Windows Socket Getting Started

Prof. Lin Weiguo

Nov. 2015

Note

- You should not assume that an example in this presentation is complete. Items may have been selected for illustration. It is best to get your code examples directly from the textbook or course website and modify them to work. Use the lectures to understand the general principles.

Outline

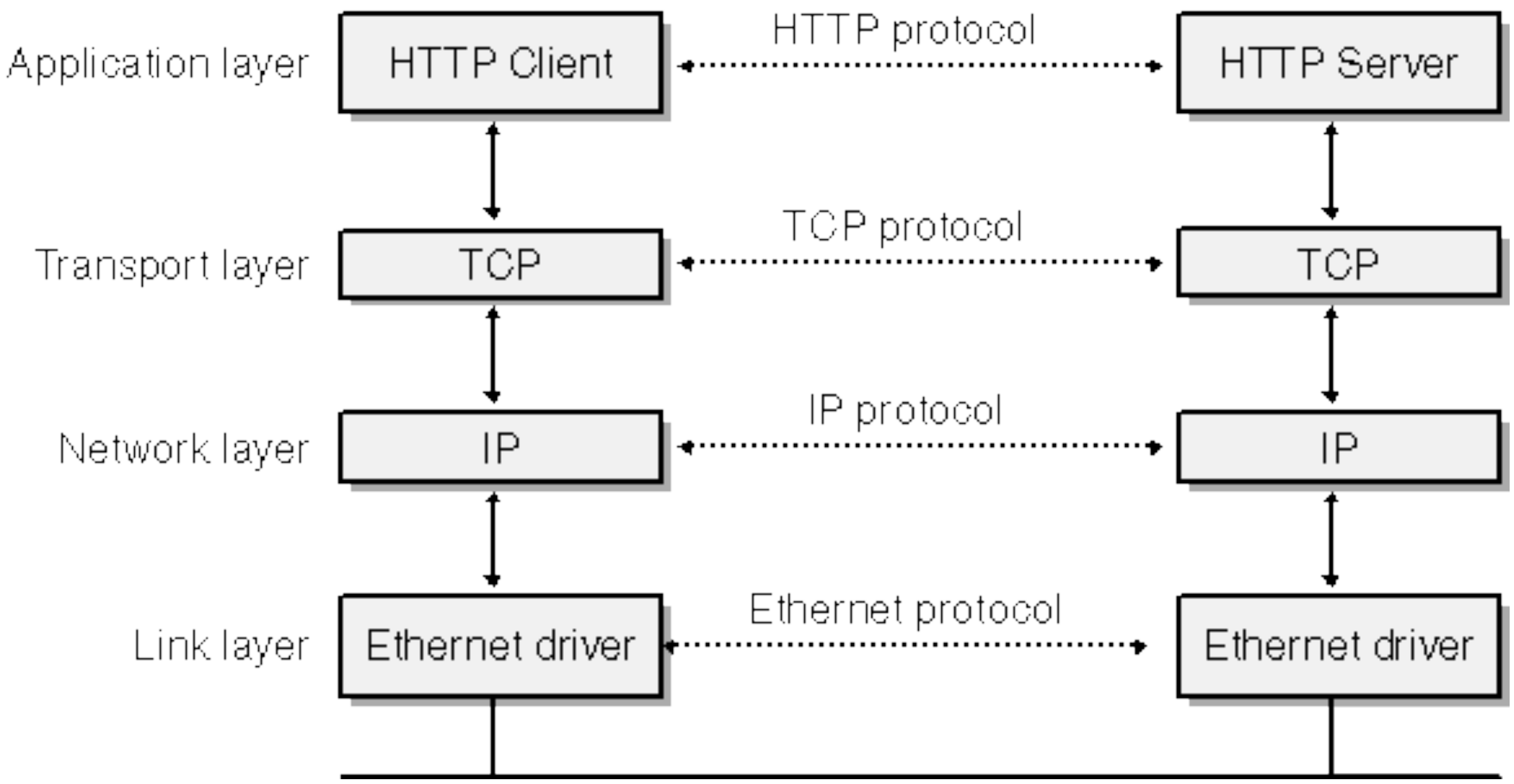
- Internet Primer
- Introduction to Windows Socket
- Basic Windows Socket Programming

Internet Primer

Internet Primer

- You can't write a good Winsock program without ***understanding the concept of a socket***, which is used to send and receive packets of data across the network.
- To fully understand sockets, you need a thorough ***knowledge of the underlying Internet protocols***.

