endif
}
```

Listening for an incoming client connection

The `listen()` function is used on the server in the case of connection-oriented communication to prepare a socket to accept messages from clients. It has the prototype:

```
int listen(int sd, int qlen);
```

- *sd* is a socket descriptor of a socket after a bind() call
- *qlen* specifies the maximum number of incoming connection requests that can wait to be processed by the server while the server is busy.

The call to listen()returns an integer: 0 on success, and –1 on failure. For example:

```
if (listen(sd, 5) < 0) {
      // print error and exit
```

Accepting a connection from a client

The `accept()` function is used on the server in the case of connection oriented communication (after a call to `listen()`) to accept a connection request from a client.

```
#define  WIN      // WIN for Winsock and BSD for BSD sockets

#ifdef WIN
   …
#include <windows.h>    // for all Winsock functions
…
#endif
```

```
#ifdef BSD
…
#include <sys/types.h>
#include <sys/socket.h> // for struct sockaddr
…
#endif

 int accept(int server_s, struct sockaddr * client_addr, int * clntaddr_len)
```

- *server_s* is a socket descriptor the server is listening on
- *client_addr* will be filled with the client address
- *clntaddr_len* contains the length of the client address structure.

The `accept()` function returns an integer representing a new socket (–1 in case of failure).

Once executed, the first queued incoming connection is accepted, and a new socket with the same properties as *`sd`* is created and returned. It is the socket that the server will use from now on to communicate with this client. Multiple successful calls to `connect()` will result in multiple new sockets returned.

An example call:

```
struct sockaddr_in client_addr;
int server_s, client_s, clntaddr_len;
   ...
if ((client_s = accept(server_s, (struct sockaddr *)&client_addr,
&clntaddr_len) < 0)
   // print error and exit

// at this stage a thread or a process can take over and handle
// communication with the client
```

Successive calls to accept on the same listening socket return different connected sockets. These connected sockets are multiplexed on the same port of the server by the running TCP stack functions.

Sending and Receiving messages on a socket

We will present only four function calls in this section. There are, however, more than four ways to send and receive data through sockets. Typical functions for TCP/IP sockets are send() and recv().

```
int send(int socket, const void *msg, unsigned int msg_length, int
flags);
int recv(int socket, void *rcv_buff, unsigned int buff_length, int
flags);
```

- *socket* is the local socket used to send and receive.
- *msg* is the pointer to a message
- *msg_length* is the message length
- *rcv_buff* is a pointer to the receive buffer
- *buff_length* is its length
- *flags* changes the default behavior of the call.

For example, a particular value of flags will be used to specify that the message is to be sent without using local routing tables (they are used by default).

Typical functions for UDP sockets are:

```
int sendto(int socket, const void *msg, unsigned int msg_length, int
flags,
                    struct sockaddr *dest_addr, unsigned int
addr_length);
int recvfrom(int socket, void *rcv_buff, unsigned int buff_length,
int flags,
                struct sockaddr *src_addr, unsigned int addr_length);
```

Most parameters are the same as for send() and recv(), except *dest_addr / src_addr* and *addr_length*. Unlike with stream sockets, datagram callers of sendto() need to be informed of the destination address to send the message to, and callers of recvfrom() need to distinguish between different sources sending datagram messages to the caller. We provide code for TCP/IP and UDP client and server applications in the following sections, where you can find the sample calls of all four functions.

Closing a socket

The prototype:

```
int closesocket(int sd); // Windows prototype
int close(int fd);       // BSD UNIX prototype
```

fd and *sd* are a file descriptor (same as socket descriptor in UNIX) and a socket descriptor.

When a socket on some reliable protocol, such as TCP/IP is closed, the kernel will still retry to send any outstanding data, and the connection enters a `TIME_WAIT` state (see Figure 1). If an application picks the same port number to connect to, the following situation can occur. When this remote application calls `connect()`, the local application assumes that the existing connection is still active and sees the incoming connection as an attempt to duplicate an existing connection. As a result, `[WSA]ECONNREFUSED` error is returned. The operating system keeps a reference counter for each active socket. A call to `close()` is essentially decrementing this counter on the argument socket. This is important to keep in mind when we are using the same socket in multiple processes. We will provide a couple of example calls in the code segments presented in the next subsections.

Report errors

All the preceding operations on sockets can exhibit a number of different failures at execution time. It is considered a good programming practice to report the returned error. Most of these errors are designed to assist the developer in the debugging process, and some of them can be displayed to the user, as well. In a Windows environment all of the returned errors are defined in `winsock.h`. On an UNIX-like system, you can find these definitions in `socket.h`. The Windows codes are computed by adding 10000 to the original BSD error number and adding the prefix `WSA` in front of the BSD error name. For example:

Windows name	BSD name	Windows value	BSD value
`WSAEPROTOTYPE`	`EPROTOTYPE`	10041	41

There are a few Windows-specific errors not present in a UNIX system:

`WSASYSNOTREADY`	*10091*	*Returned by `WSAStartup()` indicating that the network subsystem is unusable.*
`WSAVERNOTSUPPORTED`	*10092*	*Returned by `WSAStartup()` indicating that the Windows Sockets DLL cannot support this app.*
`WSANOTINITIALISED`	*10093*	*Returned by any function except `WSAStartup()`, when a successful `WSAStartup()` has not yet been performed.*

An example error-catching source file, responsible for displaying an error and exiting:

