

Akhil Mathema

Presents

Network Programming

For

Bachelor in Engineering Information
Technology
&
Bachelor in Software Engineering

Final Semester

Pokhara University
NEPAL

Chapter – 1: Basic Fundamentals

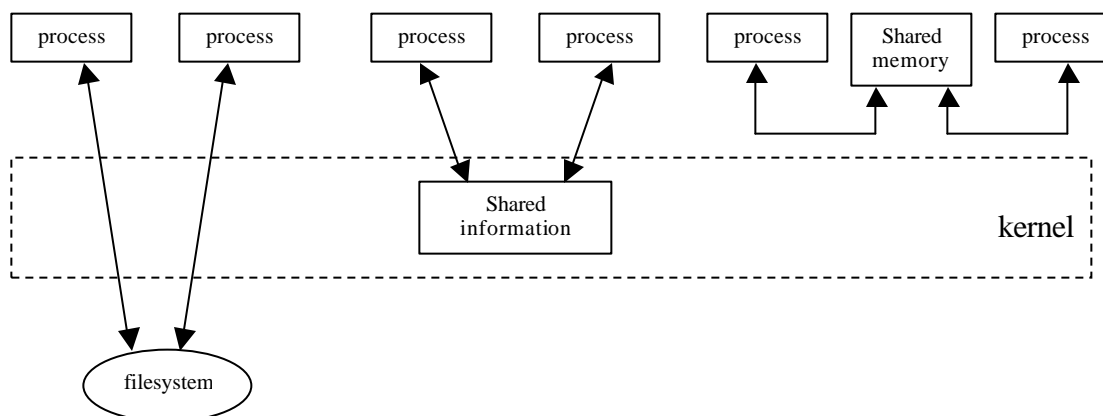
IPC: stands for “Interprocess Communications”, which describes different ways of “message passing” between different processes that are running on some operating system. In the evolution of the UNIX operating system over the past three decades, message passing has evolved through the following stages:

- ?? *Pipes* were the first widely used form of IPC, available both within programs and from the shell. Their demerit was that they are usable only between processes that have a common ancestor (parent-child relationship), but it was fixed with the introduction of “named pipes or FIFOs”.
- ?? *System V message queues* were added to System V kernels in the early 1980s, which can be used between related or unrelated processes on a given host.
- ?? *POSIX message queues* were added by the POSIX realtime standard. These can be used between related or unrelated processes on a given host.
- ?? *Remote Procedure Calls (RPC)* appeared in mid 1980s as a way of calling a function on one system (the server) from a program on another system (the client), and was developed as an alternative to explicit network programming.

Processes, Threads, and Sharing of Information:

In the traditional UNIX programming model, there are multiple processes running on a system, with each process having its own address space. Information can be shared in various ways:

- ?? The two processes share some information that resides in a file in the filesystem. To access this data, each process must go through the kernel (e.g. read, write, lseek, etc). In order to protect multiple writers from each other, and to protect one or more readers from a writer, some form of synchronization is required when a file is being updated.
- ?? The two processes share some information that resides within the kernel. Pipe, System V message queues and System V Semaphores are the examples of this type of sharing. Each operation has to access the shared information now involves a system call into the kernel.
- ?? The two processes have a region of shared memory that each process can reference. Once the shared memory is set up by each process, the processes can access the data in the shared memory without involving the kernel at all. Some form of synchronization is required by the processes that are sharing the memory.



Three ways to share information between UNIX processes

Threads: The concept of multiple “threads” within a given process is relatively new. The POSIX.1 thread standard was approved in 1995. From an IPC perspective, all the threads within a given process share the same global variables.

Semaphores:

Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s. Dijkstra's model was the operation of railroads: consider a stretch of railroad in which there is a single track over which only one train at a time is allowed. Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter. In the computer version, a semaphore appears to be a simple integer. A process (or a thread) waits for permission to proceed by waiting for the integer to become 0. The signal if it proceeds signals that this by performing incrementing the integer by 1. When it is finished, the process changes the semaphore's value by subtracting one from it.

Semaphores let processes query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments.

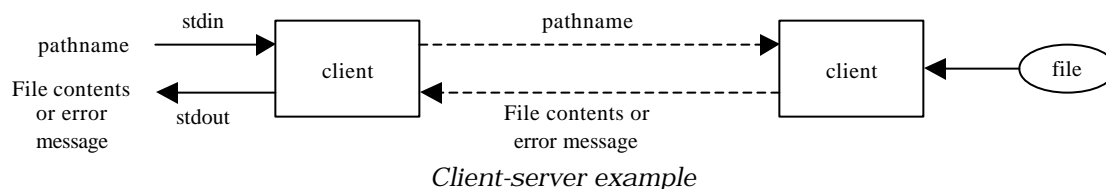
UNIX Standard: POSIX

POSIX refers to “Portable Operating System Interface”. It is not a single standard, but a family of standards being developed by the Institute of Electrical and Electronic Engineers Inc. i.e. IEEE. The POSIX standards are also being adopted as international standards by ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission). The Foreword of the 1996 POSIX.1 standard states that ISO/IEC 9945 consists of the following parts:

- ?? Part1: System application program interface (API) [C language]
- ?? Part2: Shell and utilities
- ?? Part3: System administration

The three types of IPC: POSIX message queues, POSIX semaphores and POSIX shared memory are collectively referred to as “POSIX IPC”.

A Simple Client-Server Example:

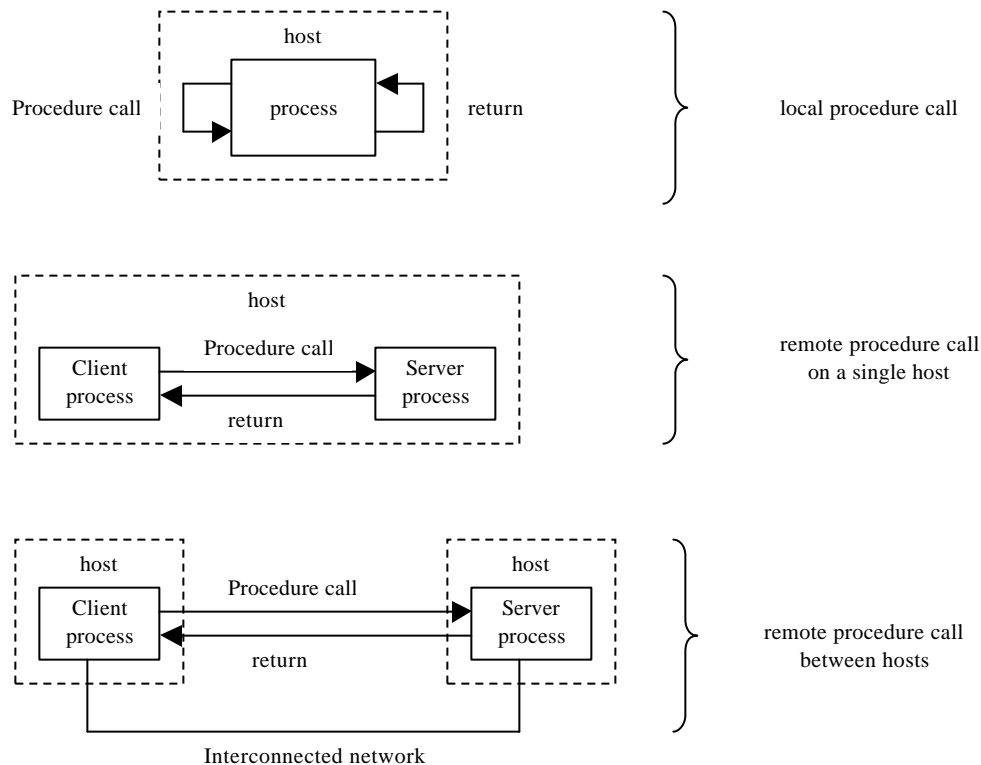


Here, the client reads a pathname from the standard input and writes it to the IPC channel. The server reads this pathname from the IPC channel and tries to open the file for reading. If the server can open the file, the server responds by reading the file and writing it to the IPC channel. Otherwise, the server responds with an error message. Then the client reads from the IPC channel, writing what it receives to the standard output. If the file cannot be read by the server, the client reads an error message from the channel. Otherwise, the client reads the contents of the file.

Remote Procedure Call, RPC:

There are three different types of procedure calls:

1. A *local procedure call* is normal C programming: the procedure (function) being called and the calling procedure are both in the same process.
2. A *remote procedure call* refers to the one when the procedure being called and the calling procedure are in different processes. Here, the caller is termed as the client and the procedure being called as the server. One process (a server) makes a procedure (function) available within that process for other processes (clients) to call by creating a door for that procedure.
3. *RPC* in general allows a client on one host to call a server procedure on another host, as long as the two hosts are connected by some form of network.

**BSD Networking History:**

The API sockets originated with the 4.2 BSD systems, released in 1983. A few changes to the sockets API also took place in 1990 with the 4.3BSD Reno release, when the OSI protocols went into the BSD kernel. In 1989, Berkeley provided the first of the BSD networking releases, which contained all of the networking code and various other pieces of the BSD system that were not constrained by the UNIX source code license. These releases were "publicly available" and eventually available by anonymous FTP to anyone on the Internet. Many UNIX systems started with some version of the BSD networking code, including the sockets API,

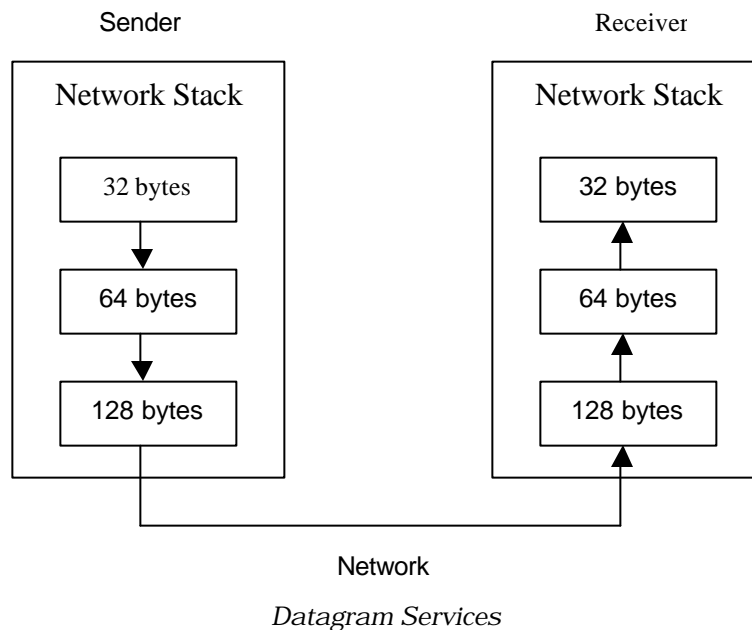
Chapter – 2: Network Principles and Protocols

Protocol Characteristics:

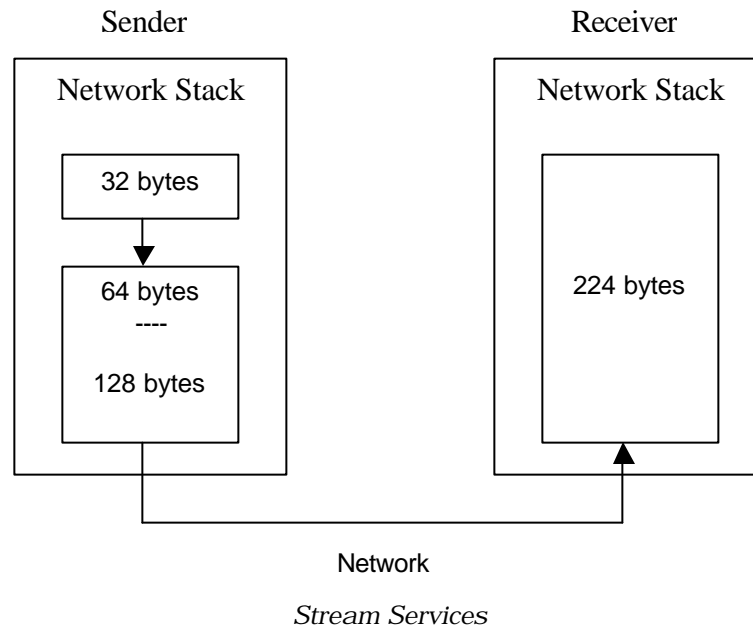
a) Message Oriented:

When a protocol transmits message for each discrete write command as a single message on the network, it is said to be message-oriented protocol. It also means when the receiver requests data, the data returned is a discrete message written by the sender. The receiver will not get more than one message. For example in the figure, the sender submits messages of 128, 64 and 32 bytes destined for the receiver. The receiver issues the three read commands with a 256-byte buffer. Each call in succession returns 128, 64 and 32 bytes. The first call to read does not return all three packets even if all the packets have been received. This logic is known as preserving message boundaries and is often desired when structured data is exchanged.

A network game is a good example of preserving message boundaries, where each player sends all other players a packet with positional information. Here, one player requests a packet of data, and that player gets exactly one packet of positional information from another player in the game.



A protocol which does not preserve the message boundaries is known as stream-based protocol. Stream service is defined as the transmitting of data in a continual process: the receiver reads as much data as is available with no respect to message boundaries. In case of the sender, the system is allowed to break up the original message into pieces or lump several messages together to form a bigger packet of data. On the receiver end, the network stack reads data as it arrives and buffers it for the process. When the process requests an amount of data, the system returns as much data as possible without overflowing the buffer supplied by the client call. In the figure below, the sender submits packets of 128, 64 and 32 bytes; however, the local system stack is free to gather the data into larger packets. Here, the second two packets are transmitted together. On the receiving end, the network stack pools together all incoming data for the given process. If the receiver performs a read with a 256 byte buffer, all 224 bytes are returned at once. And even if the receiver requests that only 20 bytes be read, the system reads only 20 bytes.



b) Connection-Oriented and Connectionless:

A protocol usually provides either connection-oriented services or connectionless services. In connection-oriented services, a path is established between the two communicating parties before any data is exchanged. This ensures that there is a route between the two parties in addition to ensuring that both parties are alive and responding. Hence, this requires a great deal of overhead. Moreover, most connection-oriented protocols guarantee delivery, further increasing overhead as additional computations are performed to verify correctness. In contrast to it, a connectionless protocol makes no guarantees that the recipient is listening. A connectionless service is similar to the postal service; the sender addresses a letter to a particular person and puts it in the mail. The sender does not know whether the recipient is expecting to receive a letter or not.

c) Reliability and Ordering:

These are the most important characteristics when designing an application to use a particular protocol. In most cases, reliability and ordering are closely tied to whether a protocol is connectionless or connection-oriented. Reliability or guaranteed delivery ensures that each byte of data from the sender will reach the intended recipient without any changes. An unreliable protocol does not ensure that each byte arrives, and it makes no guarantee as to the data's integrity.

Ordering refers to the way in which the data arrives at the recipient. A protocol that preserves ordering ensures that the recipient receives the data in the exact order it was sent.

d) Graceful Close:

It is associated only with connection-oriented protocols. In a graceful close, one side initiates the shutting down of a communication session and the other side still has the opportunity to read pending data on the transmission media or the network stack. A connection-oriented protocol that does not support graceful closes have an immediate termination of the connection and loss of any data not read by the receiving end whenever either side closes the communication channel.

In case of TCP, each side of a connection has to perform a close to fully terminate the connection. The initiating side sends a datagram with a FIN control flag to the peer. Upon receipt, the peer sends an ACK control flag back to the initiating side to acknowledge receipt of the FIN, but the peer is still able to send more data. The FIN control flag signifies that no more will be sent from the side originating the close. Once the recipient also decides no longer to

send data, it too issues a FIN, which the initiator acknowledges with an ACK control flag. At this point, the connection has been completely closed.

e) Broadcast Data:

To broadcast data is to be able to send data from one workstation so that all other workstations on the local area network can receive it. This feature is available to connectionless protocols because all machines on the LAN can receive and process a broadcast message.

When a user broadcasts a message on the LAN, the network card on each machine picks up the message and passes it up to the network stack. The stack then cycles through all network applications to see whether they should receive this message. Generally most of the machines on the network are not interested and simply discard the data. However, each machine still has to spend time for processing the packet to see whether any applications are interested in it. Consequently, high-broadcast traffic can slow down machines on a LAN as each workstation inspects the packet. In general, routers do not propagate broadcast packets.

f) Multicast Data:

It is the ability of one process to send data which will be received by one or more recipients. The method by which a process joins a multicast session differs depending on the underlying protocol. In case of an IP protocol, multicasting is a modified form of broadcasting. IP multicasting requires that all hosts interested in sending or receiving data join a special group. When a process wishes to join a multicast group, a filter is added on the network card so that data bound to only that group address will be picked up by the network device and propagated up the network stack to the appropriate process. Video conferencing applications often use multicasting.

g) Quality of Service:

QOS is an application's ability to request certain bandwidth requirements to be dedicated for exclusive use. One of its examples is real-time video streaming. In the receiving end of a real-time video streaming application, in order to display smooth and clear picture, the data to be sent must fall within certain restrictions. Previously, an application would buffer data and play back frames from the buffer to maintain a smooth video. QOS allows bandwidth to be reserved on the network, so that frames can be read off the wire within a set of guaranteed constraints. It means that the same real-time video streaming application can use QOS and eliminate the need to buffer any frames.

h) Partial Messages:

Partial messages apply only to message-oriented protocols. For instance, an application wants to receive a message but the local computer has received only part of the data. Such situation is common when the sending computer is transmitting large messages. The local machine might not have enough resources free to contain the whole message. Most message-oriented protocols impose a reasonable limit on the maximum size of a datagram. Hence such situation is not often encountered. But on the other hand, most datagram protocols support messages of a size large enough to require being broken up into a number of smaller chunks for transmission on the physical medium. If the protocol supports partial messages, the reader is notified that the data being returned is only a part of the whole message. If the protocol does not support the partial messages, the underlying network stack holds onto the pieces until the whole message arrives. If the whole message does not arrive, the incomplete datagram is simply discarded.

i) Routing Considerations:

If a protocol is routable, a successful communication path (either a virtual connection-oriented circuit or a data path for datagram communication) can be set up between two workstations, no matter what network device lies between them. For example, machine X is on a separate subnet from machine Y. A router linking the two subnets separates the two machines. A routable protocol realizes that machine Y is not on the same subnet as machine X.

Hence it directs the packet to the router, which decides how to best forward the data so that it reaches machine Y. A non-routable protocol is not able to make such provisions; the router drops any packets of non-routable protocols that it receives. The router does not forward a packet from a non-routable protocol even if the packet's intended destination is on the connected subnet. NetBEUI is the only protocol supported by Win32 platforms that is not capable of being routed.

Internet Protocol (IP)

It is commonly known as the network protocol that is used on the Internet. IP is widely available on most computer operating systems and can be used on most LANs and WANs. By design, IP is a connectionless protocol and does not guarantee delivery of data. Two higher-level protocols TCP and UDP are used for data communication over IP.

TCP:

Connection-oriented communication is possible through the Transmission Control Protocol. TCP provides reliable error-free data transmission between two computers. When applications communicate using TCP, a virtual connection is established between the source computer and the destination computer. Once a connection is established, data can be exchanged between the computers as two-way stream of bytes.

UDP:

Connectionless communication is possible through the User Datagram Protocol. UDP doesn't guarantee reliable data transmission and is capable of sending data to multiple destinations and receiving data from multiple sources. For instance, if a client sends data to a server, the data is transmitted immediately, whether or not the server is ready to receive the data. If the server receives data from the client, it doesn't acknowledge the recipient. Data is transmitted using datagram.

Both TCP and UDP use IP for data transmission and are normally referred to as TCP/IP and UDP/IP. Winsock addresses IP communication through the "AF_INET" address family, which is defined in Winsock.h and Winsock2.h.