Network Protocols and Layering

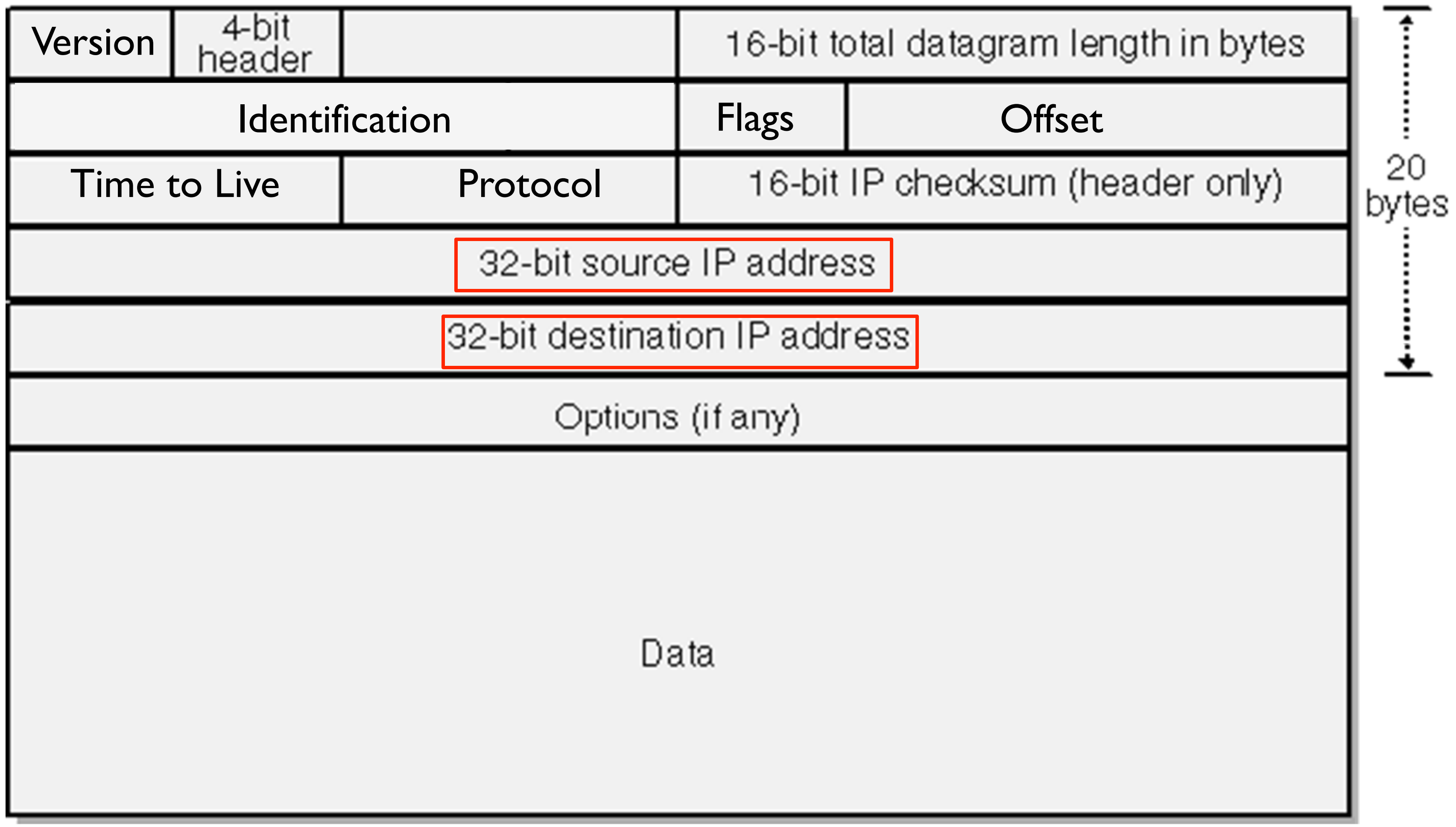


The stack for a LAN running TCP/IP.

Internet Protocol (RFC 791)

- Datagram (packet) protocol
- Best-effort service
  - Loss
  - Reordering
  - Duplication
  - Delay
- Host-to-host delivery
  (not application-to-application)

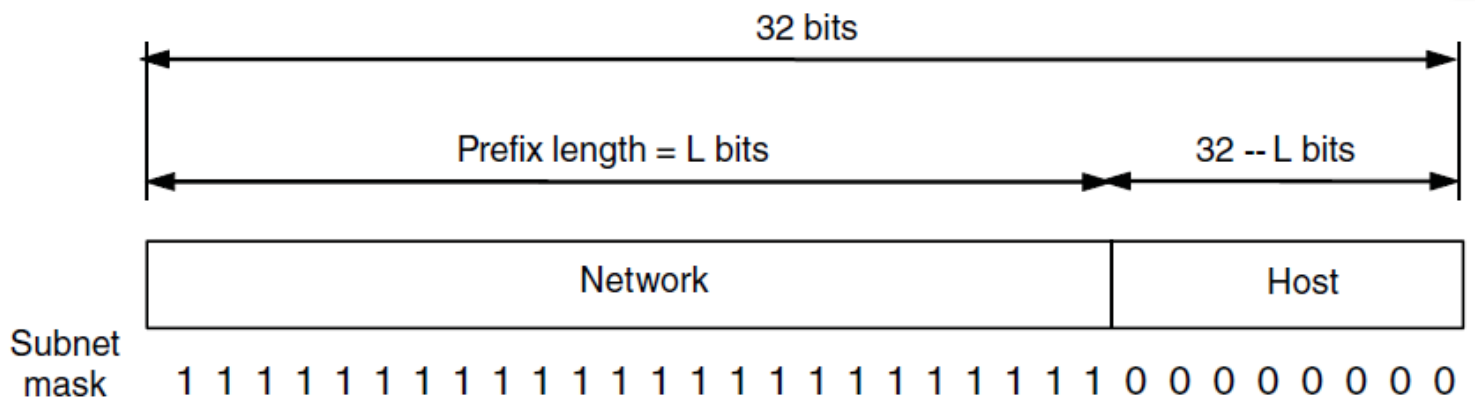
IP v4 packet format & IP Address



A simple IP datagram layout.

