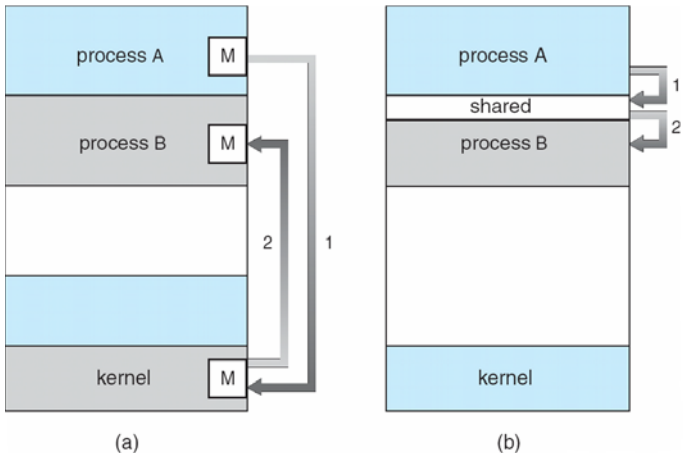
IPC
Inter-Process Communication

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models



Communicating Thru the Internet

- In most implementations, TCP/IP protocol stack is part of the computer's system software (e.g., MS Windows, LINUX).
  - The TCP/IP software operates in a multi-vendor environment.
    - Does not depend on any special vendor's internal data representation.
    - Does not use any feature that exists only on a particular computer system.
  - Basically, the software promises to send TCP segments **reliably** from one source to a destination.
  - too primitive for an application programmer
    - If a programmer is expected to fill up every field in a TCP segment, the programming effort is too complex.
  - It is better to have a set of programmer-friendly Application Program Interface (API).

The TCP/IP Standard

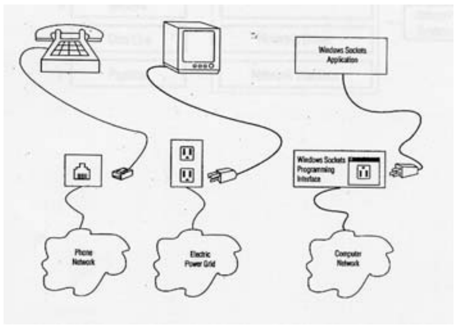
- The TCP/IP standards do not specify the details of how application software interfaces with TCP/IP protocol software; they only suggest the required functionality.
  - Advantages
    - flexibility and tolerance for different operating systems
  - Disadvantages
    - The interface details are different for different OS's.
    - Widely used APIs for TCP/IP:
      - Berkeley UNIX socket interface
      - System V UNIX Transport Layer Interface (TLI)
      - Windows Sockets Interface (WinSock, a Windows version of sockets)

Interface functionality suggested by TCP/IP

- Allocate local resources for communication.
- Specify local and remote communication endpoints.
- Initiate a connection (client side).
- Wait for an incoming connection (server side).
- Send or receive data.
- Determine when data arrives.
- Generate urgent data.
- Handle incoming urgent data.
- Terminate a connection gracefully.
- Handle connection termination from the remote site.
- Abort communication.
- Handle error conditions or a connection abort.
- Release local resources when communication finishes.

The Socket Interface

- The socket interface for TCP/IP was first developed on Berkeley UNIX (**BSD**) and later ported to other platforms.
- A socket acts as an endpoint of communication.
- In Mircosoft Windows, the socket interface is called Windows Socket, or **WinSock**, or Windows Socket API (application program interface).

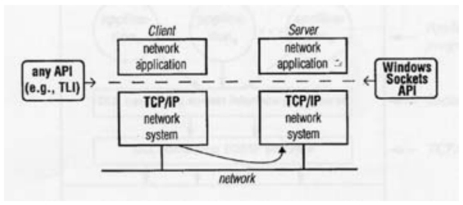

Windows Sockets Application
Windows Sockets Programming Interface
Phone Network
Electric Power Grid
Computer Network