Chapter – 3: The Transport Layer

Most client-server applications use either TCP or UDP. These two protocols in turn use the network-layer protocol IP, either Ipv4 or Ipv6. Although it is possible to use IPv4 or IPv6 directly, bypassing the transport layer, this technique (called “raw sockets”) is used less frequently. UDP is a simple, unreliable datagram protocol, while TCP is a sophisticated, reliable byte-stream protocol.

IPv4	Internet Protocol version 4, commonly known as IP has been the most important protocol for the Internet since the early 1980s. It uses 32-bit address. IPv4 provides the packet delivery service for TCP, UDP, ICMP, and IGMP.
IPv6	Internet Protocol, version 6 was designed in the mid-1990s as a replacement for IPv4. It has a larger address comprising 128 bits, to deal with the explosive growth of the Internet. IPv6 provides packet delivery service for TCP, UDP, and ICMPv6.
TCP	Transmission Control Protocol is a connection-orient protocol that provides reliable, full-duplex, byte stream for a user process. TCP socket are an example of “stream sockets”. TCP take care of details such as acknowledgments, timeouts, retransmissions, etc. Most Internet applications use TCP.
UDP	User Datagram Protocol is a connectionless protocol and UDP sockets are an example of “datagram sockets”. There is no guarantee that UDP datagram ever reach their intended destination.
ICMP	Internet Control Message Protocol handles error and control information between routers and hosts.
IGMP	Internet Group Management Protocol is used with multicasting.
ARP	Address Resolution Protocol maps an IPv4 address into a hardware address. It is normally used on broadcast networks such as Ethernet, token ring and FDDI. But it is not needed on point-to-point networks.
RARP	Reverse Address Resolution Protocol maps hardware address into an IPv4 address. It is required when a remote booting is to be done.
BPF	BSD Packet Filter provides access to the data link for a process. It is normally found on Berkeley-derived kernels.

UDP: User Datagram Protocol

UDP is a simple transport-layer protocol. The application writes a “datagram” to a UDP socket, which is encapsulated as either an IPv4 datagram or an IPv6 datagram. It is then sent to its destination. But there is no guarantee that a UDP datagram ever reaches its final destination. Problem that is encountered with network programming using UDP is its lack of reliability. If it is required to be certain that a datagram reaches its destination, numerous features will have to be built into our application such as acknowledgements from the other end, timeouts, retransmissions, etc.

Each UDP datagram has a length. If the datagram reaches its final destination correctly, then the length of the datagram is passed to the receiving application. As UDP provides a connectionless service, there need not be any long-term relationship between a UDP client and server. For instance a UDP client can create a socket and send a datagram to a given server and then immediately send another datagram on the same socket to a different server. Similarly a UDP server can receive five datagrams in a row on a single UDP socket, each from five different clients.

TCP: Transmission Control Protocol

The service provided by TCP to an application varies from that of UDP. First, TCP provides “connection” between clients and servers. A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.

TCP also provides “reliability”. When TCP sends data to the other end, it requires an acknowledgement in return. If an acknowledgement is not received, TCP automatically retransmits the data and waits for a specific time period. After some retransmissions, TCP gives up. TCP contains algorithms to estimate the “round-trip time” (RTT) between a client and server dynamically so that it knows how long to wait for an acknowledgement. For instance, the RTT on a LAN can be milliseconds while across a WAN, it can be in seconds.

TCP also “sequences” the data by associating a sequence number with every byte that it sends. For instance, assume an application writes 2048 bytes to a TCP socket, causing TCP to send two segments; the first containing the data with sequence numbers 1-1024 and the second containing the data with sequence numbers 1025-2048. Here, a “segment” refers to the unit of data that TCP passes to IP. If the segments arrive out of order, the receiving TCP will reorder the two segments based on their sequence numbers before passing the data to the receiving application (or the higher layers). If TCP receives duplicate data from its peer (say the sender thought a segment was lost and retransmitted it, when it wasn’t really lost, the network was just overloaded), it can detect that the data has been duplicated from the sequence number and the duplicate data is discarded.

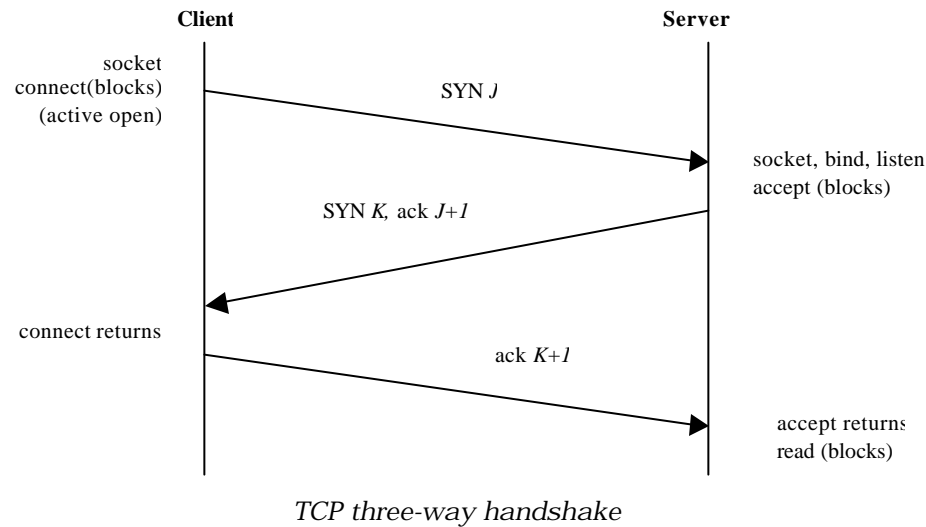
TCP further provides “flow control”. TCP always tells its peer exactly how many bytes of data it is willing to accept from the peer. This is called the “advertised window”. At a instance, the window is the amount of room currently available in the receive buffer, guaranteeing that the sender cannot overflow the receiver’s buffer. The window changes dynamically: as data is received from the sender, the window size decreases, but as the receiving application reads data from the buffer, the window increases.

Finally, a TCP connection is also “full-duplex” i.e. an application can send and receive data in both directions on a given connection at anytime. For it, TCP must keep track of state information such as sequence numbers and window sizes for each direction of data flow: sending and receiving.

TCP Connection Establishment and Termination**Three-Way Handshake:**

When a TCP connection is established, the following scenario occurs:

1. The server must be prepared to accept an incoming connection. This is normally done by called *socket*, *bind* & *listen*. It is known as a *passive open*.
2. The client issues an *active open* by calling *connect*. This causes the client TCP to send a SYN segment (synchronize) to tell the server about the client’s initial sequence number for the data. Normally, there is no data sent with the SYN; it just contains an IP header, a TCP header, and possible TCP options.
3. The server must acknowledge the client’s SYN and the server must also send its own SYN containing the initial sequence number. The sever sends its SYN and the ACK of the client’s SYN in a sing segment.
4. The client must acknowledge the server’s SYN.



The minimum number of packets required for this exchange is three; hence this is called TCP's "three-way handshake".

TCP options:

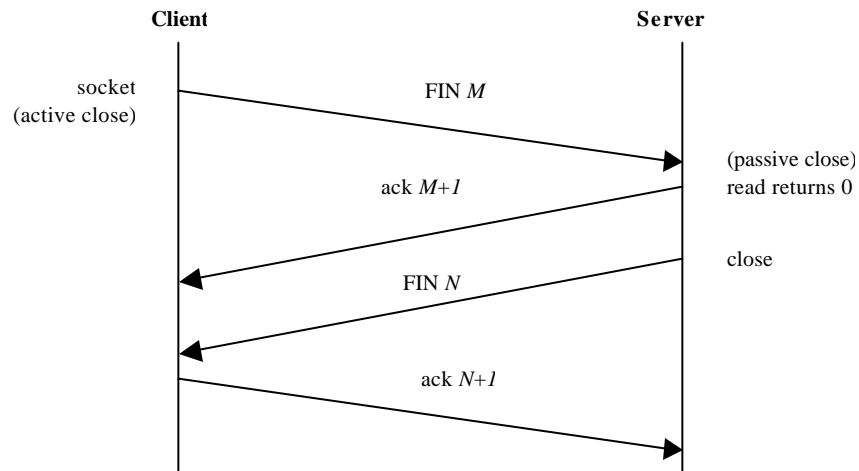
Each SYN can contain TCP options. Commonly used options are as follows:

- ?? *MSS option*: SYN with this option announces its "maximum segment size", which is the maximum amount of data that is willing to accept in each TCP segment on this connection.
- ?? *Window scale option*: refers to the maximum window that TCP can advertise to the other TCP is 65535.
- ?? *Timestamp option*: is needed for high-speed connections to prevent possible data corruption due to lost packets which reappear.

TCP Connection Termination:

While it takes three segments to establish a connection, it takes four segments to terminate connection.

1. A peer calls "close", which refers that this end performs the "active close". This end's TCP sends a FIN segment, which means it is finished sending data.
2. The other peer that receives Fin performs the "passive close". The receipt of the FIN is also passed to the application as an end-of-file, which means the application will never receive any additional data on the connection, except those that may already be queued for the application to receive.
3. The application that received the end-of-file will "close" its socket. This causes its TCP to send a FIN.
4. The TCP on the system that receives this final FIN acknowledges the FIN.



Packets exchanged when a TCP connection is closed

Maximum Segment Lifetime:

MSL is the maximum amount of time that any given IP datagram can live in the Internet. During every implementation of TCP, a value for the MSL must be chosen. The recommended value in RFC 1122 is 2 minutes, although Berkeley-derived implementations have traditionally used a value of 30 seconds instead. TIME_WAIT state is the one when the active close connection occurs. The duration of TIME_WAIT state is double of the MSL i.e. between 1 and 4 minutes. This time is bounded because every datagram contains an 8-bit hop limit with a maximum value of 255. The assumption is made that a packet with a maximum hop limit of 255 cannot exist in the Internet for more than MSL seconds.

The way in which a packet gets "lost" in the Internet is usually the result of routing irregularities. A router crashes or a link between two routers goes down and it takes the routing protocols some seconds or minutes to stabilize and find an alternate path. During that time period, routing loops can occur (router A sends packets to router B, and B sends them back to A) and packets can get caught in these loops. Meanwhile, assuming the lost packet is a TCP segment, the sending TCP times out and retransmits the packet, and the retransmitted packet gets to the final destination by some alternative path. But sometime later, if the routing loop is corrected and the packet that was lost in the loop is sent to the final destination. This original packet is called a "lost duplicate" or a "wandering duplicate". TCP must handle such conditions. There are two reasons for the TIME_WAIT state:

1. to implement TCP's full duplex connection termination reliably
2. to allow old duplicate segments to expire in the network

Chapter – 4: Socket Introduction

Port Numbers:

At any given time, multiple processes can use either UDP or TCP. Both TCP and UDP use 16-bit integer “port number” to differentiate between these processes. The port number is used between the two host computers to identify which application program is to receive the incoming traffic. The port numbers are divided into three ranges:

1. *Well-known ports:* 0 through 1023. These port numbers are controlled and assigned by IANA (Internet Assigned Numbers Authority). When possible, the same port is assigned to a given service for both TCP and UDP.
2. *Registered ports:* 1024 through 49151. These ports are not controlled by the IANA, but the IANA registers and lists the uses of these ports as a convenience to the community.
3. *Dynamic or private ports:* 49152 through 65535. These ports are also known as “ephemeral ports” i.e. short-lived ports. These ports are normally assigned automatically by TCP or UDP to the client. Clients normally don’t care about the values of these ports; they just need to be certain that the ephemeral port is unique on the client host.

Socket Pair:

The “socket pair” for a TCP connection is the 4-tuple that defines the two endpoints of the connection: the local IP address, local TCP port, foreign IP address, and foreign TCP port. A socket pair uniquely identifies every TCP connection on the Internet.

Ports and Sockets:

A “socket” is a loose term used to describe “an end point for communication.” The traditional Berkley Socket API is a set of C function calls used to support network communication. During communication, each application layer process that uses TCP/IP protocol must identify itself with a **port** number. In addition to it, TCP/IP based protocols also use an abstract identifier known as **socket**. Sockets can be classified into two topics: from the process to the kernel, and from the kernel to the process. The socket is derived from the network input/output operations of BSD 4.3 UNIX. The socket is termed as the concatenation of a port number and the network address (IP address) of the host that supports the port service.

Include Files:

When writing C or C++ programs that use the socket library it is necessary to include all these header files:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/time.h>
#include <stdlib.h>
#include <memory.h>
```

The general order of library calls for a UDP communication session is as follows:

```
socket()
bind()
sendto() and/or recvfrom()
close()
```

For TCP clients, the order of library calls is as follows:

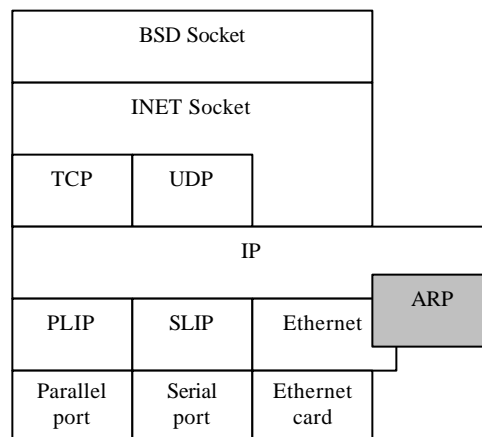
```
socket()
bind()
connect()
send() and/or recv()
close()
```

For TCP server programs, the order of library calls is as follows:

```
socket()
bind()
listen()
accept()
send() and/or recv()
close()
```

The layer model of the network implementation:

When a process communicates via the network, it uses the functions provided by the BSD socket layer, which administers a general data structure for sockets, known as BSD sockets. The BSD socket interface simplifies the porting of network applications which are pretty complex. INET socket layer is below the BSD socket layer that manages the communication end points for the IP-based protocols TCP and UDP. These are represented by the data structure *sock*, which is known as INET sockets. The layer below INET socket layer is dependent of the type of the socket which can be either TCP or UDP layer or the IP layer directly. The UDP layer implements the *User Datagram Protocol* on the basis of IP, and the TCP layer implements *Transmission Control Protocol* for reliable communication links. The IP layer contains the code for the *Internet Protocol*. Below the IP layer are the network devices, to which the IP passes the final packets. These are responsible for the physical transport of the information. True communication occurs between two sides, producing a two-way flow of information. Hence the various layers are also connected together in the opposite direction, i.e. when IP packets are received; they are passed to the IP layer by the network devices and processed.



The Layer structure of a network

Socket Address Structure

Most of the socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure. The names of these structures begin with “sockaddr_” with a distinct suffix for each protocol suite. A “sockaddr_in” structure contains an “in_addr” structure as a member field.

IPv4 Socket Address Structure:

An IPv4 socket address structure is also known as “Internet socket address structure”, which is named as “sockaddr_in” and is defined by including the “<netinet/in.h>” header. The structure is as follows:

```
struct sockaddr_in {
    uint8_t      sin_len;           /* length of structure (16) */
    sa_family_t  sin_family;       /* AF_INET */
    in_port_t    sin_port;         /* 16-bit TCP or UDP port number */
                                   /* network byte ordered */
    struct in_addr sin_addr;       /* 32-bit IPv4 address */
                                   /* network byte ordered */
    char         sin_zero;         /* unused */
};

struct in_addr {
    in_addr_t    s_addr;           /* 32-bit IPv4 address */
                                   /* network byte ordered */
};
```

Here, the length member “sin_len” is added with 4.3BSD-Reno for supporting OSI protocols, “sin_family” is an unsigned short. Despite most of the vendors do not supports a length field for socket addresses structures; it simplifies the handling of variable-length socket address structures.

In Berkeley-derived implementations, socket functions “bind, connect, sendto and sendmsg” pass a socket address structure from the process to the kernel through the “sockargs” function. This function copies the socket address structure from the process and sets its “sin_len” member to the size of the structure that was passed as an argument to these four functions. Whereas the five socket functions “accept, recvfrom, recvmsg, getpeername and getsockname” pass a socket address structure from the kernel to the process and set the “sin_len” before returning to the process.

The POSIX specification requires only three members in the structure: “sin_family, sin_addr, and sin_port”. Almost all implementations add the “sin_zero” member so that all socket address structures are at least 16 bytes in size. “sin_zero” is unused and it is always set to “0”.

Datatypes required by the POSIX specification:

Datatype	Description	Header
int8_t	Signed 8-bit integer	<sys/types.h>
uint8_t	Unsigned 8-bit integer	<sys/types.h>
int16_t	Signed 16-bit integer	<sys/types.h>
uint16_t	Unsigned 16-bit integer	<sys/types.h>
int32_t	Signed 32-bit integer	<sys/types.h>
uint32_t	Unsigned 32-bit integer	<sys/types.h>
sa_family_t	Address family of socket address structure	<sys/socket.h>
socklen_t	Length of socket address structure	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32-bit	<netinet/in.h>
in_port_t	TCP or UDP port, normally uint16-bit	<netinet/in.h>

Generic Socket Address Structure:

Socket address structures are always passed by reference when they are passed as an argument to any of the socket functions. But the socket functions that take one of the pointers as an argument must deal with socket address structures from any of the supported protocol. Here the problem is how to declare the type of pointer that is passed. As a solution, a *generic* socket address structure was defined in the “<sys/socket.h>” header. The socket functions are then defined as taking a pointer to the generic socket address structure. From an application programmer's point of view, generic socket address structure is only used to cast pointers to protocol specific structures.

```
struct sockaddr {
    uint8_t      sa_len;
    sa_family_t  sa_family;
    char         sa_data[14];
};
```

The “sa_family” field specifies the type of protocol. For TCP/IP, this field is always set to AF_INET. The remaining 14 bytes (sa_data) of this structure are always protocol dependent. For TCP/IP, IP addresses and port numbers are placed in this field. To facilitate operating with these fields, a specific type of socket address structure is used instead of the one above.

Browsing the header file reveals that this really isn't the form of the structure. It's really a very complicated union designed to hold an IP address in a variety of ways. Regardless, the “in_addr” struct is exactly 4 bytes long, which is the same size as an IP address. In the “sockaddr_in” structure, the “sin_port” field is a 16-bit unsigned value used to represent a port number. It's important to remember that these fields always need to be set and interpreted in network byte order. For example:

```
struct sockaddr_in sin;
sin.sin_family = AF_INET;
sin.sin_port = htons(9999)
sin.sin_addr.s_addr = inet_addr("128.227.224.3");
```

In the above code example, the structure sin, holds the IP address, 128.227.224.3, and references the port number 9999. Two utility functions are used to set these values. The function “htons” returns the integer argument passed into it in network byte order. The function “inet_addr” converts the string argument from a dotted-quad into a 32-bit integer. Its return value is also in network byte order.

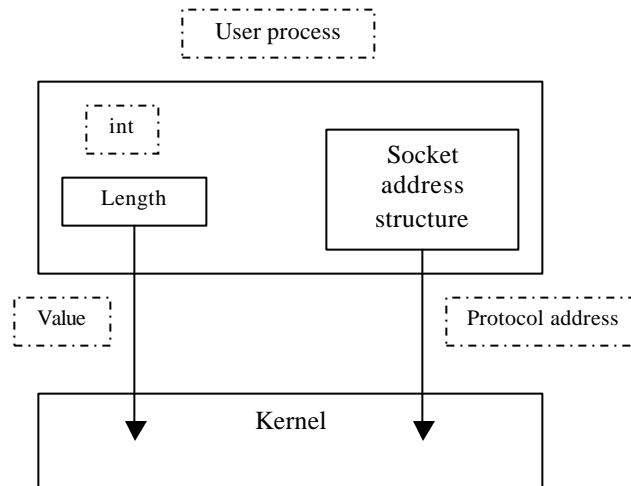
Value-Result Arguments:

When a socket address structure is passed to any of the socket functions, it is always passed by reference, i.e. **a pointer to the structure is passed**. In addition to it, the length of the structure is also passed as an argument. But the way of passing the length depends on the direction in which the structure is passed; either from the process to the kernel or vice-versa.

1. The three functions “bind, connect, & sendto” pass a socket address structure from the process to the kernel. One of the arguments to these functions is **“the pointer to the socket address structure”** and another is **“the integer size of the structure”**:

```
struct sockaddr_in serv;
/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```

Since both the pointer and the size of the structure are passed to the pointer, the kernel knows exactly how much of data is to be copied from the process into the kernel.