IP v4 Address Format

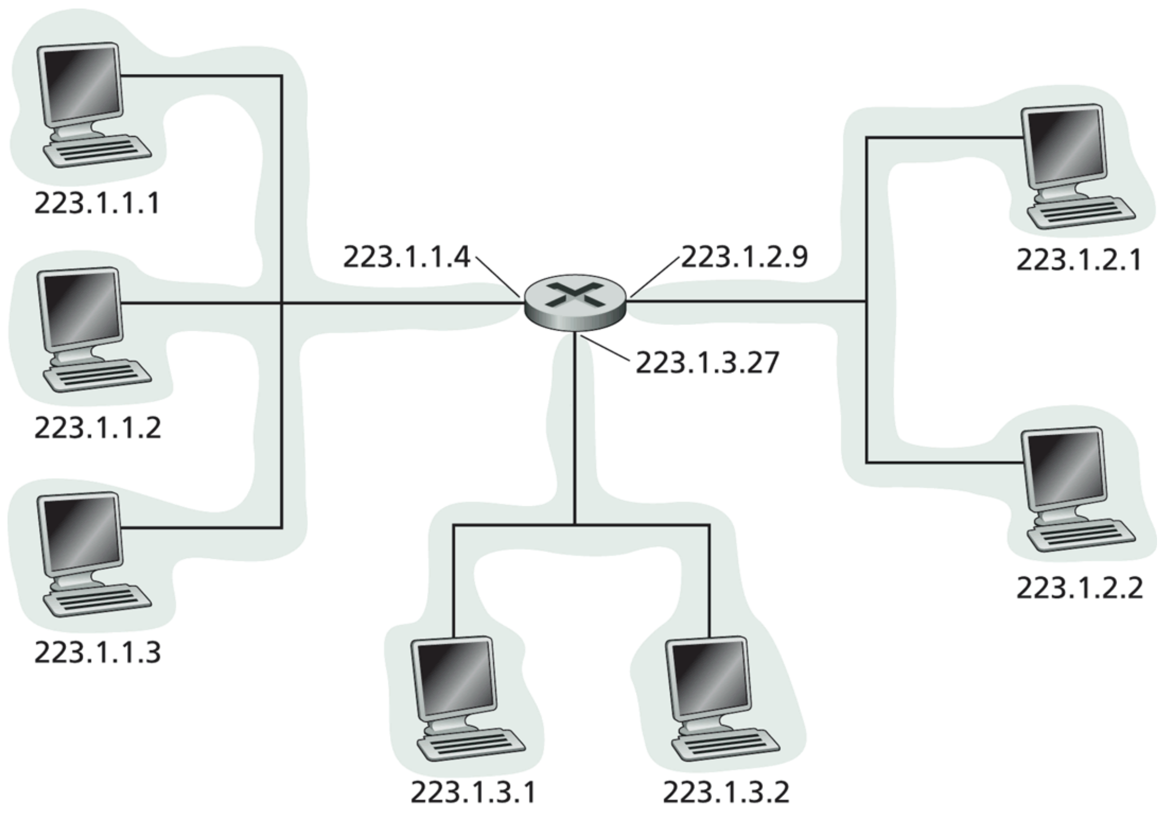
- 32-bit identifier to Identifies a host interface (not a host)
- IP addresses are written in dotted-decimal format



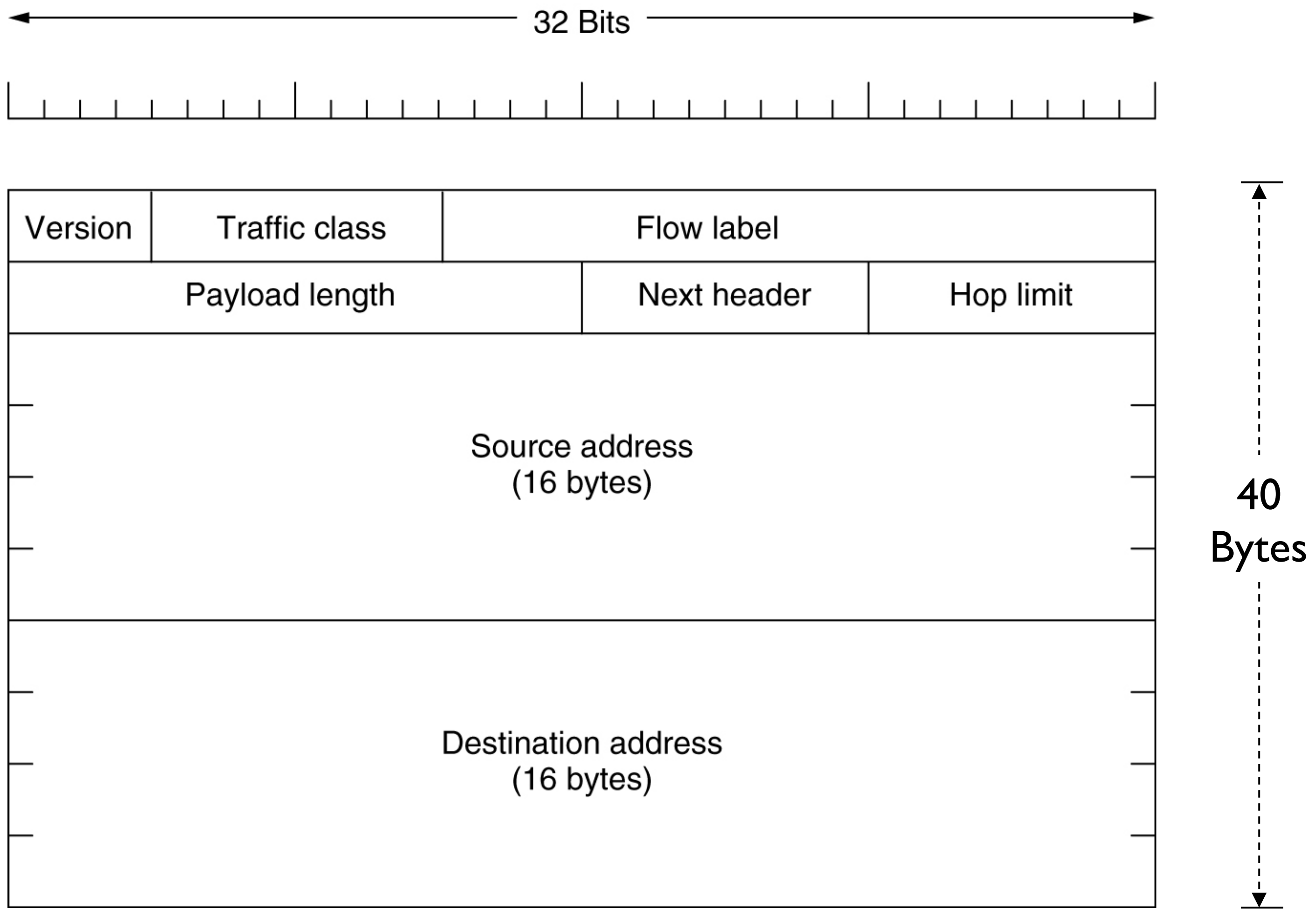
192.168.1.1/24

24 bits for the network portion, contains 2^8 addresses

Example IP networks



IP v6 (RFC 2460)