WinSock Function Library

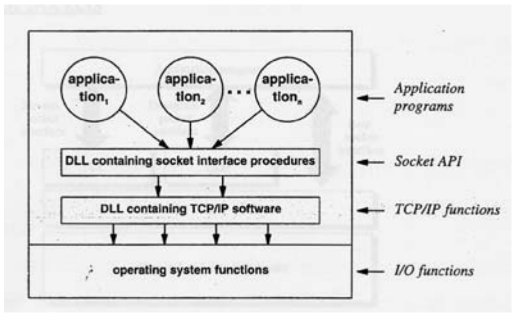
- Primary socket functions: perform specific operations to interact with the TCP/IP protocol:
  - a function that requests the TCP/IP software to establish a TCP connection to a remote server;
  - a function that requests the TCP/IP software to send data.
- Other library functions: help the programmer
  - convert a dotted decimal IP address to 32 bits.

WinSock

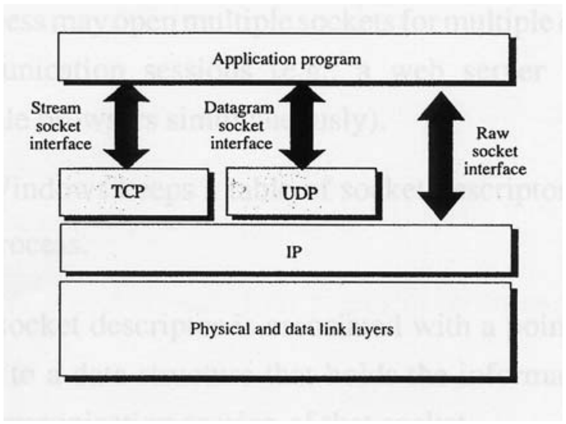
- The library of socket functions is located in a **dynamic linked library (DLL).**
  - A DLL is a library that is loaded into main memory only when the library is first used.
- WinSock is an interface but it is not a protocol.
  - If the client and the server use the same protocol suite (TCP/IP), then they can communicate even if they use different application program interfaces:



- Windows places a single copy of socket functions and TCP/IP code in memory.
  - All applications share these functions/code:

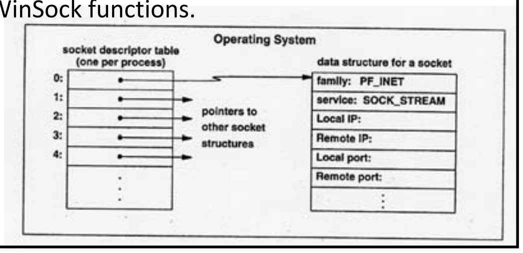


Types of Sockets



Socket Descriptor

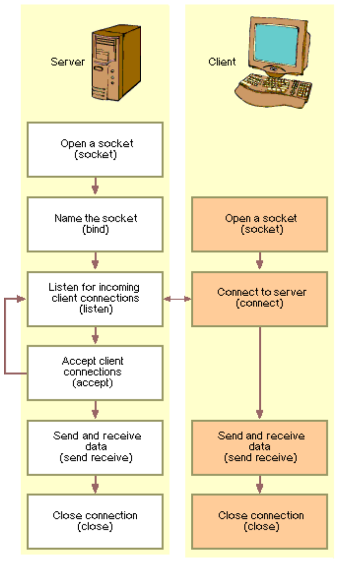
- Each socket is identified by an integer called socket descriptor, which is an unsigned integer.
- A process may open multiple sockets for multiple concurrent communication sessions (e.g., a web server is serving multiple browsers simultaneously).
    - Windows keeps a table of socket descriptors for each process.
- Each socket descriptor is associated with a pointer, which points to a data structure that holds the information about the communication session of that socket.
- The data structure contains many fields, and they will be filled as the application calls additional WinSock functions.


Operating System
socket descriptor table (one per process)
0:
1:
2:
3:
4:
pointers to other socket structures
data structure for a socket
family: PF_INET
service: SOCK_STREAM
Local IP:
Remote IP:
Local port:
Remote port:

Creating a TCP Stream Socket App

- There are two distinct types of socket network applications:

  Server and Client.