```
#ifdef WIN
#include    <stdio.h>   // for fprintf()
#include    <winsock.h> // for WSAGetLastError()
#include    <stdlib.h>  // for exit()
#endif

#ifdef  BSD
#include  <stdio.h>    // for fprintf() and perror()
#include  <stdlib.h>   // for exit()
#endif


void catch_error(char * program_msg)
{
char  err_descr[128];   // to hold error description
int err;

err = WSAGetLastError();

// record the winsock.h error description
if (err == WSANO_DATA)
    strcpy(err_descr, WSANO_DATA (11004) Valid name, no data record of
requested type.);
if (err == WSANO_RECOVERY)
    strcpy(err_descr, WSANO_RECOVERY (11003) This is a non-recoverable
error.);
if (err == WSATRY_AGAIN)

…
 fprintf(stderr,%s: %s\n, program_msg, err_descr);
 exit(1);
}
```

You can extend the list of errors to be used in your Winsock application by looking at URL http://www.sockets.com.

Example TCP/IP Client Program (Initiating Connection)

This client program is designed to receive a single message from a server (lines 39 - 41) and then terminate itself (lines 45 -56). It sends a confirmation to the server after the message is received (lines 42 - 44).

```
#define  WIN  // WIN for Winsock and BSD for BSD sockets

#include      // Needed for printf()
#include      // Needed for memcpy() and strcpy()
#ifdef WIN
#include      // Needed for all Winsock stuff
#endif
#ifdef BSD
#include      // Needed for system defined identifiers.
#include      // Needed for internet address structure.


#include      // Needed for socket(), bind(), etc...
#include      // Needed for inet_ntoa()
#include
#include
#endif

#define  PORT_NUM          1050     // Port number used at the server
#define  IP_ADDR 131.247.167.101 // IP address of server (***
HARDWIRED ***)

void main(void)
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1);       // WSA functions
WSADATA wsaData;                              // WSA functions
#endif

unsigned int         server_s;        // Server socket descriptor
struct sockaddr_in   server_addr;     // Server Internet address
char                 out_buf[100];    // 100-byte output buffer for data
char                 in_buf[100];     // 100-byte input buffer for data

#ifdef WIN
// Initialize Winsock
WSAStartup(wVersionRequested, &wsaData);
#endif

// Create a socket
server_s = socket(AF_INET, SOCK_STREAM, 0);
// Fill-in the server socket's address and do a connect with
// the listening server.  The connect() will block.
Server_addr.sin_family      = AF_INET;      // Address family
Server_addr.sin_port        = htons(PORT_NUM);    // Port num
Server_addr.sin_addr.s_addr = inet_addr(IP_ADDR); // IP address
Connect(server_s, (struct sockaddr *)&server_addr, sizeof(server_addr));

// Receive from the server
recv(server_s, in_buf, sizeof(in_buf), 0);
printf(Received from server... data = '%s' \n, in_buf);

// Send to the server
strcpy(out_buf, Message -- client to server);
send(server_s, out_buf, (strlen(out_buf) + 1), 0);

// Close all open sockets
#ifdef WIN
closesocket(server_s);
#endif
#ifdef BSD
close(server_s);
#endif

#ifdef WIN
// Clean up winsock
WSACleanup();
#endif
}
```

Example TCP/IP server program (passively awaiting connection)

All that the following server program does is serving a message to client running on another host. It creates one socket in line 37 and listens for a single incoming service request from the client through this single socket. When the request is satisfied, this server terminates (lines 62-74).

```
#define  WIN              // WIN for Winsock and BSD for BSD sockets

#include <stdio.h>          // Needed for printf()
#include <string.h>         // Needed for memcpy() and strcpy()
#ifdef WIN
  #include <windows.h>      // Needed for all Winsock calls
#endif
#ifdef BSD
  #include <sys/types.h>    // Needed for system defined identifiers.
  #include <netinet/in.h>   // Needed for internet address structure.
  #include <sys/socket.h>   // Needed for socket(), bind(), etc...
  #include <arpa/inet.h>    // Needed for inet_ntoa()
  #include <fcntl.h>
  #include <netdb.h>
#endif

#define  PORT_NUM   1050        // Arbitrary port number for the server
#define  MAX_LISTEN    3        // Maximum number of listens to queue

void main(void)
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1);       // for WSA functions
WSADATA wsaData;                              // for WSA functions
#endif

unsigned int         server_s;        // Server socket descriptor
struct sockaddr_in   server_addr;     // Server Internet address
unsigned int         client_s;        // Client socket descriptor
struct sockaddr_in   client_addr;     // Client Internet address
struct in_addr       client_ip_addr;  // Client IP address
int                  addr_len;        // Internet address length
char                 out_buf[100];    // 100-byte output buffer for data
char                 in_buf[100];     // 100-byte input buffer for data

#ifdef WIN
// Initialize Winsock
WSAStartup(wVersionRequested, &wsaData);
#endif

// Create a socket
//   - AF_INET is Address Family Internet and SOCK_STREAM is streams
server_s = socket(AF_INET, SOCK_STREAM, 0);
```