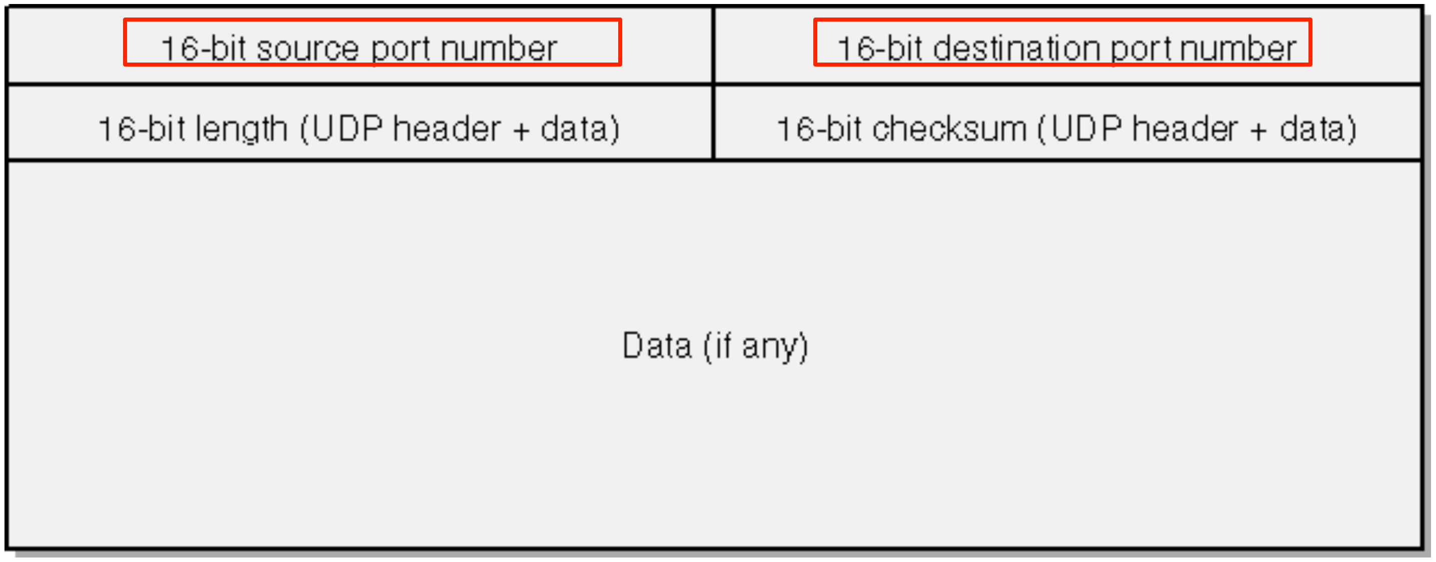
Basic IPv6 header

Transport Protocols

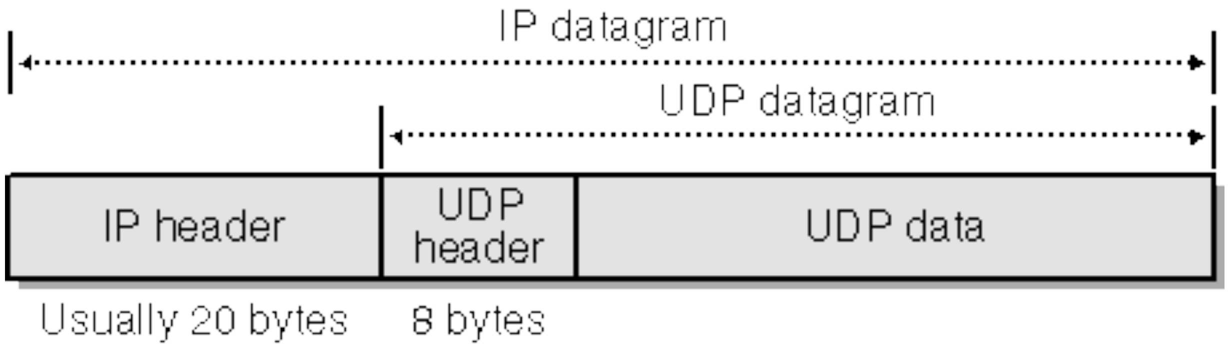
Why this layer: Best-effort is not sufficient !!!

- Add services on top of IP
- User Datagram Protocol (UDP)
  - Data checksum
  - Best-effort
- Transmission Control Protocol (TCP)
  - Data checksum
  - Reliable byte-stream delivery
  - Flow and congestion control