Socket address structure passed from process to kernel

2. The four functions “accept, recvfrom, getsockname, & getpeername” pass a socket address structure from the kernel to the process. The arguments to these functions are “the pointer to the socket address structure” along with “a pointer to an integer containing the size of the structure”:

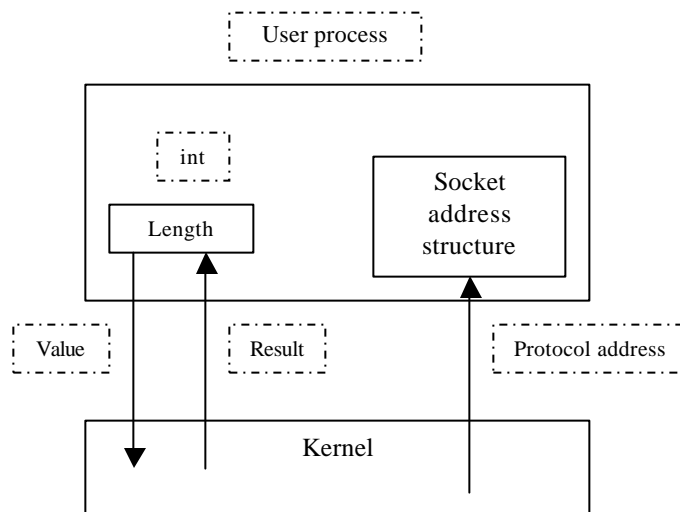
```

struct sockaddr_un cli;          /* Unix domain */
socklen_t len;

len = sizeof(cli);               /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
  
```

The size changes from an integer to be a pointer to an integer is because the size is both a *value* when the function is called, whereas it is a *result* when the function returns. Such type of argument is called a *value-result* argument. In network programming, the most common example of value-result argument is the length of a returned socket address structure.

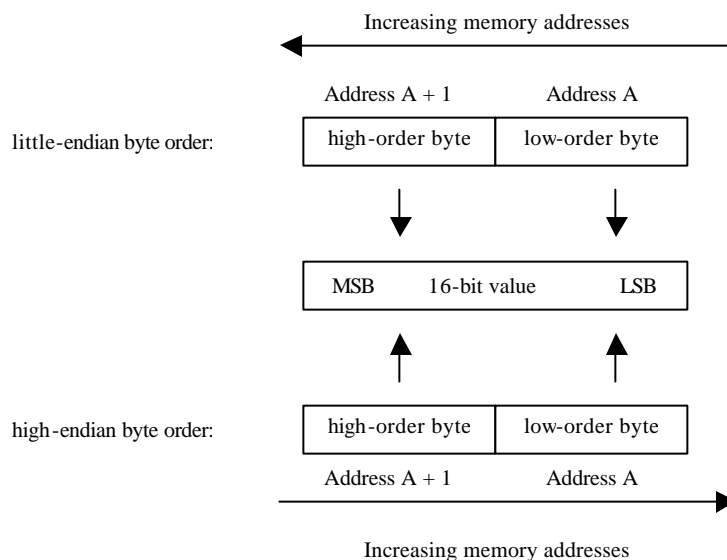
When using value-result arguments for the length of the socket address structure, if the socket address structure is fixed-length, the value returned by the kernel will be of fixed size: 16 for an IPv4 “sockaddr_in” and 28 for IPv6 “sockaddr_in6”. But with a variable-length socket address structure such as “sockaddr_un”, the value returned can be less than the maximum size of the structure.



Socket address structure passed from kernel to process

Byte Ordering Functions:

There are two ways to store the bytes in memory: with the lower-order byte at the starting address, known as “little-endian byte order”, or with the high-order byte at the starting address, known as “big-endian byte order”.



Little-endian byte order for a 16-bit integer

Here, increasing memory address is shown going from right to left in the top and from left to right in the bottom. The MSB (most significant bit) is shown as the leftmost bit of the 16-bit value and the LSB (least significant bit) as the rightmost value. The terms “little-endian” and “big-endian” indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value. The byte ordering used by a given system is known as the “host byte order”.

It is necessary to deal with the byte ordering differences as network programming because networking protocols must specify a “network byte order”. For example, in a TCP segment, there is a 16-bit port number and a 32-bit IPv4 address. The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields are transmitted.

The Internet protocols use big-endian byte ordering for these multibyte integers. An application should be able to store the fields in a socket address structure in host byte order and then convert to and from the network byte order when moving the fields to and from the protocol headers. The following functions are used to convert between host byte order and network byte order:

```
#include <netinet/in.h>
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
/* both return value in network byte order */

uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t bitvalue);
/* both returns value in host byte order */
```

In the name of these functions, “h” stands for “host”, “n” stands for “network”, “s” stands for “short” and “l” stands for “long”. When using these functions, it is not necessary to care about the actual values (big-endian or little-endian) for the host byte order and the network byte order. On those systems that have the same byte ordering as the Internet protocols i.e. big-endian, these four functions are usually defined as null macros.

Byte Manipulation Functions:

There are two groups of functions that operate on multibyte fields: without interpreting the data and without assuming that the data is a null-terminated C string. These functions are required when dealing with socket address structures, because manipulation of the fields such as an IP address is required that can contain bytes of 0, but these fields are not C character strings. The functions beginning with “str” (for string) is defined by including <string.h> header to deal with null-terminated C character strings.

The first groups of functions whose name begin with “b” (for byte) are available almost on any system that supports the socket functions. The second groups of functions whose name begin with “mem” (for memory) are from ANSI C standard and are provided with any system that supports ANSI C library.

The Berkeley-derived functions are as follows:

- ?? “bzero” sets the specified number of bytes to 0 in the destination. This function is often used to initialize a socket address structure to 0.
- ?? “bcopy” moves the specified number of bytes from the source to the destination.
- ?? “bcmp” compares two arbitrary bytes strings.

The ANSI C functions are as follows:

- ?? “memset”
- ?? “memcpy”
- ?? “memcmp”

inet_aton, inet_addr and inet_ntoa Functions:

These functions convert Internet address between a dotted-decimal string and its 32-bit network byte ordered binary value.

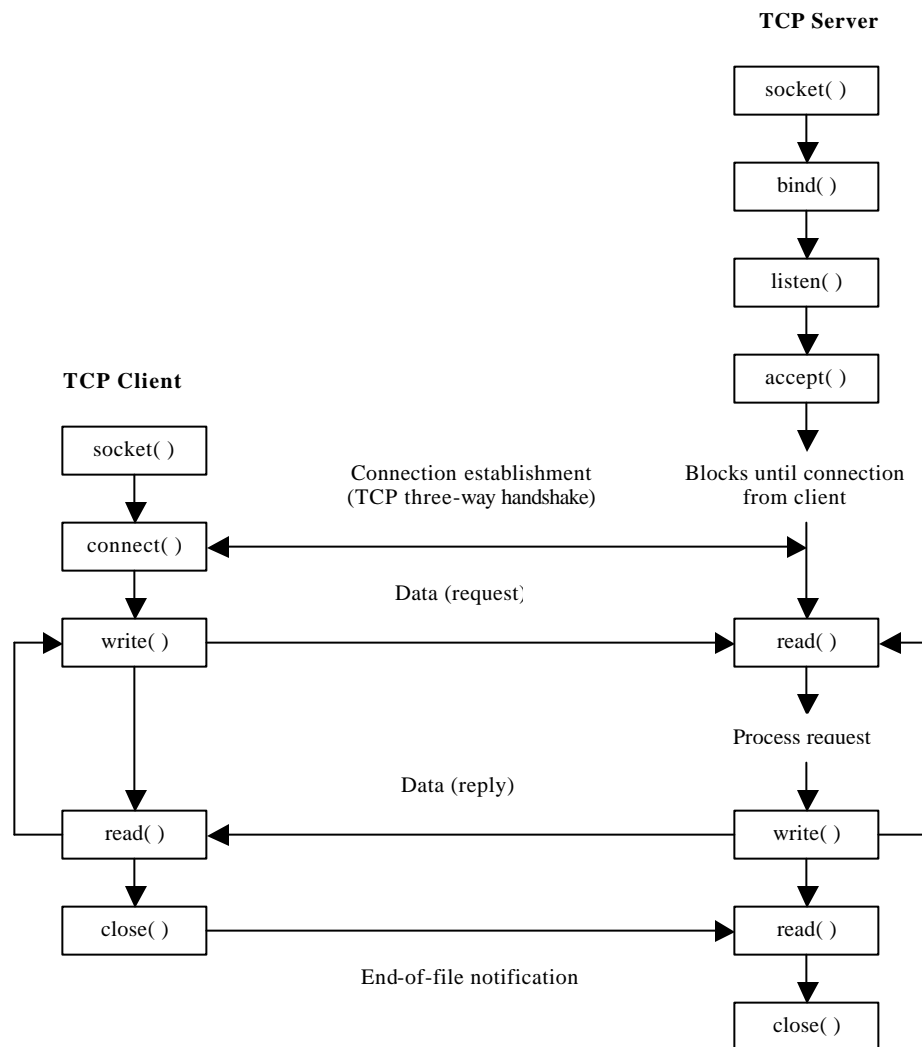
- ?? “inet_aton” converts the C character string pointed to by “strptr” into its 32-bit binary network byte ordered value, which is stored through the pointer “addrptr”.
- ?? “inet_addr” does the same conversion, returning the 32-bit binary network byte ordered value as the return value.
- ?? “inet_aton” is used instead of “inet_addr” in these days.
- ?? “inet_ntoa” converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string.

inet_pton and inet_ntop Functions:

These two functions are new with IPv6 and work with both IPv4 and IPv6 addresses. The letters “p” and “n” stand for “presentation” and “numeric” respectively. The presentation format for an address is an ASCII string and the numeric format is the binary value that goes into a socket address structure.

Chapter – 5: Elementary TCP Sockets

One of the basic component for writing a complete TCP client and server is the elementary socket function. The figure below shows the interaction between TCP client and server. At first, the server is started, and then sometime later a client is started that connects to the server. It is assumed that the client sends a request to the server, the server processes the request, and the server sends back a reply to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. Finally the server closes its end of the connection and either terminates or waits for a new client connection.



Socket functions for elementary TCP client-server

socket function:

In order to perform I/O, the first thing a process must do is call the *socket* function, specifying required type of communication protocol. The *family* specifies the protocol family, where the constants are shown in the table. Similarly, the constants for socket *type* are also below. Normally, the *protocol* argument to the socket function is set to 0 except for raw sockets.

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

<i>Family</i>	<i>Description</i>
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing sockets

Protocol family constants for socket function

<i>Type</i>	<i>Description</i>
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_RAW	Raw socket

Type of socket for socket function

<i>Protocol</i>	<i>Description</i>
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol

Protocol of sockets for AF_INET or AF_INET6

In short, this function creates "an end point for communication". The return value from this function is a handle to a socket. This number is passed as a parameter to almost all of the other library calls.

Since the focus of this document is on TCP/IP based sockets, the family parameter should be set to AF_INET. The type parameter can be either SOCK_STREAM (for TCP), or SOCK_DGRAM (for UDP). The protocol field is intended for specifying a specific protocol in case the network model supports different types of stream and datagram models. However, TCP/IP only has one protocol for each, so this field should always be set to 0.

On success, the "socket" function returns a small non-negative integer value known as "socket descriptor" or a "sockfd".

To create a UDP socket:

```
int s;
s = socket(AF_INET, SOCK_DGRAM, 0);
```

To create a TCP socket:

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

connect Function:

It is used by a TCP client to establish a connection with a TCP server.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen) ;
```

Where, “sockfd” is a socket descriptor, which is returned by the socket function. The second and third arguments are a pointer to a socket address structure, and its size. The socket address structures must contain the IP address and port number of the server.

The client does not have to call “bind” before calling “connect”. If required, the kernel will choose both temporary port and the source IP address. In TCP socket, the “connect” function initiates TCP’s three-way handshake. The function returns only when the connection is established or an error occurs. There are several types of possible errors:

1. If the client TCP receives no response to its SYN segment, “ETIMEDOUT” is returned.
2. If the server’s response to the client’s SYN is a reset (RST), it indicates that no process is waiting for connections on the server host at the specified port (or, probably the server process is not running). This is a “hard error” and the error “ECONNREFUSED” is returned to the client as soon as the RST is received.

RST is a type of TCP segment that is sent by TCP when something is wrong. There are three conditions that generate RST state:

- ?? when a SYN arrives for a port that has no listening server
- ?? when TCP wants to abort an existing connection
- ?? when TCP receives a segment for a connection that does not exist

bind Function:

It assigns a local protocol address to a socket. The protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address with a 16-bit TCP or UDP port number. Before sending and receiving data with a socket, it must first be associated with a local source port and a network interface address. **The mapping of a socket to a TCP/UDP source port and IP address is called a “binding”.**

It may be the case where the socket is being used as a server, and thus must be able to listen for client requests on a specific port. It can also be the case that a client program doesn't need a specific source port, since all it's concerned about doing is sending and receiving messages with a specific port on the remote host. Further complications arise when there are more than one network devices on the host running the program. So the question of sending through “which network” must be answered as well. **The bind function call is used to declare the mapping between the socket, the TCP/UDP source port, and the network interface device.**

The prototype for bind is as follows:

```
#include <sys/socket.h>
bind(int sockfd, const struct sockaddr *address, socklen_t addrlen);
```

The first argument is a socket descriptor. The second argument is a protocol-specific address structure and the third is the size of this address structure. With TCP, there are various conditions while calling “bind” such as specifying a port number, IP address, both or neither.

Servers bind their well-known port when they start. If a TCP client or server does not do this, the kernel chooses a temporary port for the socket when either “connect” or “listen” is called. It is normal for a TCP client to let the kernel choose an ephemeral port unless the application requires a reserved port. But since servers are known by their well-known port, it is rare for a TCP server to let the kernel choose an ephemeral port.

A process can “bind” a specific IP address to its socket. The IP address must belong to an interface on the host. For a TCP client, the source IP address is assigned, which will be used for IP datagram sent on to a socket. Whereas for a TCP server, the socket is restricted to receive incoming client connections destined on to that IP address.

Normally, a TCP client does not “bind” an IP address to its socket. The kernel chooses the source IP address when the socket is connected, based on the outgoing interface that is used. Meanwhile, if a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the client’s SYN packet as the server’s source IP address. The following table summarizes the values to set for “sin_addr” and “sin_port”, or “sin6_addr” and “sin6_port”, depending on the desired result.

Process Specifies		Result
IP address	port	
Wildcard (*)	0	Kernel chooses IP address and port
Wildcard (*)	Non zero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	Non zero	Process specifies IP address and port

If a port number is specified as “0”, the kernel chooses an ephemeral port when “bind” is called. But if a wildcard is specified as an IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).

In case of IPv4, the wildcard address is specified by the constant “INADDR_ANY”, whose value is normally “0”. This tells the kernel to choose the IP address.

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* wildcard */
```

Example:

```
struct sockaddr_in sin;
int s;
s = socket(AF_INET, SOCK_DGRAM, 0);
sin.sin_family = AF_INET;
sin.sin_port = htons(9999);
sin.sin_addr.s_addr = INADDR_ANY;

bind(s, (struct sockaddr *)&sin, sizeof(sin));
```

listen Function:

This function is normally called after both the *socket* and *bind* functions, but is necessary to be called before the *accept* function.

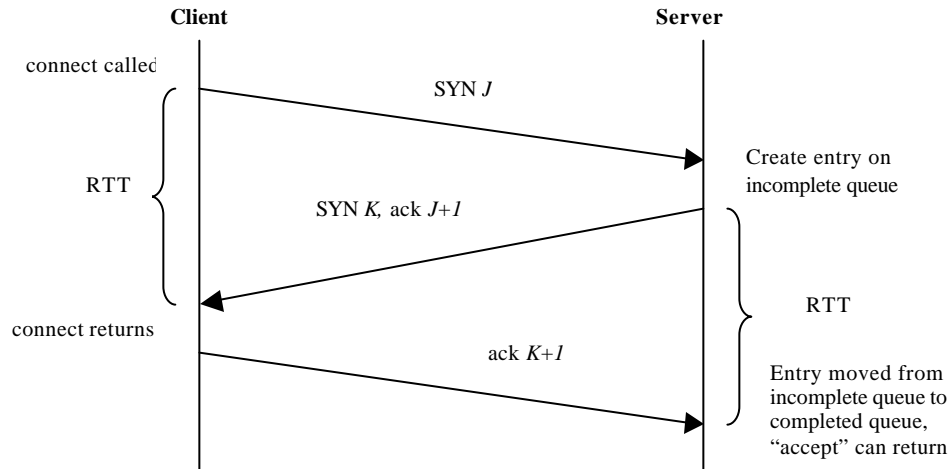
```
#include <sys/socket.h>
int listen(int sockfd, int backlog) ;
```

It is called only by a TCP server, which basically performs two actions:

1. When a socket is created by the “socket” function, it is assumed to be an active socket, i.e. a client socket that issues “connect”. The “listen” function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
2. The second argument specifies the maximum number of connections that the kernel should queue for this socket.

This function is normally called after the “socket” and “bind” functions and must be called before calling the “accept” function. In order to understand “backlog” argument, it is necessary to know that for a given listening socket, the kernel maintains two queues:

1. An *incomplete queue*, which contains an entry for each SYN that arrives from a client, for which the server is expecting the completion of TCP three-way handshake.
2. A *complete connection queue*, which contains an entry for each client with whom TCP three-way handshake is completed.



TCP three-way handshake and two queues for a listening socket

accept Function:

It is called by a TCP server to return the next completed connection from the front of the completed connection queue. If the completed connection queue is empty, the process is put to sleep.

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

The *cliaddr* & *addrlen* arguments are used to return the protocol address of the connected peer process (client). *addrlen* is a value-result argument i.e. before the call, the integer value is set, referenced by **addrlen* to the size of the socket address structure pointed to by *cliaddr*; but on return the integer contains the actual number of bytes stored by the kernel in the socket address structure.

If "accept" is successful, its return value is a new descriptor, automatically created by the kernel. The new descriptor refers to the TCP connection with the client. The first argument to "accept" is known as "listening socket" (which is the descriptor created by the socket and then used as the first argument to both "bind" and "listen") and the return value from "accept" is known as "connected socket". It is important to differentiate between these two sockets. Normally, server creates only one listening socket, which exists till the server is on. The kernel creates one connected socket for each client connection that is accepted (i.e. for which the TCP three-way handshake completes). As soon as the server finishes serving its client, the connected socket is closed.

The "accept" function returns three values:

1. Socket descriptor or an error indication
2. The protocol address of the client process (through the "cliaddr" pointer)
3. The size of this address (through the "addrlen" pointer)

If the server does not require to the protocol address of the client, both "cliaddr" and "addrlen" can be set to null pointer.

fork and exec Functions:

The *fork* function is used to create a new process in UNIX. This function is called once but it returns twice. “*fork*” returns once in the calling process (parent) with a return value that is the process ID of the newly created process (child). It also returns once in the child, with a return value of 0. Hence the return value tells the process whether it is the parent or the child.

The *fork* returns 0 in the child and it can always obtain the parent’s process ID by calling “*getppid*”, whereas, a parent can have any number of children, and it is not possible to obtain the process ID of its children. If the parent wants to keep track of the process IDs of its children, it must record the return values from “*fork*”. The parent calls “*accept*” and then calls “*fork*”. The connected socket is then shared between the parent and child. Finally, it returns “-1” on error.

```
#include <unistd.h>
pid_t fork(void)
```

There are two typical uses of “*fork*”:

1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is common for many network servers.
2. A process wants to execute another program. Since the only way to create a new process is by calling “*fork*”, the process first calls “*fork*” to make a copy of itself, and then one of the copies (typically the child) calls the “*exec*” functions to replace itself with the new program. This is common for the shell programming.