```
// Fill-in my socket's address information and bind the socket
//   - See winsock.h for a description of struct sockaddr_in
server_addr.sin_family      = AF_INET;            // Address family to use
server_addr.sin_port        = htons(PORT_NUM);    // Port number to use
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // Listen on any IP
addr.
bind(server_s, (struct sockaddr *)&server_addr, sizeof(server_addr));

// Listen for connections (queueing up to MAX_LISTEN)
listen(server_s, MAX_LISTEN);

// Accept a connection.  The accept() will block and then return with
// client_addr filled-in.
addr_len = sizeof(client_addr);
client_s = accept(server_s, (struct sockaddr *)&client_addr, &addr_len);

// Copy the four-byte client IP address into an IP address structure
//   - See winsock.h for a description of struct in_addr
memcpy(&client_ip_addr, &client_addr.sin_addr.s_addr, 4);

// Print an informational message that accept completed
printf(Accept completed!!!  IP address of client = %s  port = %d \n,
inet_ntoa(client_ip_addr), ntohs(client_addr.sin_port));

// Send to the client
strcpy(out_buf, Message -- server to client);
send(client_s, out_buf, (strlen(out_buf) + 1), 0);

// Receive from the client
recv(client_s, in_buf, sizeof(in_buf), 0);
printf(Received from client... data = '%s' \n, in_buf);

// Close all open sockets
#ifdef WIN
closesocket(server_s);
closesocket(client_s);
#endif
#ifdef BSD
close(server_s);
close(client_s);
#endif

#ifdef WIN
```

```
// Clean-up Winsock
WSACleanup();
#endif
}
```

This is not a very realistic implementation. More often server applications will contain some indefinite loop and be able to accept multiple requests. The preceding code can be easily converted into such more realistic server by inserting lines 46-61 into a loop in which the termination condition is never satisfied (e.g., `while(1){...}`). Such servers will create one permanent socket through the `socket()` call (line 37), while a temporary socket gets spun off every time when a request is accepted (line 49). In this manner each temporary socket will be responsible of handling a single incoming connection. If a server gets killed eventually, the permanent socket will be closed, as will each of the active temporary sockets. The TCP implementation determines when the same port number will become available for reuse by other applications. The status of such port will be in TIME-WAIT state for some predetermined period of time, as shown in Figure 1 for port number 1234.

M.4 STREAM AND DATAGRAM SOCKETS

When sockets are used to send a connection oriented, reliable stream of bytes across machines, they are of SOCK_STREAM type. As we previously discussed, in such cases sockets have to be connected before being used. The data are transmitted through a bidirectional stream of bytes and are guaranteed to arrive in the order they were sent.

Sockets of SOCK_DGRAM type (or datagram sockets) support a bidirectional flow of data, as well, but data may arrive out of order, and possibly duplicated (i.e., it is not guaranteed to be arriving in sequence or to be unique). Datagram sockets also do not provide reliable service since they can fail to arrive at all. It important to note, though, that the data record boundaries are preserved, as long as the records are no longer than the receiver could handle. Unlike stream sockets, datagram sockets are connectionless; hence they do not need to be connected before being used. Figure 3 shows the basic flowchart of datagram sockets communication. Taking the stream-based communication model as a base, as one can easily notice, the calls to `listen()` and `accept()` are dropped, and the calls to `send()` and `recv()` are replaced by calls to `sendto()` and `recvfrom()`.

Example UDP Client Program (Initiate Connections)

```
#define  WIN              // WIN for Winsock and BSD for BSD sockets

#include <stdio.h>          // Needed for printf()
#include <string.h>         // Needed for memcpy() and strcpy()
#ifdef WIN
  #include <windows.h>      // Needed for all Winsock stuff
#endif
#ifdef BSD
  #include <sys/types.h>    // Needed for system defined identifiers.
  #include <netinet/in.h>   // Needed for internet address structure.
  #include <sys/socket.h>   // Needed for socket(), bind(), etc...
  #include <arpa/inet.h>    // Needed for inet_ntoa()
  #include <fcntl.h>
  #include <netdb.h>
#endif

#define  PORT_NUM         1050  // Port number used
#define  IP_ADDR 131.247.167.101 // IP address of server1 (***
HARDWIRED ***)
void main(void)
{


#ifdef WIN
  WORD wVersionRequested = MAKEWORD(1,1);       // Stuff for WSA
functions
  WSADATA wsaData;                              // Stuff for WSA functions
#endif

  unsigned int         server_s;        // Server socket descriptor
  struct sockaddr_in   server_addr;     // Server Internet address
  int                  addr_len;        // Internet address length
  char                 out_buf[100];    // 100-byte buffer for output data
  char                 in_buf[100];     // 100-byte buffer for input data

#ifdef WIN
  // This stuff initializes winsock
  WSAStartup(wVersionRequested, &wsaData);
#endif

  // Create a socket
  //   - AF_INET is Address Family Internet and SOCK_DGRAM is
datagram
  server_s = socket(AF_INET, SOCK_DGRAM, 0);