Server
Client
Open a socket (socket)
Name the socket (bind)
Listen for incoming client connections (listen)
Accept client connections (accept)
Send and receive data (send receive)
Close connection (close)
Open a socket (socket)
Connect to server (connect)
Send and receive data (send receive)
Close connection (close)

Server/Client

Creating a Socket

The **socket** function creates a socket that is bound to a specific service provider.

```
SOCKET socket(
int af,   // Address family specification.
int type,   // Type specification for the new socket
int protocol   // Protocol to be used with the socket that is specific
               // to the indicated address family.

);
```

Return Values

If no error occurs, **socket** returns a descriptor referencing the new socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code can be retrieved by calling **WSAGetLastError.**

```
winSocket = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
```

server

Binding a Socket

The **bind** function associates a local address with a socket.

```
int bind(  SOCKET s,  //Descriptor identifying an unbound socket.
const struct sockaddr* name,//Address
int namelen  // Length in bytes
);
```

Return Values
If no error occurs, **bind** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code can be retrieved by calling **WSAGetLastError**

```
sockaddr_in service;
service.sin_family = AF_INET;
service.sin_addr.s_addr = inet_addr( "127.0.0.1" );
service.sin_port = htons( 27015 );
 bind( m_socket, (SOCKADDR*) &service, sizeof(service) );
```

server

Listening on a Socket

The **listen** function places a socket in a state in which it is listening for an incoming connection.

```
int listen(
  SOCKET s,
  int backlog //Maximum length of the queue of pending connections
);
```

If no error occurs, **listen** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastErrorr**.

client

Connecting to a Socket

The **connect** function establishes a connection to a specified socket

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

Parameters

s [in] Descriptor identifying an unconnected socket.

name [in] Name of the socket in the **sockaddrr** structure to which the connection should be established.

namelen [in] Length of *name*, in bytes

connect(m_socket, (SOCKADDR*) &clientService, sizeof(clientService))

server

Accepting Connection

The **accept** function permits an incoming connection attempt on a socket.

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

Parameters

s [in] Descriptor that identifies a socket that has been placed in a listening state with the **listen** function.
The connection is actually made with the socket that is returned by **accept**.

addr [out] Optional pointer to a buffer that receives the address of the connecting entity

addrlen [in, out] Optional pointer to an integer that contains the length of *addr*.

Server/Client

Sending and Receiving Data

The **send** function sends data on a connected socket.

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

The **recv** function receives data from a connected or bound socket.

```
int recv(
    SOCKET s,
    char* buf,
    int len,
    int flags
);
bytesSent = send( ConnectSocket, sendbuf, strlen(sendbuf), 0 );
bytesRecv = recv( ConnectSocket, recvbuf, 32, 0 );
```

Server/Client

Sending and Receiving Data (Cont'd)

Parameters

s [in] Descriptor identifying a connected socket.

buf [in] Buffer containing the outgoing/incoming data.

len [in] Length of the data in *buf*, in bytes.

flags [in] Indicator specifying the way in which the call is made.

Return Values

If no error occurs:

send returns the total number of bytes sent, which can be less than the number indicated by **len**.

recv returns the number of bytes received. If the connection has been gracefully closed, the return value is zero.

Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Server/Client

Terminating the Communication

The **shutdown** function disables sends or receives on a socket.

int shutdown(

SOCKET *s*,

int *how*);

How:

Flag that describes what types of operation will no longer be allowed.

SD_RECEIVE: subsequent calls to the recv function on the socket will be disallowed.

SD_SEND: subsequent calls to the send function are disallowed.

SD_BOTH: disables both sends and receives as described above.

The shutdown function does not close the socket.

Any resources attached to the socket will *not* be freed until **closesocket** is invoked.

Server/Client

Terminating the Communication (cont'd)

The closesocket function closes an existing socket.

int closesocket(

SOCKET *s*

);

Remarks:

The **closesocket** function closes a socket. Use it to release the socket descriptor **s** so that further references to *s* fail with the error. If this is the last reference to an underlying socket, the associated naming information and queued data are discarded. Any pending blocking, asynchronous calls issued by any thread in this process are canceled without posting any notification messages. An application should always have a matching call to **closesocket** for each successful call to **socket** to return any socket resources to the system.