UDP segment format

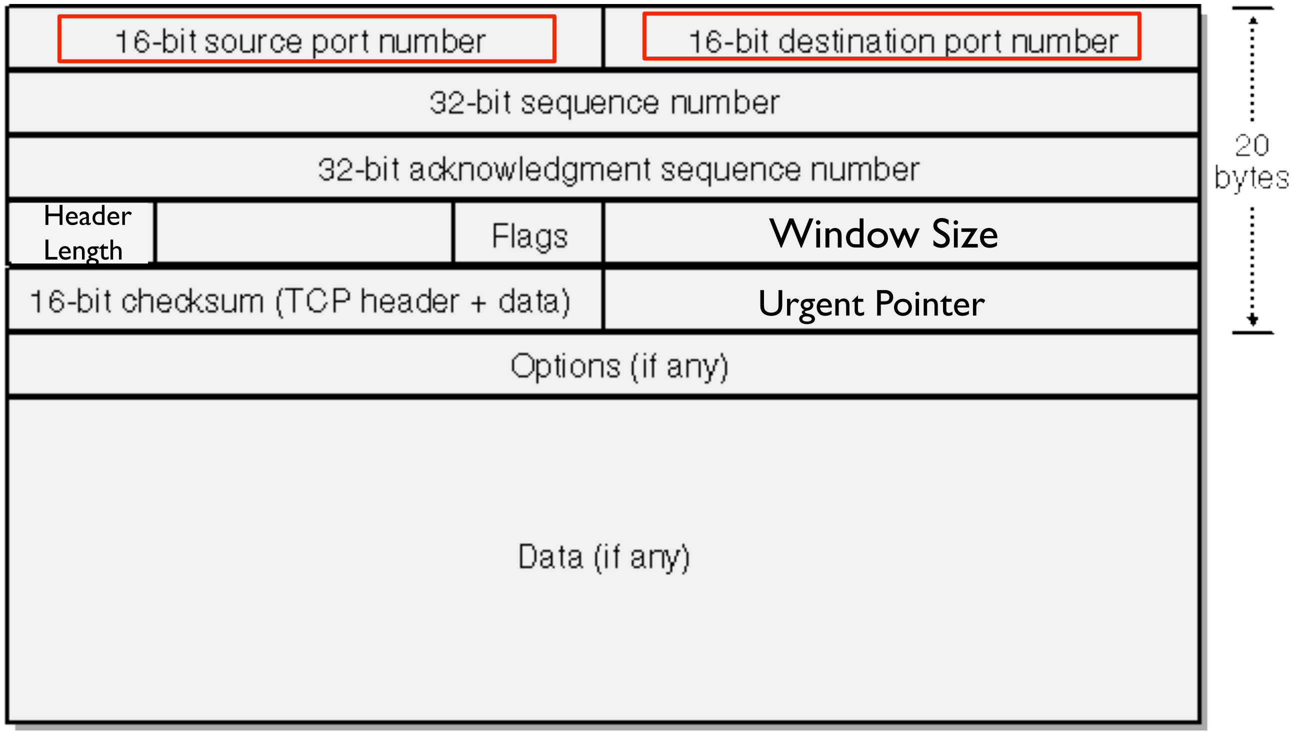

16-bit source port number
16-bit destination port number
16-bit length (UDP header + data)
16-bit checksum (UDP header + data)
Data (if any)

A simple UDP layout

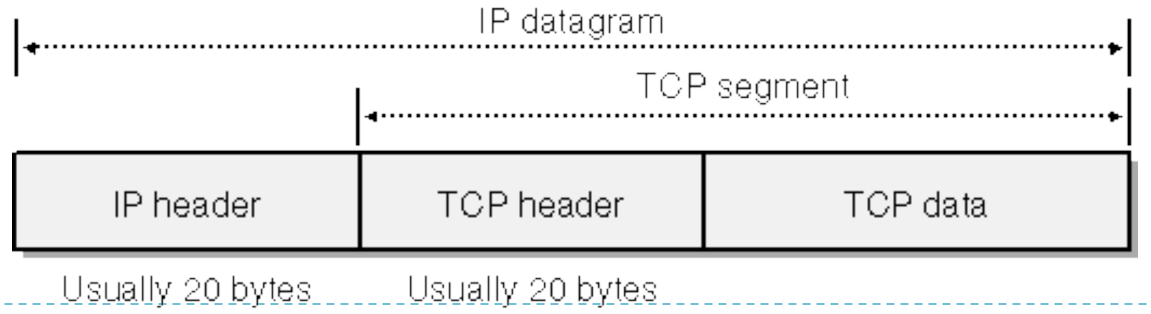


The relationship between the IP datagram and the UDP datagram.

TCP segment format


16-bit source port number
16-bit destination port number
32-bit sequence number
32-bit acknowledgment sequence number
Header Length
Flags
Window Size
16-bit checksum (TCP header + data)
Urgent Pointer
Options (if any)
Data (if any)
20 bytes

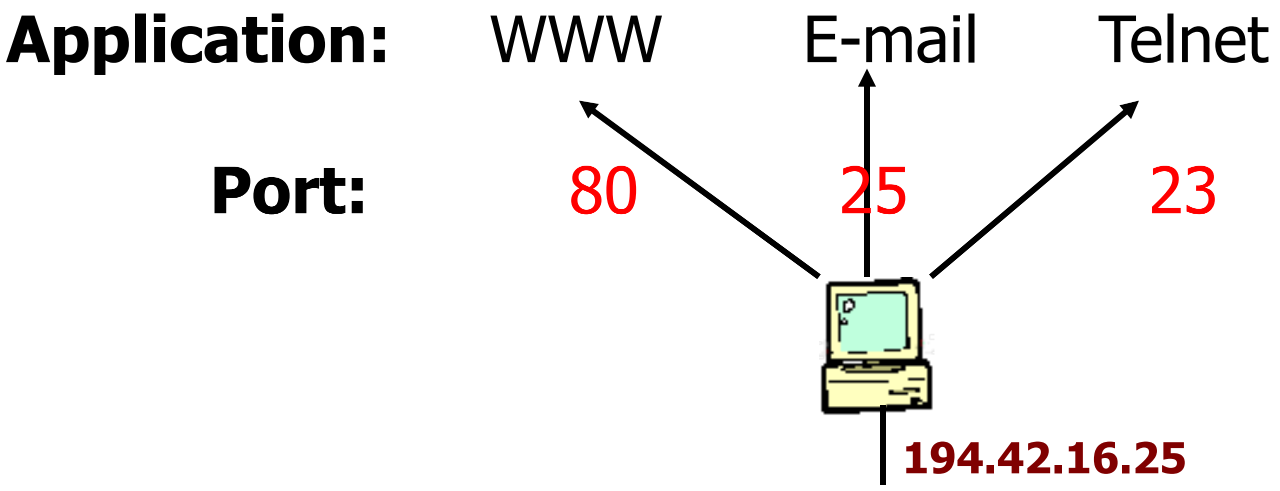
A simple layout of a TCP segment.



Ports

Identifying the ultimate destination

- IP addresses identify hosts
- Host has many applications
- Ports (16-bit identifier)