  // Fill-in server1 socket's address information
  server_addr.sin_family      = AF_INET;            // Address family to use
  server_addr.sin_port        = htons(PORT_NUM);    // Port num to use
  server_addr.sin_addr.s_addr = inet_addr(IP_ADDR); // IP address to
use

  // Assign a message to buffer out_buf
  strcpy(out_buf, Message from client1 to server1);

  // Now send the message to server1.  The + 1 includes the end-of-
string
  // delimiter
  sendto(server_s, out_buf, (strlen(out_buf) + 1), 0,
    (struct sockaddr *)&server_addr, sizeof(server_addr));

  // Wait to receive a message
  addr_len = sizeof(server_addr);
  recvfrom(server_s, in_buf, sizeof(in_buf), 0,
    (struct sockaddr *)&server_addr, &addr_len);

  // Output the received message
  printf(Message received is: '%s' \n, in_buf);

  // Close all open sockets
#ifdef WIN
  closesocket(server_s);
#endif
#ifdef BSD
  close(server_s);
```

```
#endif

#ifdef WIN
  // Clean-up Winsock
  WSACleanup();
#endif
}
```

Example UDP Server Program (Passively Await Connection)

```

#define  WIN              // WIN for Winsock and BSD for BSD sockets

#include <stdio.h>          // Needed for printf()
#include <string.h>         // Needed for memcpy() and strcpy()
#ifdef WIN
  #include <windows.h>      // Needed for all Winsock stuff
#endif

#ifdef BSD
  #include <sys/types.h>    // Needed for system defined identifiers.
  #include <netinet/in.h>   // Needed for internet address structure.
  #include <sys/socket.h>   // Needed for socket(), bind(), etc...
  #include <arpa/inet.h>    // Needed for inet_ntoa()
  #include <fcntl.h>
  #include <netdb.h>
#endif

#define  PORT_NUM         1050     // Port number used
#define  IP_ADDR 131.247.167.101 // IP address of client1

void main(void)
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1);       // Stuff for WSA
functions
WSADATA wsaData;                              // Stuff for WSA functions
#endif

unsigned int        server_s;        // Server socket descriptor
```

```
struct sockaddr_in   server_addr;     // Server1 Internet address
struct sockaddr_in   client_addr;     // Client1 Internet address
int                  addr_len;        // Internet address length
char                 out_buf[100];    // 100-byte buffer for output data
char                 in_buf[100];     // 100-byte buffer for input data
long int             i;               // Loop counter

#ifdef WIN
// This stuff initializes winsock
WSAStartup(wVersionRequested, &wsaData);
#endif

// Create a socket
// AF_INET is Address Family Internet and SOCK_DGRAM is datagram
server_s = socket(AF_INET, SOCK_DGRAM, 0);

// Fill-in my socket's address information
server_addr.sin_family      = AF_INET;        // Address family
server_addr.sin_port        = htons(PORT_NUM);    // Port number
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // Listen on any IP
address
bind(server_s, (struct sockaddr *)&server_addr, sizeof(server_addr));

// Fill-in client1 socket's address information
client_addr.sin_family = AF_INET;            // Address family to use
client_addr.sin_port = htons(PORT_NUM);      // Port num to use
client_addr.sin_addr.s_addr = inet_addr(IP_ADDR); // IP address to use



// Wait to receive a message from client1
addr_len = sizeof(client_addr);
recvfrom(server_s, in_buf, sizeof(in_buf), 0,
(struct sockaddr *)&client_addr, &addr_len);

// Output the received message
printf(Message received is: '%s' \n, in_buf);

// Spin-loop to give client1 time to turn-around
for (i=0; i>> Step #5 <<<
// Now send the message to client1.  The + 1 includes the end-of-string
// delimiter
sendto(server_s, out_buf, (strlen(out_buf) + 1), 0,
(struct sockaddr *)&client_addr, sizeof(client_addr));
// Close all open sockets
#ifdef WIN
closesocket(server_s);
#endif
#ifdef BSD
close(server_s);
#endif
#ifdef WIN

// Clean-up Winsock
WSACleanup();
#endif
}
```

M.5 RUN-TIME PROGRAM CONTROL

Nonblocking socket calls

By default a socket is created as blocking, (i.e. it blocks until the current function call is completed). For example, if we execute an `accept()` on a socket, the process will block until there is an incoming connection from a client. In UNIX two functions are involved in turning a blocking socket into a nonblocking one: `ioctl()` and `select()`. The first facilitates input/output control on a file descriptor or socket. The `select()` function is then used to determine the socket status – ready or not ready to perform action.