“*exec*” replaces the current process image with the new program file, which normally starts at the “*main*” function. The process ID does not change. The process that calls “*exec*” functions is referred as the “calling function” and the newly executed program as the “new program”.

In UNIX, the only way an executable program file on disk can be executed for an existing process is by calling one of the following six “*exec*” functions:

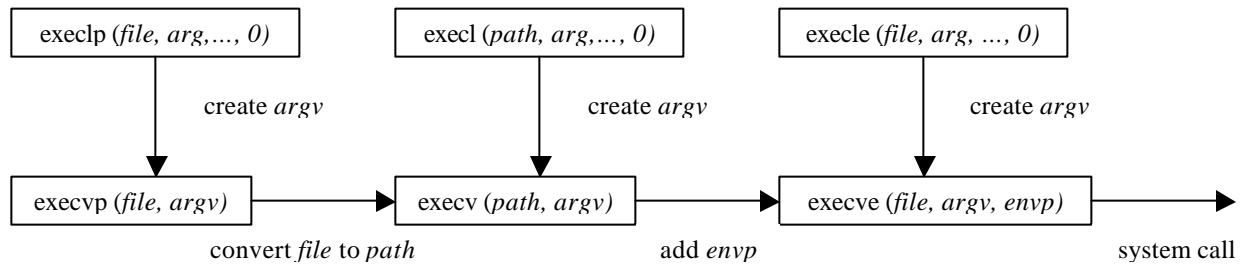
```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ...);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg0, ...);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ...);
int execvp(const char *filename, char *const argv[]);
```

The differences in six “*exec*” functions are:

- ?? whether the program file to be executed is specified by a “filename” or a “pathname”
- ?? whether the arguments to the new program are listed one by one or referenced through an array of pointers
- ?? whether the environment of the calling process is passed to the new program or whether a new environment is specified

Normally, only “execve” is a system call within the kernel and the other five are library functions that call “execve”. The relationship among six “exec” functions is shown below:



- ?? The three functions “execlp”, “execl” and “execl” specify each argument string as a separate argument to the “exec” function, with a null pointer terminating the variable number of arguments. The three functions “execvp”, “execv” and “execve” have “argv” array, containing pointers to the argument strings. The “argv” array must contain a null pointer to specify its end.
- ?? The two functions “execlp” and “execvp” specify a “filename” argument. This is converted into a “pathname” using the current PATH environment variable. The PATH variable is not used if the “filename” argument to these two functions contains a slash (/) anywhere in the string. The four functions “execl”, “execv”, “execl” and “execve” specify a fully qualified “pathname” argument.
- ?? The four functions “execlp”, “execvp”, “execl” and “execv” do not specify an explicit environment pointer. Instead, the current value of the external variable “environ” is used for building an environment list that is passed to the new program. The two functions “execl” and “execve” specify an explicit environmentlist. The “envp” array of pointers must be terminated by a null pointer.

close Function:

The “close” function is used to close a socket and terminate a TCP connection.

```
#include <unistd.h>
int close(int sockfd);
```

The default action of “close” with a TCP socket is to mark the socket as closed and return to the process immediately. The socket descriptor is no longer usable by the process: It cannot be used as an argument to “read” and “write”. But TCP will try to send any data that is already queued to be sent to other end, after this occurs; the normal TCP connection termination sequence takes place.

getsockname and getpeername Functions:

These two functions return the local protocol address associated with a socket and the foreign protocol address associated with a socket respectively.

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
```

On success, both functions return 0 or returns -1 on failure. It should be noted that the final argument for both functions is a value-result argument i.e. both functions fill in the socket address structure pointed to by “localaddr” or “peeraddr”.

These two functions are required for the following reasons:

- ?? After “connect” successfully returns in a TCP client, “getsockname” returns the local IP address and local port number assigned to the connection by the kernel.
- ?? After calling “bind” with port number “0” (telling the kernel to choose the local port number), “getsockname” returns the assigned local port number.
- ?? “getsockname” can be called to obtain the address family of a socket.
- ?? In a TCP server that “bind”s the wildcard IP address, once a connection is established with a client, the server can call “getsockname” to obtain the local IP address assigned to the connection.
- ?? When a server calls “exec”, the only way it can obtain the identity of the client is to call “getpeername”. For instance in case of a TCP server, “xinetd” calls “accept” that returns two values: the connected socket descriptor “connfd” (the return value of the function) and an Internet socket address structure (peer’s address) of the client. Then “fork” is called and a child of “xinetd” is created. Since the child starts with a copy of the parent’s memory image, the socket address structure is available to the child, as the connected socket descriptor. But when the child “exec”s the real server (say the Telnet server), the memory image of the child is replaced with the new program file for the Telnet server (i.e. the socket address structure containing the peer’s address is lost). One of the first functions called by the Telnet server is “getpeername” to obtain the IP address and port number of the client.

Chapter – 6: File Descriptors

In UNIX, all input and output is done by reading or writing files; since all peripheral devices are files in the file system. This means that a single homogenous interface handles all communication between a program and a peripheral device. In most of the cases, before a file is read or written, it is to be opened. The system checks the user's right to either read or write (such as does the file exist, or does the user have permission to access it). **If everything is okay, the system returns to the program a small non-negative integer called a "file descriptor".** Whenever input or output is to be done on the file, the file descriptor is used instead of the name to identify the file.

File Descriptors are the fundamental I/O object. These are the numbers used for I/O, **usually the result of "open" and "create" calls.**

```
int cc, fd, nbytes;
char *buf;
cc = read(fd, buf, nbytes);
cc = write(fd, buf, nbytes);
```

Here, the read attempts to read "nbytes" of data from the object referenced by the file descriptor "fd" into the buffer pointed to by "buf". The "write" does a write to the file descriptor from the buffer. Unix I/O is a byte stream. All UNIX applications run with "stdin"(<) as file descriptor 0, "stdout"(>) as file descriptor 1, and "stderr" (2>) as file descriptor 3. finally -1 indicates an error.

Any number of bytes can be read or written in one call. the most common values is "1" which refers to one character at a time and a number like 1024 or 4096 refers to a physical block size on a peripheral device. Larger sizes will be more efficient because fewer system calls will be made.

Data Transfer

The system call pairs (read, write), (recv, send), (recvfrom, sendto), (recvmsg, sendmsg), and (readv, writev) can all be used to transfer data on sockets. The most appropriate call depends on the exact functionality required. "send" and "recv" are typically used with connected stream sockets. They can also be used with datagram sockets if the sender has previously done a "connect" or the receiver does not care who the sender is. "sendto" and "recvfrom" are used with datagram sockets. "sendto" allows one to specify the destination of the datagram, while "recvfrom" returns the name of the remote socket sending the message. "read" and "write" can be used with any connected socket. These two calls may be chosen for efficiency considerations. The remaining data transfer calls can be used for more specialized purposes. "writev" and "readv" make it possible to scatter and gather data to/from separate buffers. "sendmsg" and "recvmsg" allow scatter/gather capability as well as the ability to exchange access rights. The calls "read", "write", "readv", and "writev" take either a socket descriptor or a file descriptor as their first argument; all the rest of the calls require a socket descriptor.

```
count = send(sock, buf, buflen, flags)
int count, sock, buflen, flags; char *buf;
```

```

count = recv(sock, buf, buflen, flags)
int count, sock, buflen, flags; char *buf;

count = sendto(sock, buf, buflen, flags, to, tolen)
int count, sock, buflen, flags, tolen;
char *buf;
struct sockaddr *to;

count = recvfrom(sock, buf, buflen, flags, from, fromlen)
int count, sock, buflen, flags, *fromlen;
char *buf;
struct sockaddr *from;

```

For the send calls, count returns the number of bytes accepted by the transport layer, or -1 if some error is detected locally. A positive return count is no indication of the success of the data transfer. For receive calls, count returns the number of bytes actually received, or -1 if some error is detected.

The first parameter for each call is a valid socket descriptor. The parameter “buf” is a pointer to the caller's data buffer. In the “send” calls, “buflen” is the number of bytes being sent; in the “receive” calls, it indicates the size of the data area and the maximum number of bytes the caller is willing to receive. The parameter “to” in the “sendto” call specifies the destination address and “tolen” specifies its length. The parameter from in the “recvfrom” call specifies the source address of the message. “fromlen” is a value/result parameter that initially gives the size of the structure pointed to by from and then is modified on return to indicate the actual length of the address.

The flags parameter, which is usually given zero as an argument, allows several special operations on stream sockets. It is possible to send out-of-band data or “peek” at the incoming message without actually reading it. The flags “MSG_OOB” and “MSG_PEEK” are defined in <sys/socket.h>. Out-of-band data is high priority data (such as an interrupt character) that a user might want to process quickly before all the intervening data on the stream. If out-of-band data were present, a SIGURG signal could be delivered to the user. The actual semantics of out-of band data is determined by the relevant protocol. ISO protocols treat it as “expedited data”, while Internet protocols treat it as “urgent data”.

If any of these (as well as other) system calls are interrupted by a signal, such as “SIGALRM” or “SIGIO”, the call will return -1 and the variable “errno” will be set to EINTR 2. The system call will be automatically restarted. It may be advisable to reset “errno” to zero.

Note: Error numbers are defined in the file <errno.h>

Synchronization:

By default, all the reading and writing calls are blocking: Read calls do not return until at least one byte of data is available for reading and write calls block until there is enough buffer space to accept some or all of the data being sent. Some applications need to service several network connections simultaneously, performing operations on connections as they become enabled. There are three techniques available to support such applications: non blocking sockets, asynchronous notifications, and the “select” system call. The “select” system call is by far the most commonly used; non-blocking sockets are less common, and asynchronous notifications are rarely used.

Chapter – 7: I/O Models

There are five basic types of I/O models, which are available in UNIX:

- ?? blocking I/O
- ?? nonblocking I/O
- ?? I/O multiplexing (select and poll)
- ?? Signal driven I/O (SIGIO)
- ?? Asynchronous I/O (the POSIX “aio_” functions)

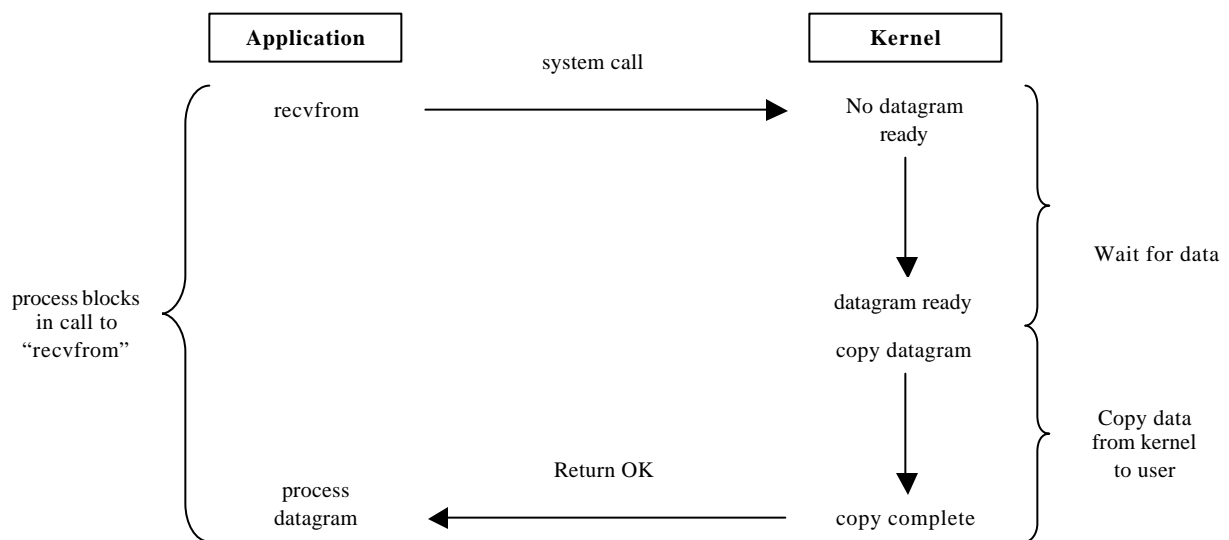
Normally, there are two distinct phases for an input operation:

1. Waiting for the data to be ready
2. Copying the data from the kernel to the process

For an input operation on a socket, the first step involves waiting for data to arrive on the network. When the packet arrives, it is copied into a buffer within the kernel. The second step is copying the data from the kernel's buffer into our application buffer.

Blocking I/O Model:

By default, all sockets are blocking. It is further explained by taking a reference of UDP in the figure below:

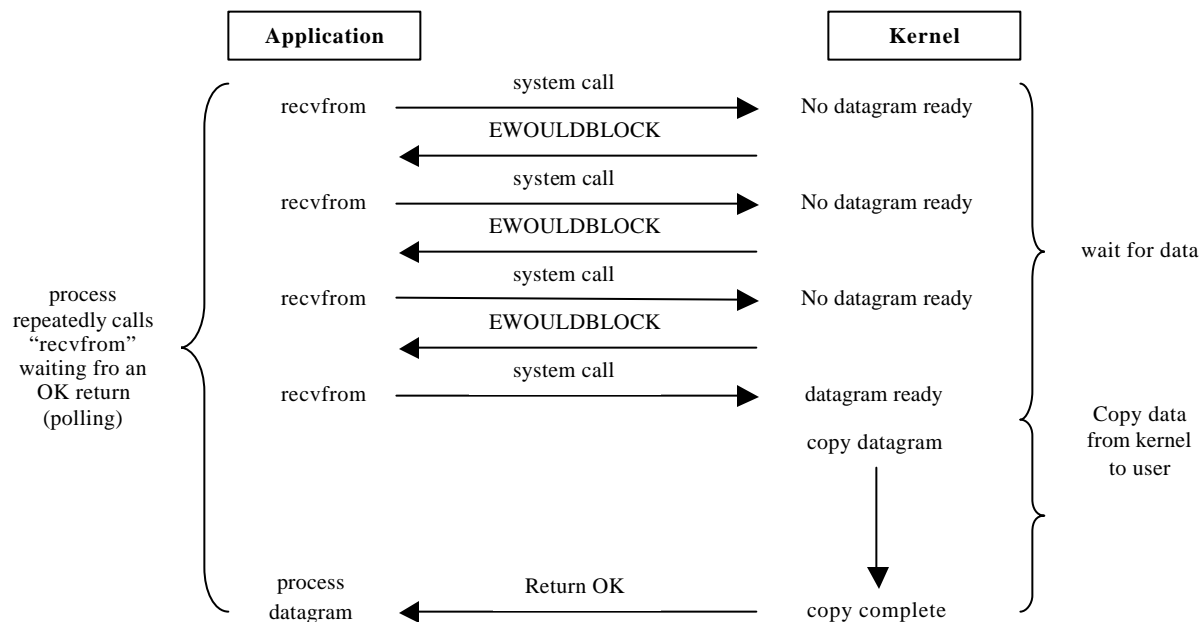


Blocking I/O Model

Here, the process calls “recvfrom” and the system call do not return until the datagram arrives and is copied into the buffer or an error occurs. The process is said to be blocked the entire time from when it calls “recvfrom” until it returns. When “recvfrom” returns successfully, an application processes the datagram.

Nonblocking I/O Model:

In nonblocking I/O model, the kernel is asked to return an error instead of putting the process to sleep in case when the requested I/O operation cannot be completed without putting the process to sleep. In the following figure, the first three “recvfrom” call has no data to return. Hence the kernel immediately returns an error of “EWOULDBLOCK” instead. In the fourth time, when “recvfrom” is called, a datagram is ready. It is copied into an application buffer and “recvfrom” returns successfully. Then the data is processed.

*Nonblocking I/O model*

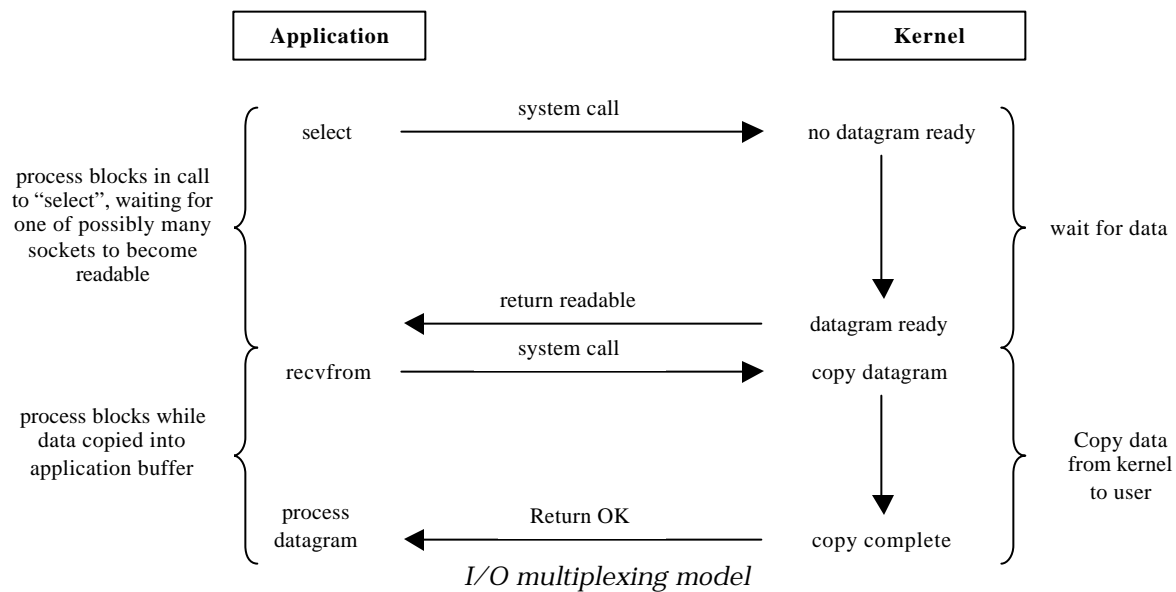
Here, when an application is in a loop calling “recvfrom” on a nonblocking descriptor, it is calling “polling”. The application is continually polling the kernel to see if some operation is ready. This is a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

I/O Multiplexing:

Let it be assumed that the TCP server correctly sends a FIN to the client, but if the client process is blocked reading from standard input, it will never see the EOF until it read from the socket. Hence it is necessary to make it capable to tell the kernel what is required to be notified if one or more I/O conditions are ready. This capability is known as “I/O multiplexing” and is provided by the “select” and “poll” functions.

I/O multiplexing is typically used in networking applications in the following cases:

- ?? When a client is handling multiple descriptor (normally interactive inputs and a network socket)
- ?? When a client is handling multiple sockets at the same time, such as in web client.
- ?? If a TCP server handles both a listening socket and its connected sockets.
- ?? If a server handles both TCP and UDP.
- ?? If a server handles multiple services and protocols.



Here, "select" is called that waits for the datagram socket to be readable. When "select" returns that the socket is readable, then "recvfrom" is called to copy from the datagram into an application buffer.

"select" Function: allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occur, or when a specified amount of time has passed.