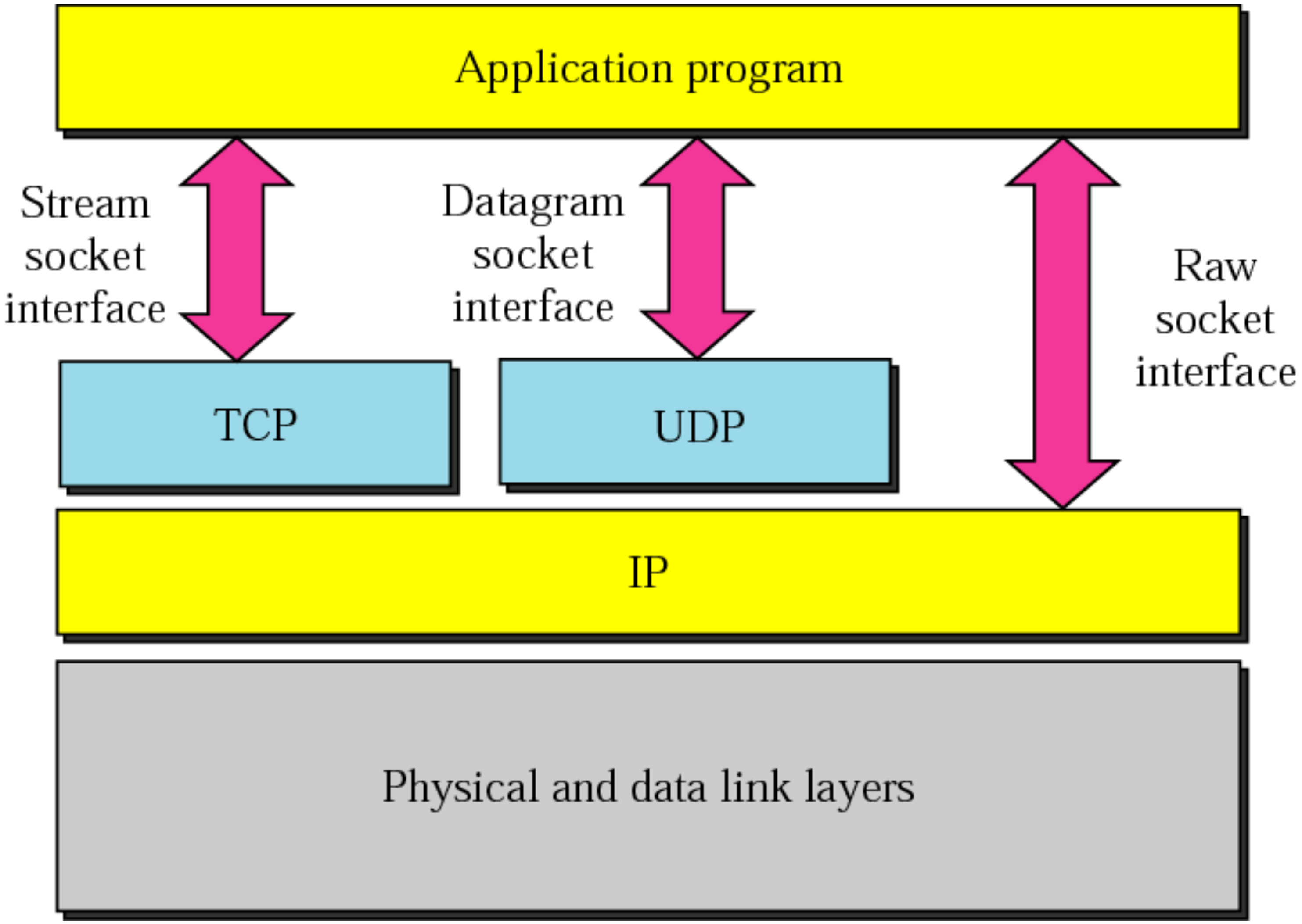
Network Byte Order

- Little endian order
  - In a computer with an Intel CPU, the address bytes are stored low-order-to-the-left
- *Big endian order*
  - In most other computers, including the UNIX machines that first supported the Internet, bytes are stored high-order-to-the-left
- Because the Internet imposes a machine-independent standard for data interchange, all multibyte numbers must be transmitted in *big endian* order.
  - This means that programs running on Intel-based machines must convert between *network byte order* (big endian) and *host byte order* (little endian).
  - This rule applies to 2-byte port numbers as well as to 4-byte IP addresses.

Socket

- The TCP protocol establishes a full-duplex, point-to-point connection between two computers, and a program at each end of this connection uses its own port.
- The combination of an IP address and a port number is called a ***socket***.
  - Establish connection : three-way handshake
  - Send/Receive data *stream:* ACK/Seq No./Piggybacking
  - Close connection: two way
- *Socket API*
  - Hide complexity of the TCP protocol
  - Your program calls a function to transmit a block of data, and Windows takes care of splitting the block into segments and stuffing them inside IP datagrams.

Sockets provides interface to TCP/IP



User Oriented view

- The bottom four layers can be seen as the *transport service provider*, whereas the upper layer(s) are the *transport service user.*
- Application programmers interact directly with the transport layer; from the programmer's perspective, *the transport layer is the `network'.* The transport layer should be oriented more towards user services.
- The network service is used only by the transport entities. Few users write their own transport entities, and thus few users or programs ever see the bare network service.

BSD Sockets

- Developed at UC Berkeley
  - Funded by ARPA in 1980.
  - Objective: to transport TCP/IP software to UNIX
  - The socket interface has become a de facto standard.
  - WinSock is based on the original BSD Sockets for UNIX
- A socket is one of the most fundamental technologies of computer networking.
  - What this means is a socket is used to allow one process to speak to another, very much like the telephone is used to allow one person to speak to another.
  - Many of today's most popular software packages -- including Web Browsers, Instant Messaging and File Sharing -- rely on sockets