```
// change the blocking state of a socket
unsigned long unblock = TRUE; // TRUE for nonblocking, FALSE for blocking
ioctl(s, FIONBIO, &unblock);
```

We then call accept() periodically:

```
while(client_s  = accept(s, NULL, NULL) > 0)
{
      if ( client_s == EWOULDBLOCK)
       // wait until a client connection arrives,
       // while executing useful tasks
   else
       // process accepted connection
}

// display error and exit
```

or use the `select`() function call to query the socket status, as in the following segment from a nonblocking socket program:

```
if (select(max_descr + 1, &sockSet, NULL, NULL, &sel_timeout)
== 0)
    // print a message for the user
else
{   …
client_s  = accept(s, NULL, NULL);
    …
}
```

In this way, when some socket descriptor is ready for I/O, the process has to be constantly polling the OS with `select()` calls, until the socket is ready. Although the process executing a `select()` call would suspend the program until the socket is ready or until the `select()` function times out (as opposed to suspending it until the socket is ready, if the socket were blocking), this solution is still inefficient. Just like calling a nonblocking `accept()` within a loop, calling `select()` within a loop results in wasting CPU cycles.

Asynchronous I/O (Signal Driven I/O)

A better solution is to use asynchronous I/O (i.e., when I/O activity is detected on the socket, the OS informs the process immediately and thus relieves it from the burden of polling all the time). In the original BSD UNIX this involves the use of calls to `sigaction()` and `fcntl()`. An alternative to poll for the status of a socket through the `select()` call is to let the kernel inform the application about events via a `SIGIO` signal. In order to do that, a valid signal handler for `SIGIO` must be installed with `sigaction()`. The following program does not involve sockets, it merely provides a simple example on how to install a signal handler. It catches an interrupt char (Cntrl-C) input by setting the signal handling for `SIGINT` (interrupt signal) via `sigaction()`:

```
#include <stdio.h>       // for printf()
#include <sys/signal.h> // for sigaction()
#include <unistd.h>      // for pause()

void catch_error(char *errorMessage);   // for error handling
void InterruptSignalHandler(int signalType); // handle interr. signal

int main(int argc, char *argv[])
{
 struct sigaction handler;  // Signal handler specification

 // Set InterruptSignalHandler() as a handler function
 handler.sa_handler =  InterruptSignalHandler;

 // Create mask for all signals
 if (sigfillset(&handler.sa_mask) < 0)
      catch_error(sigfillset() failed);
 // No flags
 handler.sa_flags = 0;

 // Set signal handling for interrupt signals
 if (sigaction(SIGINT, &handler, 0) < 0)
     catch_error(sigaction() failed);

 for(;;)
     pause();  // suspend program until signal received

 exit(0);
}

void InterruptSignalHandler(int signalType)
{
 printf(Interrupt Received. Program terminated.\n);
 exit(1);
}
```

A FASYNC flag must be set on a socket file descriptor via fcntl(). In more detail, first we notify the OS about our desire to install a new disposition for SIGIO, using sigaction(); then we force the OS to submit signals to the current process by using fcntl(). This call is needed to ensure that among all processes that access the socket, the signal is delivered to the current process (or process group)); next, we use fcntl()again to set the status flag on the same socket descriptor for asynchronous FASYNC. The following segment of a datagram sockets program follows this scheme. Note that all the unnecessary details are omitted for clarity:

```
int main()
{
  …

 // Create socket for sending/receiving datagrams

 // Set up the server address structure

 // Bind to the local address
 // Set signal handler for SIGIO

 // Create mask that mask all signals
 if (sigfillset(&handler.sa_mask) < 0)
     // print error and exit

 // No flags
 handler.sa_flags = 0;

 if (sigaction(SIGIO, &handler, 0) < 0)
 // print error and exit

 // We must own the socket to receive the SIGIO message
 if (fcntl(s_socket, F_SETOWN, getpid()) < 0)
 //print error and exit



 // Arrange for asynchronous I/O and SIGIO delivery

 if (fcntl(s_socket, F_SETFL, FASYNC | O_NONBLOCK) < 0)
 // print error and exit

 for (;;)
    pause();

 …
}
```

Under Windows the `select()` function is not implemented. The `WSAAsyncSelect()` is used to request notification of network events (i.e., request that Ws2_32.dll sends a message to the window `hWnd`):

`WSAAsyncSelect(`SOCKET *`socket`*, HWND *`hWnd`*, unsigned int *`wMsg`*, long *`lEvent`*`)`

The SOCKET type is defined in `winsock.h`. *`socket`* is a socket descriptor, *`hWnd`* is the window handle, *`wMsg`* is the message, *`lEvent`* is usually a logical

OR of all events we are expecting to be notified of, when completed. Some of the event values are FD_CONNECT (connection completed), FD_ACCEPT (ready to accept), FD_READ (ready to read), FD_WRITE (ready to write), FD_CLOSE (connection closed). You can easily incorporate the following stream sockets program segment into the previously presented programs or your own application. (Again. details are omitted):

```
 // the message for the asynchronous notification
  #define wMsg (WM_USER + 4)

 ...