```
#include <sys/select.h>
#include <sys/time.h>
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set
*exceptset, const struct timeval *timeout);
```

Description of "select" function is started with its final argument, which tells the kernel how long to wait for one of the specified descriptors to become ready. A "timeval" structure specifies the number of seconds and microseconds.

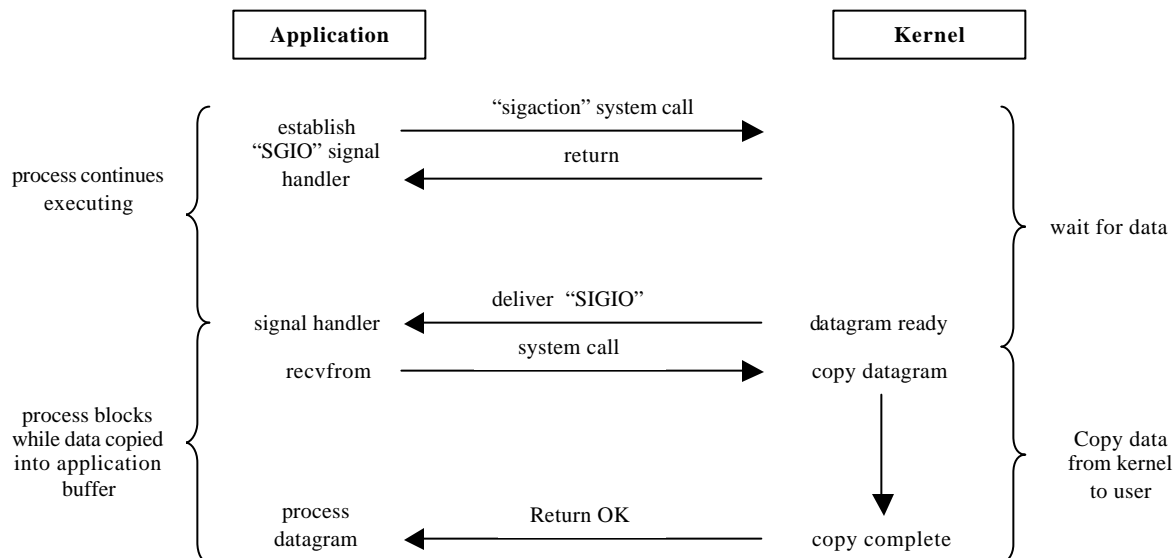
```
struct timeval {
    long tv_sec;        /* seconds */
    long tv_usec;       /* microseconds */
}
```

There are three possibilities:

1. Wait forever – return only when one of the specified descriptors is ready for I/O. For this, "timeout" argument is specified as a null pointer.
2. Wait up to a fixed amount of time – return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the "timeval" structure pointed by the "timeout" argument.
3. Do not wait at all – return immediately after checking the descriptors. This is called "polling". To specify this, "timeout" argument must point to a "timeval" structure and the timer value must be 0.

Signal-driven I/O Model:

Here, the kernel is asked to notify with the “SIGIO” signal when the descriptor is ready.



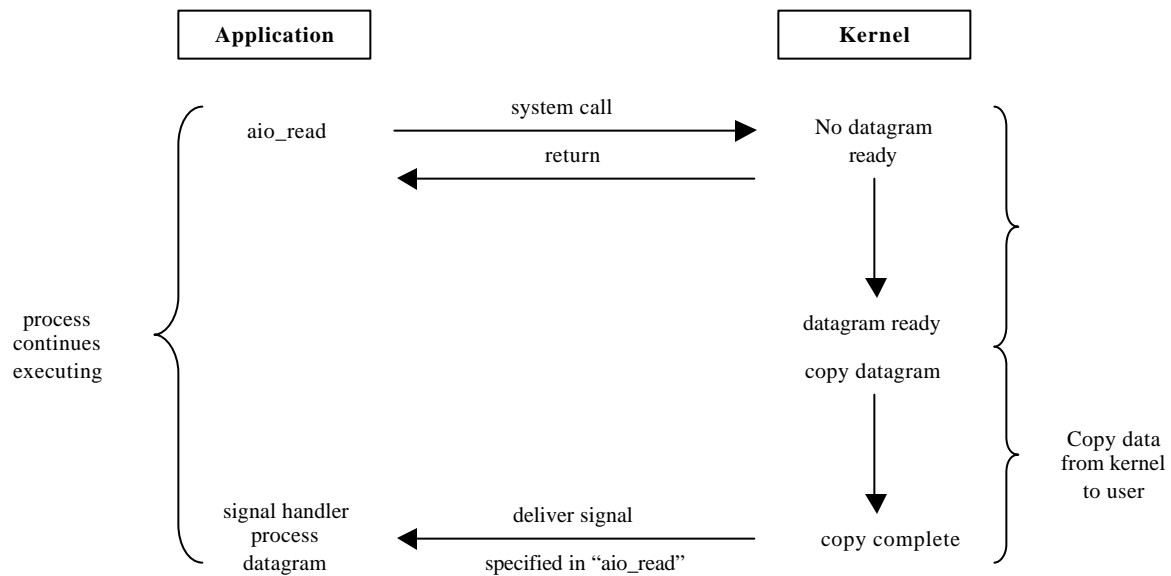
Signal-driven I/O Model

At first, the socket is enabled for signal-driven I/O and a signal handler is installed using the “sigaction” system call. There is an immediate return from “sigaction” and the process continues; i.e. it is not blocked. When the datagram is ready to be read, the “SIGIO” signal is generated for our process. Either it is possible to read the datagram from the signal handler by calling “recvfrom” and notify the main loop that the data is ready to be processed, or notify the main loop and let it read the datagram.

No matter how the signal is handled, the advantage to this model is that there is no blocking while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler (that either the data is ready to process or the datagram is ready to be read).

Asynchronous I/O Model:

Here, the kernel is asked to start the operation and notify the application when entire operation is completed. The prime difference between asynchronous I/O model and signal-driven model is that, in the later model, the kernel tells when an I/O operation can be initiated, whereas in the former one, the kernel tells when an I/O operation is completed.



Asynchronous I/O Model

Here, "aio_read" is called and the kernel descriptor, buffer pointer, buffer size, file offset are passed and the completion of an entire operation is notified. This system call returns immediately and the process is not blocked while waiting for the I/O to complete.

- ?? A "synchronous I/O operation" causes the requesting process to be blocked until that I/O operation completes.
- ?? An "asynchronous I/O operation" does not cause the requesting process to be blocked.

The first four I/O models – blocking, nonblocking, I/O multiplexing and signal driven I/O are all synchronous because the actual I/O operation "recvfrom" blocks the process. Only the asynchronous I/O model matches the asynchronous I/O definition.

Chapter – 8: Socket Options

The system calls “getsockopt” and “setsockopt” can be used to set and inspect special options associated with sockets. These might be general options for all sockets or implementation-specific options. Sample options taken from <sys/sockets.h> are SO_DEBUG and SO_REUSEADDR (allow the reuse of local addresses)

“fcntl” and “ioctl” are system calls that make it possible to control files, sockets, and devices in a variety of ways.

```
status = fcntl(sock, command, argument)
int status, sock, command, argument;
```

```
status = ioctl(sock, request, buffer)
int status, sock, request;
char *buffer;
```

The command and argument parameters for “fcntl” can be supplied with manifest constants found in “fcntl.h”. The manifest constants for “ioctl”’s request parameter are located in <sys/ioctl.h>. This parameter also specifies how the buffer argument of the call is to be used. Either one of these system calls can be used to enable asynchronous notifications on a socket. Whenever data arrives on such a socket a SIGIO signal will be delivered to the process. The process should have already declared a signal handler for this signal (see Section 6.1). This signal handler can then read the data from the socket. Program execution continues at the point of interruption.

The calling sequence

```
fcntl(sock, F_SETOWN, getpid());
fcntl(sock, F_SETFL, FASYNC);
```

enables asynchronous i/o on the given socket. The first call is necessary to specify the process to be signalled. The second call sets the descriptor status flags to enable SIGIO.

The same calling sequence can be used to make a socket non-blocking; the only change is that the flag FNDELAY is used in the second call instead of FASYNC. In this case, if a read or write operation on the socket would normally block, -1 is returned and the external system variable errno is set to EWOULDBLOCK. This error number can be checked and appropriate action taken. There are various ways to get and set the options that affect a socket:

- ?? the “getsockopt” and “setsockopt” functions
- ?? the “fcntl” options
- ?? the “ioctl” functions

“getsockopt” and “setsockopt” functions:

These two functions apply only to sockets:

```
#include <sys/socket.h>
int getsockopt (int sockfd, int level, int optname, void *optval, socklen_t
*optlen);
int setsockopt (int sockfd, int level, int optname, const void *optval,
socklen_t optlen);
```

Here, “sockfd” refers to an open socket descriptor, “level” specifies the code in the system that interprets the option: the general socket code or some protocol-specific code, “optval” is a pointer to a variable, from which the new value of the option is fetched by

“setsockopt”, or into which the current values of the option is stored by “getsockopt” and a value-result for “getsockopt”.

Generic Socket Options

SO_BROADCAST Socket Option:

This option enables or disables the ability of the process to send broadcast messages. Broadcasting is only supported for datagram sockets and only on networks that support the concept of a broadcast message, such as Ethernet, token ring, etc. It is not possible to broadcast on a point-to-point link or any connection-based transport protocol (TCP). An application must set this socket option before sending a broadcast datagram.

SO_DEBUG Socket Option:

This option is supported only by TCP. When enabled for a TCP socket, the kernel keeps track of detailed information about all the packets sent or received by TCP for the socket.

SO_DONTROUTE Socket Option:

This option specifies that outgoing packets are to bypass the normal routing mechanisms of the underlying protocol. For instance in IPv4, the packet is directed to the appropriate local interface, as specified by the network and subnet portions of the destination address. If the local interface cannot be determined from the destination address, “ENETUNREACH” is returned.

SO_ERROR Socket Option:

When an error occurs on a socket, the protocol module in a Berkeley-derived kernels sets a variable name “so_error” for that socket to one of the standard UNIX values. This is called “pending error” for the socket. The process can be immediately notified of the error in one of the following ways:

- ?? If the process is blocked in a call to “select” on the socket for either readability or writability, “select” returns with either or both conditions set.
- ?? If the process is using signal-driven I/O, the SIGIO signal is generated for either the process or the process group.

SO_KEEPALIVE Socket Option:

When the keep-alive option is set for a TCP socket and no data has been exchanged across the socket in either for two hours, TCP automatically sends a “keep-alive probe” to the peer. It is a TCP segment to which the peer must respond in either of the following ways:

- ?? The peer responds with the expected ACK. The application is not notified. TCP will then send another probe after two hours of inactivity.
- ?? The peer responds with an RST, telling the local TCP that the peer host has crashed or rebooted. The socket’s pending error is set to “ECONNRESET” and the socket is closed.
- ?? There is no response from the peer to the keep-alive probe. Berkeley-derived TCPs send 8 additional probes, 75 seconds apart, trying to obtain a response. TCP will give up if there is no response within 11minutes and 15 seconds after sending the first probe. If there is no response at all to TCP’s keep-alive probes, the socket’s pending error is set to “ETIMEDOUT” and the socket is closed. But if the socket receives an ICMP error, the corresponding error is returned instead. A common ICMP error in this scenario is “host unreachable” and the pending error is set to “EHOSTUNREACH”. This can occur either due to network failure or the remote host has crashed.

SO_LINGER Socket Option:

This option specifies how the “close” function operates for a connection-oriented protocol. By default, “close” returns immediately; but if there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer. This can be accomplished by “SO_LINGER” socket option that requires the following structure to be passed between the user process and the kernel. It is defined by including “<sys/socket.h>”.

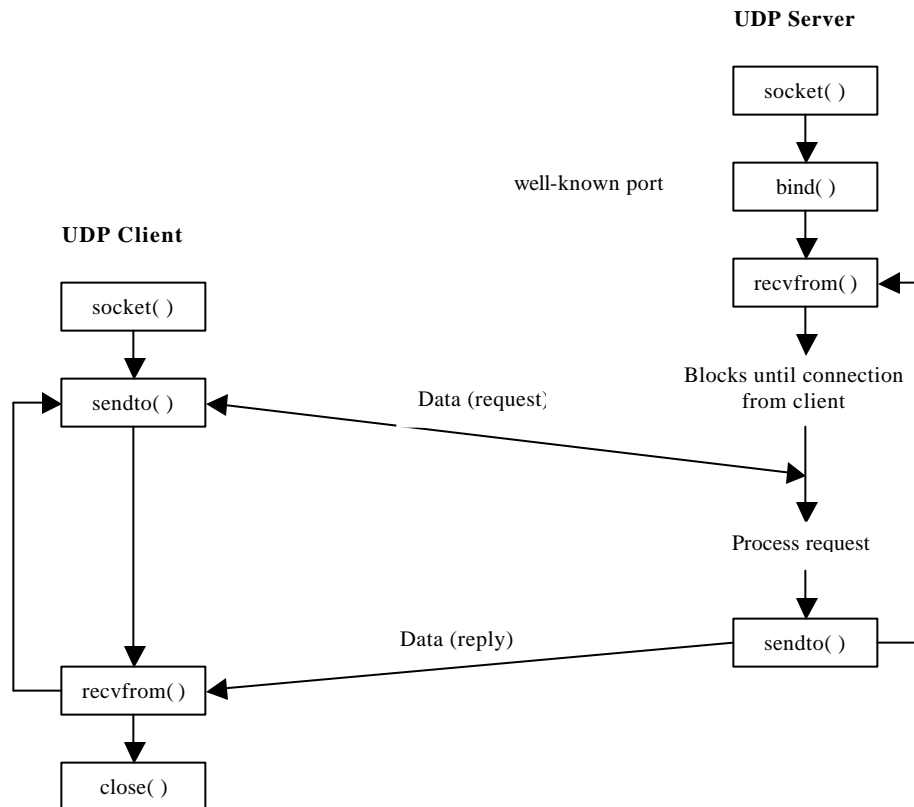
```
struct linger {  
    int l_onoff; /* 0=off, nonzero=on */  
    int l_linger; /* linger time in seconds */  
}
```

Calling “setsockopt” leads to one of the following three scenarios, depending on the values of the two structure members:

1. If “l_onoff” is 0, the option is turned off and “close” returns immediately.
2. If “l_onoff” is nonzero, and “l_linger” is zero, TCP aborts the connection when it is closed; i.e. TCP discards any data still remaining in the socket send buffer and sends RST to the peer, not the normal 4-packet connection termination sequence.
3. If “l_onoff” is nonzero and “l_linger” is nonzero, the kernel will “linger” when the socket is closed; i.e. if there is any data still remaining in the socket send buffer, the process is put to sleep until either all the data is sent and acknowledged by the peer TCP or the linger time expires.

Chapter – 9: Elementary UDP Sockets

DNS, NFS and SNMP are some examples for UDP applications. The following figure shows the function calls for a typical UDP client/server. The client does not establish a connection with the server. Instead, the client just sends a datagram into the server using the “sendto” function, which requires the address of the destination (the server) as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the “recvfrom” function, which waits until data arrives from some client. “recvfrom” returns the protocol address of the client along with the datagram, so the server can send a response to the correct client.



Socket functions for UDP client/server

“recvfrom” and “sendto” Functions:

These functions are similar to the standard “read” and “write” functions along with three additional arguments:

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
                 struct sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
               const struct sockaddr *to, socklen_t addrlen);
```

Here, the first three arguments: “sockfd, buff and nbytes” are identical to the first three arguments for “read” and “write”: descriptor, pointer to buffer to read into or write from, and number of bytes to read and write. The “recvfrom” function fills in the socket address pointed to by “from” with the protocol address of who sent the datagram. The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by “addrlen”. The “to” argument is a socket address structure containing the protocol address of where the data is to be sent. The size of this socket address structure is specified by “addrlen”. It should be noted that the final argument to “sendto” is an integer value, whereas it is a pointer to an integer value (a value-result argument) in case of “recvfrom”.

The final two arguments of “recvfrom” are similar to the final two arguments of “accept”, i.e. the contents of the socket address structure upon return call tells either who sent the datagram (in case of UDP), or who initiated the connection (in case of TCP).

The final two arguments of “sendto” are similar to the final two arguments of “connect”. The socket address structure is filled either with the protocol address of where to send the datagram (in case of UDP) or, with whom to establish a connection (in case of TCP).

Both functions return the length of the data that was read or written as the value of the function. In “recvfrom”, the return value is the amount of user data in the received datagram. Writing a datagram of length “0” is acceptable in UDP, since it is connectionless and there is no necessity for closing UDP connection. If the “from” argument to “recvfrom” is a null pointer, then the corresponding length argument (addrlen) must also be a null pointer, indicating that receiver is not interested in knowing the protocol address of who sent the data.

Chapter -10: UNIX Domain Protocols

UDP is not an actual protocol suite, but a way of performing client/server communication on a single host using the same API, which is used for clients and servers on different hosts. These are the alternatives to IPC, when the client and server are on the same host. Basically, there are two types of sockets available in the UNIX domain: stream sockets (similar to TCP) and datagram sockets (similar to UDP). UNIX domain sockets are used for the following reasons:

1. On Berkeley-derived implementations, UNIX domain sockets are often twice as fast as a TCP socket when both peers are on the same host. The X Window System is one of its examples. When an X11 client starts and opens a connection to the X11 server, the client checks the value of the DISPLAY environment variable, which refers to the server's hostname, window and screen. If the server is on the same host as the client, the client opens a UNIX domain stream connection to the server; otherwise a TCP connection to the server.
2. UNIX domain sockets are used when passing descriptors between processes on the same host.
3. Newer implementations of UNIX domain sockets provide the client's information (user ID and group ID) to the server, which can provide additional security checking.

UNIX Domain Socket Address Structure:

UNIX domain socket address structure is defined by including the <sys/un.h> header.

```
struct sockaddr_un {
    sa_family_t  sun_family;      /* AF_LOCAL */
    char         sun_path[104];   /* null-terminated pathname */
};
```

Here, the pathname stored in the "sun_path" array must be null-terminated. The macro "SUN_LEN" is provided and it takes a pointer to a "sockaddr_un" structure and returns the length of the structure. The unspecified address is indicated by a null string as the pathname, i.e. a structure with "sun_path[0]" equal to 0.

socketpair Function:

The "socketpair" function creates two sockets which are then connected together. This function applies only to UNIX domain sockets.

```
#include <sys/socket.h>
int socketpair(int family, int type, int protocol, int sockfd[2]);
```

Here, the "family" must be "AF_LOCAL" and the protocol must be "0". However, the "type" can be either "SOCK_STREAM" or "SOCK_DGRAM". The two socket descriptors that are created are returned as "sockfd[0]" and "sockfd[1]". The two created sockets are unnamed; i.e. there is no implicit "bind" involved. The result of "socketpair" with a "type" of "SOCK_STREAM" is called a "stream pipe", which is similar to a regular UNIX pipe. But a stream pipe is "full_duplex", i.e. both descriptors can be read and written.

Socket Functions:

There are several differences and restrictions in the socket functions when using UNIX domain sockets. Some of the POSIX requirements for the application of UNIX domain sockets are as follows:

1. The default file access permissions for a pathname created by "bind" should be 0777.
2. The pathname associated with a UNIX domain socket should be an absolute pathname, not a relative pathname. This is because, if the server binds a relative pathname, then

the client must be in the same directory as the server for the client's call to either "connect" or "sendto" to succeed.

3. The pathname specified in "connect" must be a pathname that is currently bound to an open UNIX domain socket of the same type (stream or datagram).
4. If a "connect" finds that the listening socket's queue is full, "ECONNREFUSED" is returned immediately.

Passing Descriptors:

Passing an open descriptor from one process to another often occurs either in the following states:

- ?? A child sharing all the open descriptors with the parent after a call to "fork".
- ?? All descriptors normally remaining open when "exec" is called.

The process opens a descriptor, calls "fork" and then the parent closes the descriptor, letting the child to handle the descriptor. This passes an open descriptor from the parent to the child. But it may be necessary for the child to open a descriptor and pass it back to the parent. Current UNIX systems enable to pass any open descriptor from one process to any other process; i.e. there is no need for the process to be related (parent and child). For this, it is necessary to establish a UNIX domain socket between the two processes and then use "sendmsg" to send a special message across the UNIX domain socket. The message is specially handled by the kernel, passing the open descriptor from the sender to the receiver. The steps involved in passing a descriptor between two processes are as follows:

1. Create a UNIX domain socket, either a stream socket or a datagram socket. If it is required to call "fork" and have the child open the descriptor and pass the descriptor back to the parent, the parent can call "socketpair" to create a stream pipe that can be used to pipe that can be used to exchange the descriptor. But if the processes are unrelated, the server must create a UNIX domain stream socket and "bind" a pathname to it, allowing the client to "connect" to that socket. The client can then send a request to the server to open some descriptor and the server can pass back the descriptor across the UNIX domain socket.
2. One process opens a descriptor by calling any of the UNIX functions that returns a descriptor such as "open, pipe, mkfifo, socket or accept".
3. The sending process builds "msg_hdr" structure containing the descriptor to be passed and then calls "sendmsg" to send the descriptor across the UNIX domain socket.
4. The receiving process calls "recvmsg" to receive the descriptor on the UNIX domain socket. It is normal for the descriptor number in the receiving process to differ from the descriptor number in the sending process.