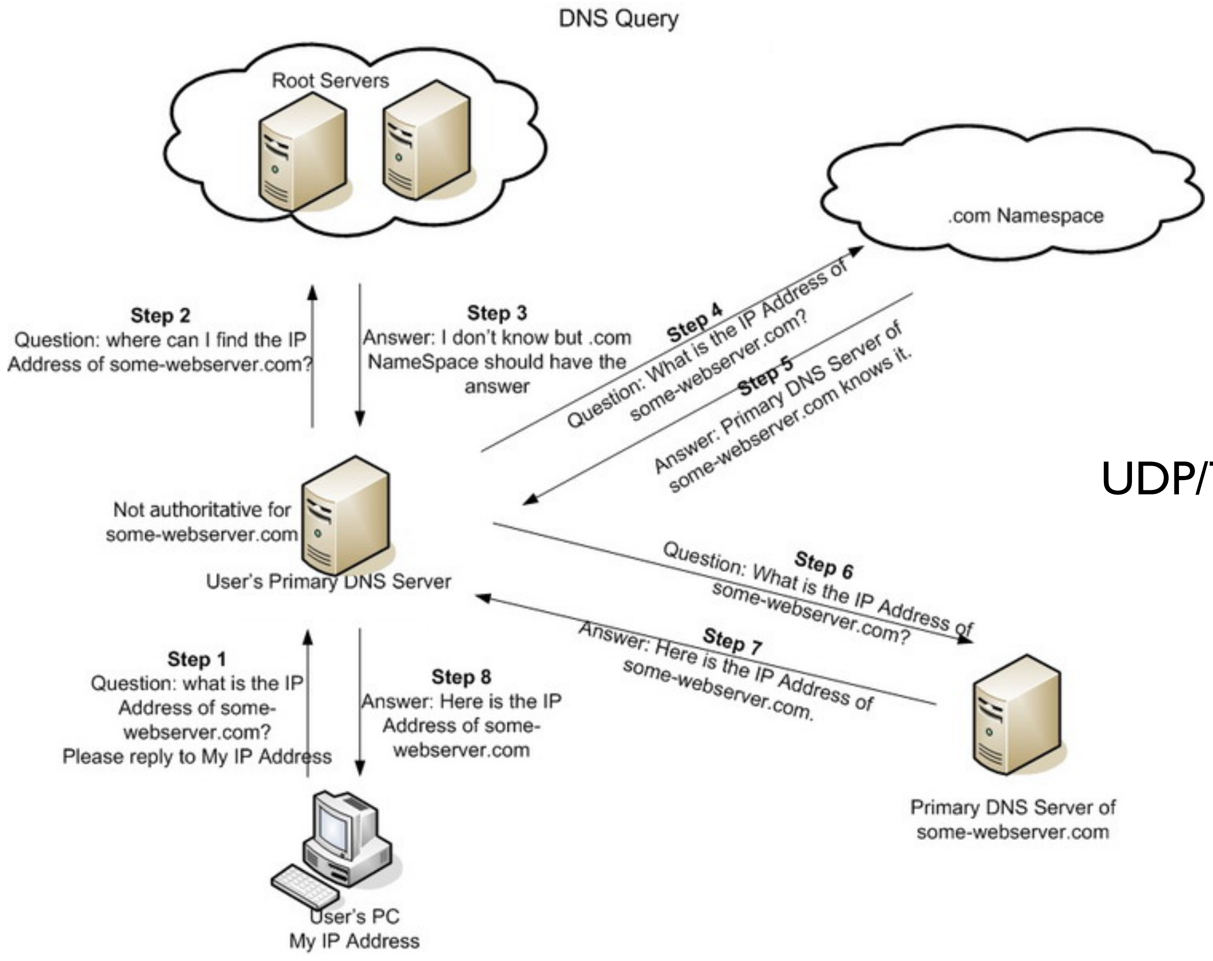
History of Sockets

- Sockets were introduced in 1981 as the Unix BSD 4.2 generic interface for Unix to Unix communications over networks.
- In 1985, SunOS introduced NFS and RPC over sockets.
- In 1986, AT&T introduced the Transport Layer Interface (TLI) with socket-like functionality but more network independent.
  - Unix after SVR4 includes both TLI and Sockets.
  - The two are very similar from a programmers perspective. TLI is just a cleaner version of sockets that is, in theory, stack independent.
  - TLI has about 25 API calls.

DNS (RFC 1034,1035)

- When we surf the Web, we don't use IP addresses. Instead, we use human-friendly names such as microsoft.com or www.cnn.com.
- A significant portion of Internet resources is consumed when *host names are translated into IP addresses* that TCP/IP can use.
- A distributed network of *name server* (domain server) computers performs this translation by processing *DNS queries*.
- The entire Internet namespace is organized into *domains*, starting with an unnamed *root domain*. Under the root is a series of *top-level domains* such as com, edu, gov and org.

DNS Query Process


DNS Query
Root Servers
.com Namespace
Step 2
Question: where can I find the IP Address of some-webserver.com?
Step 3
Answer: I don't know but .com NameSpace should have the answer
Step 4
Question: What is the IP Address of some-webserver.com?
Step 5
Answer: Primary DNS Server of some-webserver.com knows it.
Not authoritative for some-webserver.com
User's Primary DNS Server
Step 6
Question: What is the IP Address of some-webserver.com?
Step 7
Answer: Here is the IP Address of some-webserver.com.
Step 1
Question: what is the IP Address of some-webserver.com?
Please reply to My IP Address
Step 8
Answer: Here is the IP Address of some-webserver.com
Primary DNS Server of some-webserver.com
User's PC
My IP Address

UDP/TCP Port:53

HTTP (RFC 2616)