// socket_s has already been created and bound to a name

// listen for connections
if (listen(socket_s, 3) == SOCKET_ERROR)
    // print error message
     // exit after clean-up

// get notification on connection accept
// to this window
if (WSAAsyncSelect(s, hWnd, wMsg, FD_ACCEPT)== SOCKET_ERROR)
    // print cannot process asynchronously
     // exit after clean-up
 else
     // accept the incoming connection
```

Further references on Asynchronous I/O are "The Pocket Guide to TCP/IP Sockets – C version" by Donahoo and Calvert (for UNIX), and "Windows Sockets Network Programming" (for Windows), by Bob Quinn.

M.6 REMOTE EXECUTION OF A WINDOWS CONSOLE APPLICATION

Simple sockets operations can be used to accomplish tasks that otherwise hard to achieve. For example, by using sockets we can remotely execute an application. The sample code[1] is presented. Two sockets programs, a local and remote, are used to transfer a Windows console application (an `.exe` file) from the local host to the remote host. The program is executed on the remote host, and then `stdout` is returned to the local host.

Local code

```
#include <stdio.h>              // Needed for printf()
#include <stdlib.h>             // Needed for exit()
#include <string.h>             // Needed for memcpy() and strcpy()
#include <windows.h>            // Needed for Sleep() and Winsock stuff
#include <fcntl.h>              // Needed for file i/o constants
#include <sys\stat.h>           // Needed for file i/o constants
#include <io.h>                 // Needed for open(), close(), and eof()


#define  PORT_NUM         1050    // Arbitrary port number for the
server
#define  MAX_LISTEN          1    // Maximum number of listens to
queue
#define  SIZE             256    // Size in bytes of transfer buffer


void main(int argc, char *argv[])
{
WORD wVersionRequested = MAKEWORD(1,1); // WSA functions
WSADATA wsaData;                        // Winsock API data structure

unsigned int        remote_s;        // Remote socket descriptor
struct sockaddr_in   remote_addr;     // Remote Internet address
struct sockaddr_in   server_addr;     // Server Internet address
```

[1] This and other code presented is in part written by Ken Christensen and Karl S. Lataxes at the Computer Science Department of the University of South Florida, URL http://www.csee.usf.edu/~christen/tools/.

```
unsigned char        bin_buf[SIZE];   // Buffer for file transfer
unsigned int         fh;              // File handle
unsigned int         length;          // Length of buffers transferred
struct hostent       *host;           // Structure for gethostbyname()
struct in_addr       address;         // Structure for Internet address
char                 host_name[256];  // String for host name
int                  addr_len;        // Internet address length
unsigned int         local_s;         // Local socket descriptor
struct sockaddr_in   local_addr;      // Local Internet address
struct in_addr       remote_ip_addr;  // Remote IP address

// Check if number of command line arguments is valid
if (argc !=4)
{
printf(  *** ERROR - Must be 'local (host) (exefile) (outfile)'    \n);
printf(              where host is the hostname *or* IP address    \n);
printf(              of the host running remote.c, exefile is the  \n);
printf(              name of the file to be remotely run, and      \n);
printf(              outfile is the name of the local output file. \n);
exit(1);
}



// Initialization of winsock
WSAStartup(wVersionRequested, &wsaData);
// Copy host name into host_name
strcpy(host_name, argv[1]);

// Do a gethostbyname()
host = gethostbyname(argv[1]);
if (host == NULL)
{
printf(  *** ERROR - IP address for '%s' not be found \n, host_name);
exit(1);
}

// Copy the four-byte client IP address into an IP address structure
memcpy(&address, host->h_addr, 4);

// Create a socket for remote
remote_s = socket(AF_INET, SOCK_STREAM, 0);

// Fill-in the server (remote) socket's address information and connect
// with the listening server.
server_addr.sin_family      = AF_INET;            // Address family to use
server_addr.sin_port        = htons(PORT_NUM);    // Port num to use
server_addr.sin_addr.s_addr = inet_addr(inet_ntoa(address)); // IP
address
connect(remote_s, (struct sockaddr *)&server_addr,
sizeof(server_addr));

// Open and read *.exe file
if((fh = open(argv[2], O_RDONLY | O_BINARY, S_IREAD | S_IWRITE))
== -1)
{
printf(  ERROR - Unable to open file '%s'\n, argv[2]);
exit(1);
}

// Output message stating sending executable file
printf(Sending '%s' to remote server on '%s' \n, argv[2], argv[1]);

// Send *.exe file to remote
while(!eof(fh))
{
length = read(fh, bin_buf, SIZE);
send(remote_s, bin_buf, length, 0);
}

// Close the *.exe file that was sent to the server (remote)
close(fh);

// Close the socket
closesocket(remote_s);

// Cleanup Winsock
WSACleanup();

// Output message stating remote is executing
printf(  '%s' is executing on remote server \n, argv[2]);

// Delay to allow everything to clean-up
Sleep(100);

// Initialization of winsock
WSAStartup(wVersionRequested, &wsaData);
```

```
// Create a new socket to receive output file from remote server
local_s = socket(AF_INET, SOCK_STREAM, 0);

// Fill-in the socket's address information and bind the socket
local_addr.sin_family      = AF_INET;           // Address family to use
local_addr.sin_port        = htons(PORT_NUM);    // Port num to use
local_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // Listen on any IP
addr
bind(local_s, (struct sockaddr *)&local_addr, sizeof(local_addr));

// Listen for connections (queueing up to MAX_LISTEN)
listen(local_s, MAX_LISTEN);

// Accept a connection, the accept will block and then return with
// remote_addr filled in.
addr_len = sizeof(remote_addr);
remote_s = accept(local_s, (struct sockaddr*)&remote_addr,
&addr_len);

// Copy the four-byte client IP address into an IP address structure
memcpy(&remote_ip_addr, &remote_addr.sin_addr.s_addr, 4);