Chapter – 11: Windows Socket

NetBIOS

Network Basic Input/Output (NetBIOS) is a standard application programming interface (API) developed for IBM in 1983. NetBIOS defines a programming interface for network communication. In 1985, IBM created the NetBIOS Extended User Interface (NetBEUI), which was integrated with the NetBIOS interface to form an exact protocol. By this time, most of the vendors implement the NetBIOS programming interface on other protocols such as TCP/IP and IPX/SPX.

Microsoft NetBIOS:

If an application is developed according to the NetBIOS specification, the application can run over TCP/IP, NetBEUI or even IPX/SPX. But for two NetBIOS applications to communicate with each other over the network, they must be running on workstations that have at least one transport protocol in common. For example, if Ram's machine has only TCP/IP installed and Shyam's machine has only NetBEUI, NetBIOS applications on Ram's machine will not be able to communicate with applications on Mary's machine.

Actually, only certain protocols implement a NetBIOS interface. By default, Microsoft TCP/IP and NetBEUI offer a NetBIOS interface, whereas IPX/SPX does not. Hence, Microsoft provides a version of IPX/SPX that is NetBIOS-capable IPX/SPX protocol. For instance, Windows 2000 offers the protocol NWLink IPX/SPX/NetBIOS Compatible Transport Protocol. In Windows 95 and Windows 98, the IPX/SPX protocol Properties dialog box has a check box that enables NetBIOS over IPX/SPX.

Moreover, NetBEUI is not a routable protocol. If there is a router between the client machine and the server machine, applications on those machines will not be able to communicate. The router will drop the packets as it receives them. TCP/IP and IPX/SPX are both routable protocols and do not have this limitation.

LANA Numbers:

In the early implementation of NetBIOS, each physical network card was assigned a unique value: LANA number i.e. LAN Adapter number. It corresponds to the unique pairings of network adapter with transport protocol. LANA numbers range from 0 to 9, and the operating system assigns them in no particular order except for LANA0, which is the default one.

NetBIOS Names:

There are two types of NetBIOS names: unique and group. A name is exactly one, such that no other process on the network can have that name registered. Machine names in Microsoft networks are NetBIOS names. When a machine boots, it registers its name with the local Windows Internet Naming Server (WINS) server, which reports error if another machine has that name in use. A WINS server maintains a list of all registered NetBIOS names. Moreover, protocol-specific information can be kept along with the name.

On the other hand, group names are used to send or receive data to multiple recipients. A group name need not be unique. Group names are used for multicast data transmissions.

A NetBIOS name is 16 characters long, with the 16th character reserved for special use, which distinguishes most Microsoft networking services. Various networking service and group names are registered with a WINS server by direct name registration from WINS-enabled computers or by broadcast on the local subnet by non-WINS-enabled computers. The "Nbtstat" command is a utility by which it is possible to obtain information about NetBIOS names that are registered on the local (or remote) computer.

For instance, "nbtstat -n" displays the information related to 4 fields: NetBIOS name, 16th Byte, Name Type (UNIQUE or GROUP) and status.

Addressing:

In IP, computers are assigned an IP address that is represented as a 32-bit quantity, formally known as an IP version 4 (IPv4) address. When a client wants to communicate with a server through TCP or UDP, it must specify the server's IP address along with a service port number. Apart from that, when servers want to listen for incoming client requests, they must specify an IP address and a port number. In Winsock, applications specify IP addresses and service port information through the SOCKADDR_IN structure, which is defined as

```
struct sockaddr_in
{
    short        sin_family;
    u_short      sin_port;
    struct       sin_addr;
    char         sin_zero[8];
};
```

The “sin_family” field must be set to “AF_INET”, which tells Winsock that the IP address family is being used. The “sin_port” defines which TCP or UDP communication port will be used to identify a server service. Applications should be particularly careful in choosing a port because some of the available port numbers are reserved for well-known services such as File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP). The port used by well-known services are controlled and assigned by the Internet Assigned Numbers Authority (IANA). The port numbers are classified into three ranges: well known, registered and dynamic or private.

?? 0-1023 are controlled by IANA, reserved for well-known services.

?? 1024-49151 are registered ports listed by the IANA and can be sued by ordinary user processes.

?? 49152-65535 are dynamic and/or private ports.

The “sin_addr” is used for storing an IP address as a 4 byte quantity, which is an unsigned long integer data type. Depending on how this field is used, it can represent a local or remote IP address. And “sin_zero” enables the “SOCKADDR_IN” structure the same size as the “SOCKADDR” structure. A useful support function “inet_addr” converts a dotted IP address to a 32-bit unsigned long integer quantity. The “inet_addr” function is defined as

```
unsigned long inet_addr (
    const char FAR *cp
);
```

Creating a Socket:

To open an IP socket using the TCP protocol, call the “socket” function or the “WSASocket” function with the address family “AF_INET” and the socket type “SOCK_STREAM”, and set the protocol field to “0” as follows:

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

```
s = WSASocket (AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
```

To open an IP socket using the UDP protocol, simply specify the socket type “SOCK_DGRAM” instead of “SOCK_STREAM” in the “socket” and “WSASocket” calls above. It is also possible to open a socket type directly over IP, which can be accomplished by setting the socket type to “SOCK_RAW”.

Name Resolution:

Winsock provides two support functions that help to resolve a host name to an IP address. The Window Sockets “gethostbyname” and “WSAAsyncGetHostByName” API functions

retrieve host information corresponding to a host name from a host database. Both functions return a "HOSTENT" structure that is defined as:

```
struct hostent
{
    char FAR *      h_name;
    char FAR * FAR  h_aliases;
    short           h_addrtype;
    short           h_length;
    char FAR * FAR * h_addr_list;
};
```

The "h_name" is the official name of the host. If the network uses the DNS, it is the Fully Qualified Domain Name (FQDN) that causes the name server to return a reply. But if the network uses a local "hosts" file, it is the first entry after the IP address. The "h_aliases" is a null-terminated array alternative name for the host. The "h_addrtype" represents the address family being returned. The "h_length" defines the length in bytes of each address in the "h_addr_list". The "h_addr_list" field is a null-terminated array of IP addresses from the host. Normally applications use the first address in the array. But if more than one address is returned, applications should randomly choose an available address.

The "gethostbyname" API function is defined as

```
struct hostent FAR * gethostbyname (
    const char FAR * name
);
```

Here, "name" represents a friendly name of the host being looked up. If this function succeeds, a pointer to a "HOSTENT" structure is returned.

The "WSAAsyncGetHostByName" API function is an asynchronous version of the "gethostbyname" function that uses Windows messages to inform an application when this function completes. "WSAAsyncGetHostByName" is defined as

```
HANDLE WSAAsyncGetHostByName(
    HWND hWnd,
    unsigned int wMsg,
    const char FAR * name,
    char FAR * buf,
    int buflen
);
```

Here, "hWnd" is the handle of the window that will receive a message when the asynchronous request completes. The "wMsg" is the Windows message to be received when the asynchronous request completes. The "name" parameter represents a user-friendly name of the host being looked up. The "buf" parameter is a pointer to the data area to receive the "HOSTENT" data.

Port numbers:

Apart from the IP address of a remote computer, an application must know the service's port number to communicate with a service running on a local or remote computer. When using TCP and UDP, applications must decide which ports they plan to communicate over. It is possible to retrieve port numbers for well-known services by calling the "getservbyname" and "WSAAsyncGetServByName" functions. These functions retrieve static information from a file named "services". In Windows95 and Windows98, the services file is located under %WINDOWS% and in WindowsNT and Windows2000; it is located under %WINDOWS%\System32\Drivers\etc. The "getservbyname" function is defined as

```
struct servent FAR * getservbyname(
    const char FAR * name,
    const char FAR * proto
);
```

Here, the “name” represents the name of the service you are looking for. The “proto” parameter optionally points to a string that indicates the protocol that the service in “name” is registered under. The “WSAAsyncGetSrvByName” function is asynchronous version of “getservbyname”.

IPX/SPX

The Internetwork Packet Exchange (IPX) protocol is commonly used with computers featuring Novell NetWare client/server networking services. IPX provides connectionless communication between two processes. Hence if a workstation transmits a data packet, there is no guarantee that the packet will be delivered to the destination. If an application needs guaranteed delivery of data and insists on using IPX, it can use a higher-level protocol over IPX, which is Sequence Packet Exchange (SPX). Here, SPX packets are transmitted through IPX. Winsock provides applications with the capability to communicate through IPX on Windows 95, Windows 98, Windows NT and Windows 2000.

Addressing:

In an IPX network, network segments are bridged together using an IPX router. Each network segment is assigned a unique 4-byte network number. As more network segments are bridged together, IPX routers manage communication between different network segments using the unique segment numbers. When a computer is attached to a network segment, it is identified using a unique 6-byte node number, which is identified using a unique 6-byte node number i.e. the physical/hardware address of the network adapter. A host is typically capable of having one or more processes forming communication over IPX. IPX uses socket numbers to distinguish communication for processes on a node.

To prepare a Winsock client or server application for IPX communication, it is necessary to set up a “SOCKADDR_IPX” structure. The “SOCKADDR_IPX” structure is defined in the “Wsipx.h” header file, which must be included by an application after including “Winsock2.h”. The “SOCKADDR_IPX” structure is defined as

```
typedef struct sockaddr_ipx
{
    short        sa_family;
    char         sa_netnum[4];
    char         sa_nodenum[6];
    unsigned short sa_socket;
}
SOCKADDR_IPX, *PSOCKADDR_IPX, FAR *LPSOCKADDR_IPX;
```

Here, “sa_family” should always be set to the “AF_IPX” value. The “sa_netnum” field is a 4-byte number that represents a network number of a network segment on a IPX network. The “sa_nodenum” field is a 6-byte number that represents host’s hardware address. The “sa_socket” field represents a socket or port used to distinguish IPX communication on a single node.

Creating a Socket:

To open an IPX socket, call the “socket” function or the “WSASocket” function with the address family “AF_IPX”, the socket type “SOCK_DGRAM” and the protocol “NSPROTO_IPX”, as follows:

```
s = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX);

s = WSASocket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX, NULL, 0, WSA_FLAG_OVERLAPPED);
```

IPX provides unreliable connectionless communication using datagram. But if an application needs reliable connectionless communication using IPX, it can use higher-level protocols over IPX, such as SPX. This can be accomplished by setting the type and protocol fields of the “socket” and “WSASocket” calls to the socket type “SOCK_SEQPACKET” or “SOCK_STREAM”, and the protocol “NSPROTO_SPX”.

If “SOCK_STREAM” is specified, data is transmitted as a continuous streams of bytes with no message boundaries same as in TCP/IP. But if “SOCK_SEQPACKET” is specified, data is transmitted with message boundaries. For instance, if a sender transmits 2000 bytes, the receiver will not return until all 2000 bytes have arrived. SPX accomplish this by setting an end-of-message bit in an SPX header. When “SOCK_SEQPACKET” is specified, the bit is respected i.e. Winsock “recv” and “WSARecv” calls will not complete until a packet is received with this bit set. But if “SOCK_STREAM” is specified, the end-of-message bit is not respected, and “recv” completes as soon as any data is received, regardless to the setting of the end-of-message bit set.

Binding a socket:

When an IPX application associates a local address with a socket using “bind”, it is not necessary to specify a network number and a node address in a “SOCKADDR_IPX” structure. The “bind” function settles these fields using the first IPX network interface available on the system. Windows 95, Windows 98, Windows NT and Windows 2000 provide a virtual internal network it is attached to. After an application successfully binds to a local interface, it is possible to retrieve local network number and node number information using the “getsockname” function as in the following code fragment:

```
SOCKET sdServer;
SOCKADDR_IPX IPXAddr;
int addrlen = sizeof(SOCKADDR_IPX);

if ((sdServer = socket (AF_IPX, SOCK_DGRAM, NSPROTO_IPX))
    == INVALID_SOCKET)
{
    printf("socket failed with error %d\n", WSAGetLastError());
    return;
}

ZeroMemory(&IPXAddr, sizeof(SOCKADDR_IPX));
IPXAddr.sa_family = AF_IPX;
IPXAddr.sa_socket = htons(5150);

if (bind(sdServer, (PSOCKADDR) &IPXAddr, sizeof(SOCKADDR_IPX))
    == SOCKET_ERROR)
{
    printf("bind failed with error %d\n",
        WSAGetLastError());
    Return;
}
```

```

}

if (getsockname((unsigned) sdServer, (PSOCKADDR) &IPXAddr, &&addrlen)
    == SOCKET_ERROR)
{
    printf("getsockname failed with error %d", WSAGetLastError());
    return;
}

//Print out SOCKADDR_IPX information returned from getsockname()

```

Initializing Winsock:

Before calling a Winsock function, it is necessary to load the correct version of the Winsock library. The Winsock initialization routine is *WSAStartup*, defined as

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSADATA)
```

The first parameter is the version of the Winsock library that is required to load. For current Win32 platforms, the latest Winsock 2 library is version 2.2. If this version to be used, either the value (0x0202) is to be specified, or the macro *MAKEWORD(2, 2)* is to be used. The high-order byte specifies the minor version number, while the low-order byte specifies the major version number.

The second parameter is a structure *WSADATA*, which is returned upon completion. *WSADATA* contains information about the version of Winsock that *WSAStartup* loaded. It is defined as

```

typedef struct WSADATA {
    WORD    wVersion;
    WORD    wHighVersion;
    char     szDescription[WSADESCRIPTION_LEN + 1];
    char     szSystemStatus[WSASYS_STATUS_LEN + 1];
    unsigned short    iMaxSockets;
    unsigned short    iMaxUdpDg;
    char FAR *    lpVendorInfo;
} WSADATA, FAR* LPWSADATA;

```

Field	Description
wVersion	The Winsock version the caller is expected to use
wHighVersion	The highest Winsock version supported by the loaded library
szDescription	A test description of the loaded library
szSystemStatus	A text string containing relevant status or configuration information
iMaxSockets	Maximum number of sockets
iMaxUdpDg	Maximum UDP datagram size
lpVendorInfo	Vendor-specific information

It should be noted that when the Winsock functions are no longer required to be called, the companion routine *WSACleanup* unloads the library and frees any resources. This function is defined as

```
int WSACleanup (void);
```

For each call to *WSAStartup*, a matching call to *WSACleanup* is required as each startup call increments the reference count to the loaded Winsock DLL, requiring an equal number of calls to *WSACleanup* to decrement the count.

Winsock 2 Protocol Information:

Winsock 2 provides a method for determining which protocols are installed on a given workstation and returning a variety of characteristics for each protocol. For instance, if TCP/IP is installed on a system, there will be two IP entries; one for TCP, which is reliable and connection-oriented and other for UDP, which is unreliable and connectionless.

The function call to obtain information on installed network protocols is *WSAEnumProtocols* and is defined as:

```
int WSAEnumProtocols (
    LPINT lpiProtocols,
    LPWSAPROTOCOL_INFO lpProtocolBuffer,
    LPDWORD lpdwBufferLength
);
```

WSAEnumProtocols returns an array of *WSAPROTOCOL_INFO* structure, which is defined as

```
typedef struct _WSAPROTOCOL_INFOW {
    DWORD dwServiceFlags1;
    DWORD dwServiceFlags2;
    DWORD dwServiceFlags3;
    DWORD dwServiceFlags4;
    DWORD dwProviderFlags;
    GUID ProviderId;
    DWORD dwCatalogEntryId;
    WSAPROTOCOLCHAIN ProtocolChain;
    int iVersion;
    int iAddressFamily;
    int iMaxSockAddr;
    int iMinSockAddr;
    int iSocketType;
    int iProtocol;
    int iProtocolMaxOffset;
    int iNetworkByteOrder;
    int iSecurityScheme;
    DWORD dwMessageSize;
    DWORD dwProviderReserved;
    WCHAR szProtocol[WSAPROTOCOL_LEN + 1];
} WSAPROTOCOL_INFOW, FAR * LPWSAPROTOCOL_INFOW;
```

The easiest way call *WSAEnumProtocols* is to make the first call with *lpProtocolBuffer* equal to *NULL* and *lpdwBufferLength* set to 0. The most commonly used field of the *WSAPROTOCOL_INFO* structure is *dwServiceFlags1*, which is a bit field for the various protocol attributes.

Property	Description
XP1_CONNECTIONLESS	The protocol provides connectionless service. If not set, the protocol also supports connection-oriented data transfers.
XP1_GUARANTEED_DELIVERY	The protocol guarantees that all data sent will reach the intended recipient.
XP1_GUARANTEED_ORDER	The protocol guarantees that the data will arrive in the order in which it was sent and that it will not be duplicated. But this does not guarantee delivery.
XP1_MESSAGE_ORIENTED	The protocol honors message boundaries.
XP1_PSEUDO_STREAM	The protocol is message-oriented but the message boundaries are ignored on the receiver side.

XP1_GRACEFUL_CLOSE	In this protocol, each party is notified of the other's intent to close the communication channel.
XP1_CONNECT_DATA	The protocol supports transferring data with the connection request.
XP1_DISCONNECT_DATA	The protocol supports transferring data with the disconnect request.
XP1_SUPPORT_BROADCAST	The protocol supports broadcast mechanism.
XP1_SUPPORT_MULTIPPOINT	The protocol supports multipoint or multicast mechanism.
XP1_UNI_SEND	The protocol is unidirectional in the send direction.
XP1_UNI_RECV	The protocol is unidirectional in the receive direction.

The other important fields are *iProtocol*, *iSocketType* and *iAddressFamily*. The *iProtocol* defines which protocol this entry belongs to. The *iSocketType* field is important if the protocol is capable of multiple behaviors, such as stream-oriented connections or datagram connections. Finally, *iAddressFamily* is used to distinguish the correct addressing structure to use for the given protocol. These three fields are very important when creating a socket for a given protocol.

Windows Sockets:

A socket is a handle to a transport provider. In Win32, a socket is not the same thing as a file descriptor and therefore is a separate type, *SOCKET*. Two functions create a socket:

```
SOCKET WSASocket (
    int af,
    int type,
    int protocol,
    LPWSAProtocolInfo lpProtocolInfo,
    GROUP g,
    DWORD dwFlags
);