// Create and open the output file for writing
if ((fh=open(argv[3], O_WRONLY | O_CREAT | O_TRUNC | O_BINARY,
S_IREAD | S_IWRITE)) == -1)
{
printf(  *** ERROR - Unable to open '%s'\n, argv[3]);
exit(1);
}

// Receive output file from server
length = SIZE;
while(length > 0)
{
length = recv(remote_s, bin_buf, SIZE, 0);
write(fh, bin_buf, length);
}

// Close output file that was received from the remote
close(fh);

// Close the sockets
```

```
closesocket(local_s);
closesocket(remote_s);

// Output final status message
printf(Execution of '%s' and transfer of output to '%s' done! \n,
argv[2], argv[3]);

// Cleanup Winsock
WSACleanup();
}
```

Remote Code

```
#include <stdio.h>                // Needed for printf()
#include <stdlib.h>               // Needed for exit()



#include <string.h>               // Needed for memcpy() and strcpy()
#include <windows.h>              // Needed for Sleep() and Winsock stuff
#include <fcntl.h>                // Needed for file i/o constants
#include <sys\stat.h>             // Needed for file i/o constants
#include <io.h>                   // Needed for open(), close(), and eof()
#define  PORT_NUM           1050    // Arbitrary port number for the
server
#define  MAX_LISTEN            1    // Maximum number of listens to queue
#define  IN_FILE        run.exe   // Name given to transferred *.exe file
#define  TEXT_FILE       output   // Name of output file for stdout
#define  SIZE                256    // Size in bytes of transfer buffer

void main(void)
{
WORD wVersionRequested = MAKEWORD(1,1);       // WSA functions
WSADATA wsaData;                              // WSA functions
unsigned int         remote_s;        // Remote socket descriptor
struct sockaddr_in   remote_addr;     // Remote Internet address
struct sockaddr_in   server_addr;     // Server Internet address
unsigned int         local_s;         // Local socket descriptor
struct sockaddr_in   local_addr;      // Local Internet address

struct in_addr       local_ip_addr;   // Local IP address
int                  addr_len;        // Internet address length
unsigned char        bin_buf[SIZE];   // File transfer buffer
unsigned int         fh;              // File handle
unsigned int         length;          // Length of transf. buffers
// Do forever
while(1)
{
// Winsock initialization
WSAStartup(wVersionRequested, &wsaData);

// Create a socket
remote_s = socket(AF_INET, SOCK_STREAM, 0);

// Fill-in my socket's address information and bind the socket
remote_addr.sin_family      = AF_INET;            // Address family to use
remote_addr.sin_port        = htons(PORT_NUM);    // Port number to use
remote_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // Listen on any IP addr
bind(remote_s, (struct sockaddr *)&remote_addr, sizeof(remote_addr));

// Output waiting message
printf(Waiting for a connection... \n);

// Listen for connections (queueing up to MAX_LISTEN)
listen(remote_s, MAX_LISTEN);

// Accept a connection, accept() will block and return with local_addr
addr_len = sizeof(local_addr);
local_s = accept(remote_s, (struct sockaddr *)&local_addr, &addr_len);

// Copy the four-byte client IP address into an IP address structure
memcpy(&local_ip_addr, &local_addr.sin_addr.s_addr, 4);

// Output message acknowledging receipt, saving of *.exe
printf(  Connection established, receiving remote executable file \n);

// Open IN_FILE for remote executable file
if((fh = open(IN_FILE, O_WRONLY | O_CREAT | O_TRUNC | O_BINARY,
S_IREAD | S_IWRITE)) == -1)


{
printf(  *** ERROR - unable to open executable file \n);
exit(1);
}

// Receive executable file from local
length = 256;
while(length > 0)
{
length = recv(local_s, bin_buf, SIZE, 0);
write(fh, bin_buf, length);
}

// Close the received IN_FILE
close(fh);
// Close sockets
closesocket(remote_s);
closesocket(local_s);

// Cleanup Winsock
WSACleanup();

// Print message acknowledging execution of *.exe
printf(  Executing remote executable (stdout to output file) \n);

// Execute remote executable file (in IN_FILE)
system(IN_FILE > TEXT_FILE);

// Winsock initialization to re-open socket to send output file to local
WSAStartup(wVersionRequested, &wsaData);

// Create a socket
//   - AF_INET is Address Family Internet and SOCK_STREAM is streams
local_s = socket(AF_INET, SOCK_STREAM, 0);

// Fill in the server's socket address information and connect with
// the listening local
server_addr.sin_family      = AF_INET;
server_addr.sin_port        = htons(PORT_NUM);
server_addr.sin_addr.s_addr = inet_addr(inet_ntoa(local_ip_addr));
connect(local_s, (struct sockaddr *)&server_addr, sizeof(server_addr));

// Print message acknowledging transfer of output to client
printf(  Sending output file to local host \n);

// Open output file to send to client
if((fh = open(TEXT_FILE, O_RDONLY | O_BINARY, S_IREAD | S_IWRITE)) == -1)
{
printf(  *** ERROR - unable to open output file \n);
exit(1);
}

// Send output file to client
while(!eof(fh))
{
length = read(fh, bin_buf, SIZE);
send(local_s, bin_buf, length, 0);
}




// Close output file
close(fh);

// Close sockets
closesocket(remote_s);
closesocket(local_s);

// Cleanup Winsock
WSACleanup();

// Delay to allow everything to clean-up
Sleep(100);
}
}
```

Proto	Local Address	Foreign Address	State
TCP	Mycomp:1025	Mycomp:0	LISTENING
TCP	Mycomp:1026	Mycomp:0	LISTENING
TCP	Mycomp:6666	Mycomp:0	LISTENING
TCP	Mycomp:6667	Mycomp:0	LISTENING
TCP	Mycomp:1234	mycomp:1234	TIME_WAIT
TCP	Mycomp:1025	2hfc327.any.com:6667	ESTABLISHED
TCP	Mycomp:1026	46c311.any.com:6668	ESTABLISHED
UDP	Mycomp:6667	*.*	

Figure 1 Sample netstat Output

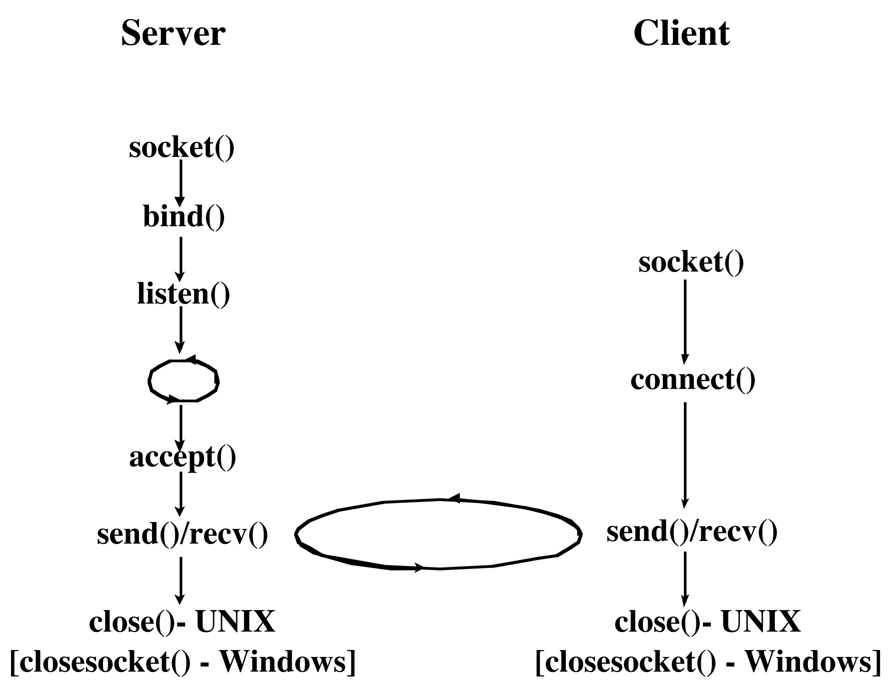

Figure 2 Flowchart of Stream Sockets Communication

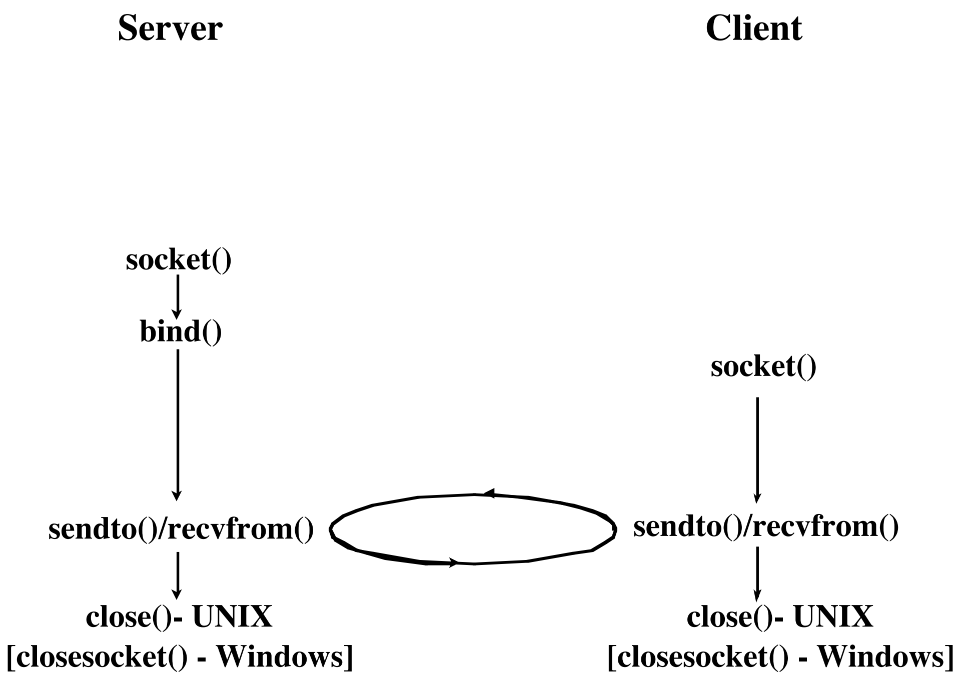

Figure 3 Flowchart of Datagram Sockets Communication