SOCKET socket (
    int af,
    int type,
    int protocol,
);
```

Here, “af” is the address family of the protocol. For instance, if it is required to create either a UDP or TCP socket, the constant “AF_INET” is used to indicate the Internet Protocol (IP). The “type” is the socket type of the protocol. A socket type can be one of five values: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET, SOCK_RAW and SOCK_RDM. The “protocol” is used to qualify a specific transport if there are multiple entries for the given address family and socket type. The table below shows the values used for the address family, socket, and protocol fields for a given network transport.

Protocol	Address Family	Type	Socket Type	Protocol
Internet Protocol (IP)	AF_INET	TCP	SOCK_STREAM	IPPROTO_IP
		UDP	SOCK_DGRAM	IPPROTO_UDP
		Raw Sockets	SOCK_RAW	IPPROTO_RAW IPPROTO_ICMP
IPX/SPX	AF_NS	nwnlkipx [IPX]	SOCK_DGRAM	NSPROTO_IPX
	AF_IPX	nwnlkspix [SPX]	SOCK_SEQPACKET	NSPROTO_SPX
NetBIOS	AF_NETBIOS	Sequential Packets	SOCK_SEQPACKET	LANA number

Socket Parameters

Raw Sockets:

Raw sockets are a form of communication that allows you to encapsulate other protocols within the UDP packet, such as Internet Control Message Protocol (ICMP). ICMP's purpose is to deliver control, error and informational messages among Internet hosts. Because ICMP does not provide any data transfer facilities and is not considered to be at the same level as UDP or TCP, but is considered as at the same level of IP.

Error Checking and Handling:

The most common return value for an unsuccessful Winsock call is "SOCKET_ERROR". The constant "SOCKET_ERROR" actually is -1. While calling a Winsock function, if an error condition occurs, the function "WSAGetLastError" can be used to obtain a code that indicates specifically what had happened.

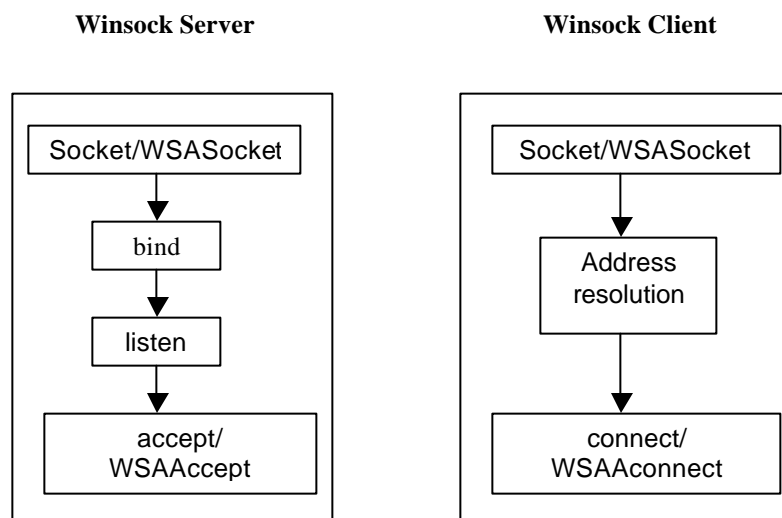
This function is defined as:

```
int WSAGetLastError (void);
```

A call to the function after an error occurs will return an integer code for the particular error that occurred. The error codes returned from "WSAGetLastError" have predefined constant values that are declared either in Winsock.h or Winsock2.h, depending on the version of Winsock.

Connection-Oriented Protocols**Server API Functions:**

A server is a process that waits for any number of client connections with the purpose of servicing their requests. A server must listen for connections on a well-known name. In TCP/IP, the name refers to the IP address of the local interface and a port number. Every protocol has a different addressing scheme and therefore a different naming method. The first step in Winsock is to bind a socket of the given protocol to its well-known name, which is accomplished with the "bind" API call. The next step is to put the socket into listening mode, which is performed with the "listen" API function. Finally, when a client attempts a connection, the server must accept the connection either with the "accept" or "WSAAccept" call. The figure below shows the basic calls a server and a client must perform in order to establish a communication channel.



Winsock basics for server and client

bind

Once the socket of a particular protocol is created, it is compulsory to bind the socket to a well-known address. The “bind” function associates the given socket with a well-known address. This function is declared as:

```
int bind (
    SOCKET          s,
    const struct sockaddr FAR * name,
    int namelen
);
```

The first parameter “s” is the socket on which it is required to wait for client connections. The second parameter is of type “struct sockaddr”, which is simply is a generic buffer. It is necessary to fill out an address buffer specific to the protocol being used and cast that as a “struct sockaddr” when calling “bind”. The Winsock header file defines the type “SOCKADDR” as “struct sockaddr”. The third parameter “namelen” is the size of the protocol-specific address structure being passed.

For example, the following code illustrates how this is done on a TCP connection:

```
SOCKET          s;
struct          sockaddr_in tcpaddr;
int             port = 5150;
s = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
tcpaddr.sin_family    = AF_INET;
tcpaddr.sin_port      = htons(port);
tcpaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
bind(s, (SOCKADDR *)&tcpaddr, sizeof(tcpaddr));
```

Here, the socket is being bound to the default IP interface on port number 5150. The call to “bind” formally establishes this association of the socket with the IP interface and port.

On error, “bind” returns “SOCKET_ERROR”. The most common error encountered with “bind” is “WSAEADDRINUSE”. While using TCP/IP, the “WSAEADDRINUSE” error indicates that another process is already bound to the local IP interface and port number.

listen

The “bind” function hardly associates the socket with a given address. The API function that tells a socket to wait for incoming connections is “listen”, which is defined as

```
int listen(
    SOCKET s,
    int backlog
);
```

Here, the “backlog” parameter specifies the maximum queue length for pending connections. This is important when several simultaneous requests are made to the server. For instance, let the “backlog” is set to 2, and if 3 client requests are made simultaneously, the first two will be placed in a “pending” queue so that the application can serve their requests, whereas, the third connection request will fail with “WSAECONNREFUSED”. It should be noted that once the server accepts a connection, the connection request is removed from the queue so that others can make a request.

One of the most common errors associated with “listen” is “WSAEINVAL”, which usually indicates that “bind” was not called before “listen”.

accept and WSAAccept

The functions “accept” or “WSAAccept” are used to accept client connections. The “accept” function is defined as

```
SOCKET accept(
    SOCK s,
    struct sockaddr FAR* addr,
    int FAR* addrlen
);
```

Here, “s” is the bound socket that is in a listening state and “sockaddr” is the address of a valid SOCKADDR_IN structure, while “addrlen” is a reference to the length of the “SOCKADDR_IN” structure.

A call to “accept” serves the first connection request in the queue of pending connections. When the “accept” function returns, the “addr” structure contains the IP address information of the client making the connection request, while the “addrlen” indicates the size of the structure. Additionally, “accept” returns a new socket descriptor that corresponds to the accepted client connection. For all subsequent operations with this client, the new socket should be used. The original listening socket is still used to accept other client connections and is still in listening mode.

Winsock2 introduced the function “WSAAccept” that has the ability to conditionally accept a connection based on the return value of a condition function. The “WSAAccept” is defined as

```
SOCKET WSAAccept
    SOCKET s,
    struct      sockaddr FAR * addr,
    LPINT addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD dwCallbackData
);
```

Here, the “lpfnCondition” argument is a pointer to a function that is called upon a request. This function determines whether to accept the client’s connection request. It is defined as

```
int CALLBACK ConditionFunc(
    LPWSABUF lpCallerId,
    LPWSABUF lpCallerData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    LPWSABUF lpCalleeId,
    LPWSAF lpCalleeData,
    GROUP FAR * g,
    DWORD dwCallbackData
);
```

Here, the “lpCallerId” is a value that contains the address of the connecting entity. The “WSABUF” structure is commonly used by many Winsock 2 functions. It is declared as

```
typedef struct __WSABUF {
    u_long len;
    char FAR * buf;
} WSABUF, FAR * LPWSABUF;
```

Here, the “len” field refers either to the size of the buffer pointed to by the “buf” field or to the amount of data contained in the data buffer “buf”. For “lpCallerId”, the “buf” pointer points to an address structure for the given protocol on which the connection is made. The “lpCallerData” contains any connection data sent by the client along with the connection request. The next two parameters “lpSQOS” and “lpGQOS” specify any quality of service (QOS) parameter that are being requested by the client, which contains information regarding bandwidth requirements for both sending and receiving data. The “lpSOS” refers to a single connection, while “lpGQOS” is used for socket groups. The “lpCalleeId” is another “WSABUF” structure containing the local address to which the client has connected. The “lpCalleeData” is the complement of “lpCallerData”. The “lpCalleeData” parameter points to a “WSABUF” structure that the server can use to send data back to the client as a part of the connection request process.

Client API Functions:

The client is much simpler and involves fewer steps to set up a successful connection. There are only three steps for a client:

1. Create a socket with “socket” of “WSASocket”.
2. Resolve the server’s name.
3. Initiate the connection with “connect” of “WSAConnect”.

connect and WSAConnect:

The “connect” function is defined as

```
int connect(
    SOCKET s,
    Const struct sockaddr FAR* name,
    int namelen
);
```

Here, the parameter “s” is the valid ICP socket on which to establish the connection, “name” is the socket address “SOCKADDR_IN” for TCP that describes the server to connect to, and “namelen” is the length of the “name” variable. The Winsock 2 defines “WSASocket” as

```
int WSAConnect(
    SOCKET S,
    const struct sockaddr FAR * name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS
);
```

Here, the first three parameters are exactly the same as the “connect” API function. The next two, “lpCallerData” and “lpCalleeData” are string buffers used to send and receive data at the time of the connection request. The “lpCallerData” parameter is a pointer to a buffer that holds data the client sends to the server with the connection request. The “lpCalleeData” parameter points to a buffer that will be filled with any data sent back from the server at the time of connection setup. And finally, the last two parameters are also same as that of “connect” function.

Data Transmission:

For sending data on a connected socket, there are two API functions: “send” and “WSASend”. Similarly, for receiving data on a connected socket: “recv” and “WSARecv” are used.

send and WSASend:

The “send” function is defined as

```
int send(
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags,
);
```

Here, the “SOCKET” parameter is the connected socket to send the data on. The second parameter “buf” is a pointer to the character buffer that contains the data to be sent. The third parameter “len” specifies the number of characters in the buffer to send. Finally, the “flags” parameter can be either “0”, “MSG_DONTROUTE”, or “MS_OOB”. The “MSG_DONTROUTE” flag tells the transport not to route the packets it sends. The “MS_OOB” flag signifies that the data should be sent out of band.

On successful return, “send” returns the number of bytes sent; otherwise, if an error occurs, “SOCKET_ERROR” is returned. A common error is “WSAECONNABORTED”, which occurs when the virtual circuit terminates because of timeout failure or a protocol error. When this error occurs, the socket should be closed, as it is no longer usable. The error “WSAECONNRESET” occurs when the application on the remote host resets the virtual circuit unexpectedly or when the remote host is rebooted. The next common error is “WSAETIMEDOUT”, which occurs when the connection is dropped because of the network failure or the remote connected system going down without notice.

The Winsock 2 version of the “send” API function “WSASend” is defined as

```
int WSASend(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Here, “lpBuffers” is a pointer to one or more “WSABUF” structures. This can be either a single structure or an array of such structures. The third parameter “dwBufferCount” indicates the number of “WSABUF” structures being passed. The “WSABUF” itself is a character buffer and the length of that buffer. The “lpNumberOfBytesSent” is a pointer to a “DWORD” and the “dwFlags” parameter is similar to that in “send” function. The last two parameters “lpOverlapped” and “lpCompletionROUTINE” are used for overlapped I/O, one of the asynchronous I/O models supported by Winsock.

The “WSASend” function sets “lpNumberOfBytesSent” to the number of bytes written. The function returns 0 on success and “SOCKET_ERROR” on any error.

recv and WSARecv:

The “recv” function is the most basic way to accept incoming data on a connected socket. This function is defined as

```
int recv(
```

```

    SOCKET s,
    char FAR* buf,
    int len,
    int flags
);

```

Here, “s” is the socket on which data will be received, “buf” is the character buffer that will receive the data, while “len” is either the number of bytes required to receive or the size of the buffer “buf”. The “flags” parameter can be one of the following values: 0, “MSG_PEEK” OR “MSG_OOB”. The “0” specifies no special actions, “MSG_PEEK” causes the data is available to be copied into the supplied receive buffer and “MSG_OOB” is same as that in “send” function.

There are some considerations when using “recv” on a datagram based socket. In case the data pending is larger than the supplied buffer, the buffer is filled with as much data as it will contain. In this case, the “recv” call generates the error “WSAEMSGSIZE”. It should be noted that the message-size error occurs with message-oriented protocols. Stream protocols buffer incoming data and will return as much data as the application requests, even if the amount of pending data is greater. Thus for streaming protocols, the “WSAEMSGSIZE” error will not be encountered.

The “WSARecv” function strengthens “recv” by adding some new capabilities such as overlapped I/O and partial datagram notifications. The “WSARecv” is defined as

```

int WSARecv(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecv,
    LPDWORD lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);

```

The “lpNumberOfBytesRecv” parameter points to the number of bytes received by this call if the receive operation completes immediately. The “lpFlags” can be one of the following values: “MSG_PEEK”, “MSG_OOB” or “MSG_PARTIAL”. The “MSG_PARTIAL” flag has several different meanings depending on where it is used or encountered. For message-oriented protocols, this flag is set upon return from “WSARecv”. In this case, subsequent “WSARecv” calls set this flag until the entire message is returned i.e. when the “MSG_PARTIAL” flag is cleared. The “MSG_PARTIAL” flag is used only with message-oriented protocols.

Breaking the Connection:

Once the socket connection is completed, it is required to close the connection and release any resources associated with that socket handle. It is one with the “closesocket” call. But “shutdown” function should be called before the “closesocket” function.

shutdown:

In order to ensure that all data an application sends is received by the peer, a well-written application is used, which notifies the receiver that no more data is to be sent. Likewise, the peer should do the same. This is known as a graceful close is performed by the “shutdown” function, defined as

```

int shutdown(
    SOCKET s,
    int how
);

```

The “how” parameter can be one of the following: “SD_RECEIVE”, “SD_SEND” or “SD_BOTH”. For “SD_RECEIVE”, subsequent calls to any receive function on the socket are disallowed. For TCP sockets, if data is queued for receive or if data subsequently arrives, the connection is reset. But for UDP sockets, incoming data is still accepted and queued. For “SD_SEND”, subsequent calls to any send function are disallowed. For TCP sockets, this causes a “FIN” packet to be generated after all data is sent and acknowledged by the receiver. Finally, “SD_BOTH” specifies to disable both send and receive.

closesocket:

The “closesocket” function closes a socket and is defined as

```
int closesocket(SOCKET s);
```

Here, calling “closesocket” releases the socket descriptor and any further calls using the socket fail with “WSAENOTSTOCK”. If there are no other references to this socket, all resources associated with the descriptor are released, including any queued data.

Connectionless Protocols

Receiver:

For a process to receive data on a connectionless socket, first create the socket with either “socket” or “WSASocket”. Next bind the socket to the interface on which the data is to be received. This is done with the “bind” function. The difference between connectionless sockets is that it is not necessary to call “listen” or “accept”. Instead, simply wait to receive the incoming data. Because there is no connection, the receiving socket can receive datagram originating from any machine on the network. The “recvfrom” function is defined as

```
int recvfrom(
    SOCKET s,
    Char FAR* buf,
    int len,
    int flags,
    struct sockaddr FAR* from,
    int FAR* fromlen
);
```

Here, most of the parameters are almost same as that in “recv” function. The “from” parameter is a “SOCKADDR” structure for the given protocol of the listening socket, with “fromlen” pointing to the size of the address structure. When the API call returns with data, the “SOCKADDR” structure is filled with the address of the workstation that sent the data. The Winsock 2 version of the “recvfrom” function is “WSARecvFrom”, which is defined as

```
int WSARecvFrom(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecv,
    LPDWORD lpFlags,
    struct sockaddr FAR * lpFrom,
    LPINT lpFromlen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED _COMPLETION_ROUTINE lpCompletionROUTINE
);
```


The difference is the use of “WSABUF” structures for receiving the data. It is possible to supply one or more “WSABUF” buffers to “WSARecvFrom” with “dwBufferCount”. The total number of bytes read is returned in “lpNumberOfBytesRecv”. When “WSARecvFrom” is called, the “lpFlags” parameter can be either “MSG_OOB”, or “MSG_PEEK”, or “MSG_PARTIAL”. While calling the function, if “MSG_PARTIAL” is specified, the provider knows to return data even if only a partial message was received. Upon return, the flag “MSG_PARTIAL” is set only if a partial message was received. Upon return, “WSARecvFrom” will set the “lpFrom” parameter (a pointer to a “SOCKADDR” structure) to the address of the sending machine. Again, “lpFromLen” points to the size of the “SOCKADDR” structure, as well as to a “DWORD”. The last two parameters are “lpOverlapped” and “lpCompletionROUTINE”, which are used for overlapped I/O.

Another method of receiving and sending data on a connectionless socket is to establish a connection. Once a connectionless socket is created, “connect” or “WSAConnect” can be called with the “SOCKADDR” parameter set to the address of the remote machine. The socket address passed into a connect function is associate with the socket so that “recv” and “WSARecv” can be used instead of “recvfrom” or “WSARecvFrom” as the sender is known.

Sender:

For sending data on a connectionless socket, there are two options: the first is simple; create a socket and call either “sendto” or “WSASendTo”. The “sendto” function is defined as

```
int sendto(
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags,
    const struct sockaddr FAR * to,
    int tolen
);
```

Here, the parameters are the same as “recvfrom” except that “buf” is the buffer of data to send and “len” indicates how many bytes to send. The “to” parameter is a pointer to a “SOCKADDR” structure with the destination address of the workstation to receive the data.

The second one, the Winsock 2 function “WSASendTo” is defined as

```
int WSASendTo(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    Const struct sockaddr FAR * lpTo,
    int iToLen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Here, before returning, “WSASendTo” sets the “lpNumberOfBytesSent” to the number of bytes actually sent to the receiver. The “lpTo” is a “SOCKADDR” structure for the given protocol, with the recipient’s address. The “iToLen” parameter is the length of the “SOCKADDR” structure.

A connectionless socket can be connected to an end-point address and data can be sent with “send” and “WSASend”. Once it is initiated, it is not possible to go back to “sendto” or “WSASendTo” with an address other than the address passed to one of the connect functions.

The only way to disconnect the socket from the destination is to call “closesocket” and create a new socket.

Chapter – 12: Winsock I/O Methods

Winsock features socket modes and socket I/O models to control how I/O is processed on a socket. A socket “mode” determines how Winsock functions behave when called with a socket. Whereas, a socket “model” describes how an application manages and processes I/O on a socket. Socket I/O models are independent of socket modes.

Winsock features two socket modes: “blocking” and “nonblocking”. All Windows platforms offer blocking and nonblocking socket operating modes. However, not every I/O model is available for every platform. Windows 95 and Windows 98 support most of the I/O models with the exception of I/O completion ports. Every I/O model is available on Windows NT and Windows 2000.

Platform	Select	WSAAsync select	WSAEvent select	Overlapped	Completion port
Windows 95 (Winsock 1)	Yes	Yes	No	No	No
Windows 95 (Winsock 2)	Yes	Yes	Yes	Yes	No
Windows 98	Yes	Yes	Yes	Yes	No
Windows NT	Yes	Yes	Yes	Yes	Yes
Windows 2000	Yes	Yes	Yes	Yes	Yes

Available socket I/O models

Socket Modes

As stated earlier, Windows sockets perform I/O operation in two socket operating modes: blocking and nonblocking. In blocking mode, Winsock calls that perform I/O – such as “send” and “recv” – wait until the operation is complete before they return to the program. But in nonblocking mode, the Winsock function returns immediately.

Blocking Mode:

Blocking sockets block for some period of time. Most Winsock applications follow a producer-consumer model in which the application reads (or writes) a specified number of bytes and performs some computation on that data.

```

SOCKET sock;
char buff[256];
int done = 0;

.....

while(!done)
[
    nBytes = recv(sock, buff, 65);
    if (nBytes == SOCKET_ERROR)
    {
        printf("recv failed with error %d\n", WSAGetLastError());
        Return;
    }
    DoComputationOnData(buf);
}

.....

```

Simple blocking socket sample

Here the problem is that the “recv” function might never return if no data is pending because the statement says to return only after reading some bytes from the system’s input buffer. Some programmers only go for peeking the necessary number of bytes in the system’s buffer by using the “MSG_PEEK” flag in “recv” or by calling “ioctlsocket” with the “FIONREAD” option. Peeking a data without actually reading the data (reading the data actually removes it from the system’s buffer) is generally not preferred. Moreover, one or more system calls are necessary just to check the number of bytes available. And further, the actual “recv” call is to be made for removing data from the system buffer.

For the solution, it is necessary to prevent the application from freezing totally due to lack to data without continually peeking at the system network buffers. One method is to separate the application into a reading thread and a computation thread. Both threads share a common data buffer. The reading thread continually reads data from the network and places it in the shared buffer. When the reading thread has read the minimum amount of data necessary for the computation thread to do its work, it can signal an event that notifies the computation thread to begin. The computation thread then removes a chunk of data from the buffer and performs the necessary computations. This method can be defined by providing two functions, one responsible for reading network data “ReadThread” and one for performing the computations on the data “ProcessThread”.

```
// initialise critical section (data) and create an auto-reset event (hEvent)
// before creating the two threads
```

```
CRITICAL_SECTION  data;
HANDLE            hEvent;
TCHAR             buff[MAX_BUFFER_SIZE];
int               nbytes;
```

```
.....
```

```
//Reader Thread
```

```
void ReadThread(void)
{
    int    nTotal = 0;
          nRead = 0;
          nLeft = 0;
          nBytes = 0;

    while (!done)
    {
        nTotal = 0;
        nLeft = NUM_BYTES_REQUIRED;
        while (nTotal != NUM_BYTES_REQUIRED)
        {
            EnterCriticalSection(&data);
            nRead = recv(sock, &(buff[MAX_BUFFER_SIZE - nBytes]), nLeft);
            if (nRead == -1)
            {
                printf("error\n");
                ExitThread();
            }
            nTotal += nRead;
            nLeft -= nRead;

            nBytes += nRead;
```

```

        LeaveCriticalSection(&data);
    }
    SetEvent(hEvent);
}

//Computation Thread

void ProcessThread(void)
{
    WaitForSingleObject(hEvent)

    EnterCriticalSection(&data);
    DoSomeComputationOnData(buff);
//Remove the processed data from the input buffer, and shift the remaining
//data to the start of the array
    nBytes -= NUM_BYTES_REQUIRED;

    LeaveCriticalSection(&data);
}

```

Multithreaded blocking sockets example

One demerit of blocking sockets is that communicating via more than one connected socket at a time becomes difficult for the application. Using the above method, the application could be modified to have a reading thread and a processing thread per connected socket. But it further increases the overhead in using resource.

Chapter- 13: Socket I/O Models

Basically, there are 5 types of socket I/O models which allow Winsock application to manage I/O: “select, WSAAsyncSelect, WSAEventSelect, overlapped and completion port”.

a) The select Model:

It is the most widely available I/O model in Winsock. It focuses on using the “select” function to manage I/O. This model’s design is originated on Unix-based computers featuring Berkeley socket implementations. The “select” function can be used to determine whether there is data on a socket and whether a socket can be written to. This prevents an application from blocking on an I/O bound call such as “send” or “recv” when a socket is in a blocking mode. The “select” function blocks for I/O operations until the conditions specified as parameters are met. The “select” function is defined as

```

int select(
    int nfds,
    fd_set FAR * readfs,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR* timeouit
);

```

Here, the first parameter “nfds” is ignored and is included only for compatibility with Berkeley socket applications. The “fd_set” data type represents a collection of sockets. There are three “fd_set” parameters: one for checking readability (readfds), one for writability (writefds), and one for out-of-band data (exceptfds). Finally, “timeval” is a pointer to a “timeval”

structure that determines how long the “select” function will wait for I/O to complete. The “timeval” structure is defined as

```
struct timeval {
    long    tv_sec;
    long    tv_usec;
};
```

Here, “tv_sec” indicates how long to wait in seconds; the “tv_usec” indicates how long to wait in microseconds.

b) The WSAAsyncSelect Model:

Winsock provides an asynchronous I/O model that allows an application to receive Windows message-based notification of network events on a socket. This is accomplished by calling the “WSAAsyncSelect” function after creating the socket.

Message Notification: To use the “WSAAsyncSelect” model, an application must first create a window using the “CreateWindow” function and supply a window procedure (“winproc”) support function for this window. Once the window infrastructure have been setup, it is possible to create sockets and turning on window message notification by calling the “WSAAsyncSelect” function, which is defined as

```
WSAAsyncSelect(
    SOCKET s,
    HWND hWnd,
    u_int wMsg,
    long lEvent
);
```

Here, “s” represents the socket, “hWnd” is a window handle identifying the window or the dialog box that receives a message when a network event occurs, “wMsg” identifies the message to be received when a network event occurs. This message is posted to the window that is identified by the “hWnd” window handle.

When an application calls “WSAAsyncSelect” on a socket, the socket mode automatically changes from blocking to nonblocking mode. Consequently, if a Winsock I/O call such as “WSARecv” is called and has to wait for data, it will fail with error “WSAEWOULDBLOCK”. In order to avoid this error, applications should rely on the user-defined window message specified in the “wMsg” parameter to indicate when network event types occur on the socket.

Event Type	Description
FD_READ	The application wants to receive notification of readiness for reading.
FD_WRITE	The application wants to receive notification of readiness for writing.
FD_OOB	The application wants to receive notification of the arrival of out-of-band data.
FD_ACCEPT	The application wants to receive notification of incoming connections.
FD_CONNECT	The application wants to receive notification of a completed connection.
FD_CLOSE	The application wants to receive notification of socket closure.
FD_QOS	The application wants to receive notification of socket Quality of Service changes.

Network event types for the “WSAAsyncSelect” function

As an application successfully calls “WSAAsyncSelect” on a socket, the application begins to receive network event notification as Window messages in the window procedure associate with the “hWnd” parameter. A window procedure is defined as

```
LRESULT CALLBACK WindowProc(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
);
```

Here, “hWnd” is a handle to the window that invoked the window procedure. The “uMsg” indicates which message needs to be processed. In this case, it is the message defined in the “WSAAsyncSelect” call. The “wParam” identifies the socket on which a network event has occurred. It is important if there is more than one socket assigned to this window procedure. And “lParam” specifies the network event that has occurred.

c) The WSAEventSelect Model:

The “WSAEventSocket” model allows an application to receive event-based notification of network events on one or more sockets. This model is similar to the “WSAAsyncSelect” model; where only the difference is that here, network events are posted to an even object handle instead of a window procedure.

Event notification: The event notification requires an application to create an event object for each socket used by calling the “WSACreateEvent” function, which is defined as

```
WSAEVENT WSACreateEvent (void);
```

The “WSACreateEvent” function simply returns an event object handle, which should be associated with a socket and register the type of network event. This is accomplished by calling the “WSAEventSelect” function, which is defined as

```
WSAEventSelect (
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents
);
```

Here, “hEventObject” represents the event object, obtained with “WSACreateEvent” and “lNetworkEvents” represents a bitmask that specifies a combination of network event types, in which the application is interested in.

The event created for “WSAEventSelect” has two operating states and two operating modes. The operating states are known as “signaled” and “nonsignaled”. The operating modes are known as “manual reset” and “auto reset”. “WSACreateEvent” initially creates event handles in a nonsignaled operating state with a manual reset operating mode. As network events trigger an event object associated with a socket, the operating state changes from nonsignaled to signaled state. Since the event object is created in a manual reset mode, an application is responsible for changing the operating state from signaled to nonsignaled after processing an I/O request. This can be accomplished by calling the “WSAResetEvent” function which is defined as

```
BOOL WSAResetEvent (
    WSAEVENT hEvent
);
```

Here, “hEvent” is an event handle that returns TRUE or FALSE based on the success or failure of the call. When an application is finished with an event object, it should call the

“WSACloseEvent” function to free the system resources used by an event handle. The “WSACloseEvent” function is defined as

```
BOOL WSACloseEvent (
    WSAEVENT hEvent
);
```

This function also takes an event handle as its only parameter the returns either TRUE or FALSE if successful or failure respectively.

Once a socket is associated with an event object handle, the application can begin processing I/O by waiting for network events to trigger the operating state of the event object handle. The “WSAWaitForMultipleEvents” function is designed to wait on one or more event object handles and returns either when one or all of the specified handles are in the signaled state or when a specified timeout interval expires. The “WSAWaitForMultipleEvents” function is defined as

```
WSAWaitForMultipleEvents (
    DWORD cEvents,
    const WSAEVENT FAR * lphEvents,
    BOOL fWaitAll,
    DWORD dwTimeout,
    BOOL fAlertable
);
```

Here, “cEvents” and “lphEvents” define an array of “WSAEVENT” objects, in which “cEvents” represents the number of event objects in the array and “lphEvents” is a pointer to the array. “WSAWaitForMultipleEvents” can support only a maximum of “WSA_MAXIMUM_WAIT_EVENTS” objects, which is defined as 64. Hence, this I/O model is capable of supporting only a maximum of 64 sockets at a time for each thread that makes the “WSAWaitForMultipleEvents” call. “fWaitAll” specifies how “WSAWaitForMultipleEvents” waits for objects in the event array. When all event objects in the “lphEvents” array are signaled, the function returns TRUE. But when any one of the event objects is signaled, the function returns FALSE. Typically, applications set this parameter to FALSE and service on one socket event at a time. The “dwTimeout” parameter specifies how long “WSAWaitForMultipleEvents” will wait for a network event to occur. The function returns if the interval expires. Finally, “fAlertable” can be ignored when “WSAEventSelect” model is being used. It should be set to “FALSE”.

d) The Overlapped Model:

In Winsock, the overlapped I/O model offers applications better system performance than any of the I/O models. The basic design of the overlapped model allows an application to post one or more Winsock I/O requests at a time using an overlapped data structure. Later on, the application can service the submitted requests after they have completed. This model is available on most of the Windows platform. The overall design of the model is based on the Win32 overlapped I/O mechanisms available for performing I/O operations on devices using the “ReadFile” and “WriteFile” functions.

To use the overlapped I/O model on a socket, at first a socket is to be created by using the flag “WSA_FLAG_OVERLAPPED” as follows:

```
s = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
```

If a socket is created using the “socket” function instead of the “WSASocket” function, “WSA_FLAG_OVERLAPPED” is implied. After creating a socket, it has to be bound to a local interface, where overlapped I/O operations can be commenced by calling the Winsock functions listed below and specifying an optional “WSAOVERLAPPED” structure.

```
?? WSASend
?? WSASentTo
?? WSARcv
```

```

?? WSARcvFrom
?? WSAIoctl
?? AcceptEx
?? TransmitFile

```

Since each one of these functions is associated with sending data, receiving data, and accepting connections on a socket, this activity can potentially take a long time to complete. Hence each function can accept a “WSAOVERLAPPED” structure as a parameter. When these functions are called with a “WSAOVERLAPPED” structure, they complete immediately. They rely on the “WSAOVERLAPPED” structure to manage the return of the return of an I/O request. There are two methods for managing the completion of an overlapped I/O request: an application can wait for “event object notification”, or it can process completed requests through “completion routines”.

Event notification: The event notification method of overlapped I/O requires associating Win32 event objects with “WSAOVERLAPPED” structures. When I/O calls like “WSASend” and “WSARcv” are made using a “WSAOVERLAPPED” structure, they return immediately. The “WSAOVERLAPPED” structure provides the communication medium between the initiation of an overlapped I/O request and its subsequent completion and is defined as

```

typedef struct _WSAOVERLAPPED {
    DWORD      Internal;
    DWORD      InternalHigh;
    DWORD      Offset;
    DWORD      OffsetHigh;
    WSAEVENT   hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;

```

Here, the “Internal, InternalHigh, Offset, and OffsetHigh” fields are all used internally by the system and should not be manipulated or used directly by an application. The “hEvent” field is a special field that allows an application to associate an event object handle with a socket. The “WSACreateEvent” function is used to create an event object handle. When an overlapped I/O request finally completes, an application is responsible for retrieving the overlapped results. In the event notification method, Winsock will change the event-signaling state of an event object from nonsignaled to signaled when an overlapped request finally completes.

Completion routines: Completion routines are those functions which are optionally passed to an overlapped I/O request and that the system invokes when an overlapped I/O request completes. Their primary role is to service a completed I/O request using the caller’s thread. Moreover, applications can continue overlapped I/O processing through the completion routine. To use completion routines for overlapped I/O requests, an application must specify a completion routine, along with a “WSAOVERLAPPED” structure, to an I/O bound Winsock function. A completion routine must have the following function prototype:

```

void CALLBACK CompletionROUTINE(
    DWORD dwError,
    DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwFlags
);

```

Here, “dwError” specifies the completion status for the overlapped operation, “cbTransferred” specifies the number of bytes that were transmitted during the overlapped operation, “lpOverlapped” is the “WSAOVERLAPPED” structure passed into the originating I/O call and “dwFlags” parameter is not used and will be set to 0.

e) The Completion Port Model:

The completion port model is the most complicated I/O model. However, it offers the best system performance possible when an application has to manage many sockets at once. It is available only on Windows NT and Windows 2000. This model is considered to be used only when an application requires to manage hundred or even thousands of sockets simultaneously and an application requires to operate efficiently when more CPUs are added to the system. If high-performance servers for Windows NT or Windows 2000 (which are expected to service many socket I/O requests, web server for an instance) are required to be developed, completion port model is the best choice.

The completion port model requires creating a Win32 completion port object that will manage overlapped I/O requests using a specified number of threads to service the completed I/O overlapped requests. This is accomplished by calling the "CreateIoCompletionPort" function, which is defined as

```
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    DWORD CompletionKey,
    DWORD NumberOfConcurrentThreads
);
```

This function is actually used for two distinct purposes:

- ?? to create a completion port object
- ?? to associate a handle with a completion port

Here, the first three parameters are ignored; the only one to be concerned is "NumberOfConcurrentThreads" that defines the number of threads that are allowed to execute concurrently on a completion port. The value 0 for this parameter tells the system to allow as many threads as there are processors in the system.

Chapter – 14: Systems Network Architecture

(Systems Network Architecture) is IBM's mainframe network standards introduced in 1974. With the rise of personal computers, workstations, and client/server computing, the need for a peer-based networking strategy was addressed by IBM with the creation of Advanced Peer-to-Peer Networking (APPN) and Advanced Program-to-Program Computing (APPC). Basically a centralized architecture with a host computer controlling many terminals, enhancements, such as APPN and APPC (LU 6.2), have adapted SNA to today's peer-to-peer communications and distributed computing environment.

SNA was developed in the 1970s with an overall structure that parallels the OSI reference model. With SNA, a mainframe running Advanced Communication Facility/Virtual Telecommunication Access Method (ACF/VTAM) serves as the hub of an SNA network. ACF/VTAM is responsible for establishing all sessions and for activating and deactivating resources.

Following are some of SNA's basic concepts.

a) Nodes and Data Links

In SNA, nodes are end points or junctions, and data links are the pathways between them. Nodes are defined as Type 5 (hosts), Type 4 (communications controllers) and Type 2 (peripheral; terminals, PCs and midrange computers). Type 2.0 nodes can communicate only with the host, and Type 2.1 nodes can communicate with other 2.1 nodes (peer-to-peer) without going to the host. Data links include high-speed local channels, the SDLC data link protocol and Token Ring.

b) SSCPs, PUs and LUs

The heart of a mainframe-based SNA network is the SSCP (System Services Control Point) software that resides in the host. It manages all resources in its domain. Within all nodes of an SNA network, except for Type 2.1, there is PU (Physical Unit) software that manages node resources, such as data links, and controls the transmission of network management information. In Node Type 2.1, Control Point software performs these functions. In order to communicate user data, a session path is created between two end points, or LUs (Logical Units). When a session takes place, an LU-LU session is established between an LU in the host (CICS, TSO, user application, etc.) and an LU in the terminal controller or PC. An LU 6.2 session provides peer-to-peer communication and lets either side initiate the session.

c) VTAM and NCP








VTAM (Virtual Telecommunications Access Method) resides in the host and contains the SSCP, the PU for the host, and establishes the LU sessions within the host. NCP (Network Control Program) resides in the communications controller (front end processor) and manages the routing and data link protocols, such as SDLC and Token Ring.

d) SNA Layers

SNA is implemented in functional layers starting with the application that triggers the communications down to the bottom layers which transmit packets from station to station. This layering is called a "protocol stack." The SNA stack is compared with the OSI model below. Although SNA had major influence on the OSI model, there are differences in implementation.

From Computer Desktop Encyclopedia
 © 1998 The Computer Language Co. Inc.



OSI MODEL			SNA
7	 Application Layer Type of communication: E-mail, file transfer, client/server.		Transaction Services
6	 Presentation Layer Encryption, data conversion: ASCII to EBCDIC, BCD to binary, etc.		Presentation Services
5	 Session Layer Starts, stops session. Maintains order.		Data Flow Control
4	 Transport Layer Ensures delivery of entire file or message.		Transmission Control
3	 Network Layer Routes data to different LANs and WANs based on network address.		Path Control
2	 Data Link (MAC) Layer Transmits packets from node to node based on station address.		Data Link Control
1	 Physical Layer Electrical signals and cabling.		Physical Control

Chapter – 15: Novell NetWare

NetWare Security

Novell NetWare implements 4 types of security to restrict or allow access to a file server, its directories and files:

- ?? a user ID and password are required for file server access
 - ?? “trustee rights” can be granted to a person or a group of persons, which can allow or disallow various levels of access to a directory and its subdirectories
 - ?? each directory has security restrictions that apply to that its subdirectories
 - ?? a file can be marked Read-only, to prevent unintentional modification
- Each directory has a “Maximum Rights Mask” that represents the highest level of privilege any of the directory’s trustees can be granted. Each directory can have as many as five trustees, and each trustee can have the following rights. A user can
- ?? read files
 - ?? write files
 - ?? open existing files
 - ?? create new files
 - ?? delete existing files
 - ?? delete, create or rename subdirectories and set trustee rights and directory rights in the directory and its subdirectories
 - ?? search for files in the directory
 - ?? modify file attributes

NetWare Fault Tolerance

There are two versions of NetWare available: Advanced NetWare 286 and SFT Netware 286 (System Fault Tolerance). Both versions employ strategies and techniques that minimize and transparently handle the disk surface’s failure to correctly record data. Furthermore, SFT provides “disk mirroring” and “disk duplexing”, the mechanisms for maintaining duplicate copies of disk data on two separate disks.

The NetWare operating system is also designed to recognize signals from an uninterruptible power supply through UPS monitoring, which knows when the UPS is supplying power and notifies users of how much time they have left before the UPS batteries run down. If AC power is not restored within the time period, NetWare closes any open files and shuts itself down gracefully.

Finally, SFT NetWare offers the NetWare Transaction Tracking System (TTS). An application programmed to use TTS can treat a series of database updates as a single, atomic operation (either all of the updates are made or none of them is made). A system failure in the midst of multiple file update does not cause inconsistencies between the files.

NetWare Workstation

IPX and NET3 are the two components that run on each Novell NetWare workstation. IPX manages the PC-to-PC and PC-to-File Server Communication by implementing IPX/SPX communication protocol. NET3 is the shell/redirector that shunts DOS file requests to and from the file server by issuing commands to IPX. Collectively, these components make the file server’s disks and printers look like DOS-managed peripherals. IPX takes about 19 kb of memory whereas NET3 about 38 kb.

It is not necessary to run NetBIOS on a NetWare workstation, because the NetWare shell uses IPX to communicate with the file server. If NetBIOS is required, Novell supplies a NetBIOS emulator that can be loaded on top of IPX, which converts NetBIOS commands into IPX commands for transmission across the network. The NetWare NetBIOS Emulator add roughly 20 kb of memory to the resident portion of NetWare.

Performance

As many as 100 workstations can be concurrently logged into each NetWare 286 file server, whereas NetWare 386 supports 4 gb of memory for caching and as many as 250 users can be logged into a server and as much as 32 tb of disk storage can reside on a single server; each file can be as much as 4 gb. And as many as 100,000 files can be opened concurrently. NetWare 386 incorporates the concept of NetWare Loadable Modules (NLMS), software that can be loaded into the file server even while the server is still running or vice-versa. Moreover, Novell offers the Developer's Workbench, which consists of C compiler, linker, symbolic debugger, libraries of LAN-related code and NetWare RPC.