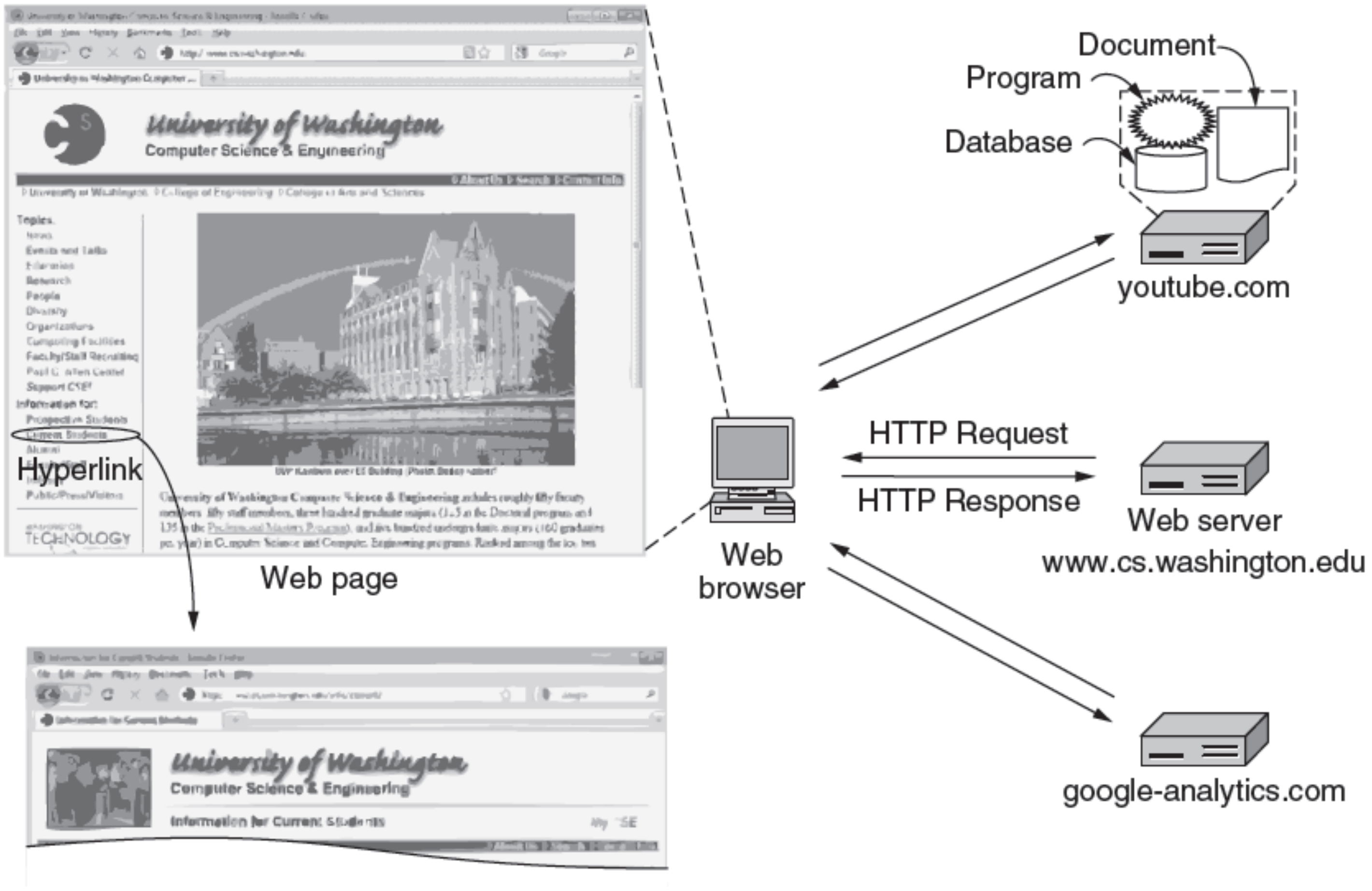
- HTTP is built on TCP, and this is the way it works:
  - First, a server program listens on port 80.
  - Then a client program (typically a browser) connects to the server after receiving the server's IP address from a name server.
  - Using its own port number, the client sets up a two-way TCP connection to the server.
  - When the connection is established, the client sends a request to the server, which might look like this (with optional header):

```
GET /customers/newproducts.html HTTP/1.1
```

  - The server sends back the html file following the next OK response (with header),

```
HTTP/1.1 200 OK
```

Architecture of the Web.



Introduction to Windows Socket

What is Winsock?

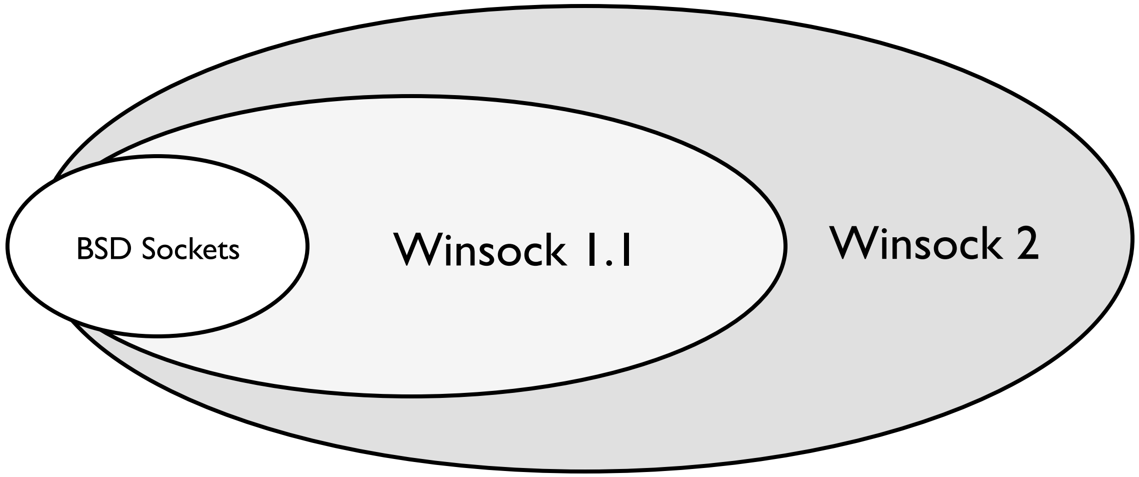
- The Windows Sockets (abbreviated "Winsock" or "WinSock") specification defines a network programming interface for Microsoft Windows which is based on the "socket" paradigm popularized in BSD Unix.
- It encompasses both familiar Berkeley socket style routines and a set of Windows-specific extensions.

Windows Socket overview

- WinSock version 1.1 is released in 1993
  - many TCP/IP vendors
  - limited the scope of the API specification to TCP/IP primarily
  - Windows 95 and Windows NT 3.x shipped with Winsock 1.1
- WinSock version 2.2, Aug. 1997
  - Microsoft's free TCP/IP implementations built into the operating systems
  - Support other protocol suites: ATM, IPX/SPX …
  - Adds substantial new functionality
  - Full backward compatibility with the existing 1.1
  - Windows 98 and all subsequent versions ship with Winsock 2

Relation to BSD Sockets

- Not a pure superset
- Winsock has many Windows-specific functions(those start with WSA prefix, e.g. WSASend())
- Some BSD Socket functions/options are not supported(e.g. sendv(), recev())



BSD-Style Winsock Functions

- Core functions
  - accept, bind, closesocket, connect, listen, recv, recvfrom, select, send, sendto, shutdown, socket
- Auxiliary functions
  - getpeername, getsockname, getsockopt, ioctlsocket
- Utility functions
  - htonl, htons, inet_addr, inet_ntoa, ntohl, ntohs
- These functions are compatible with the BSD version, and are provided to facilitate porting of Unix applications to the Windows platform.
- However a Unix sockets application does NOT compile nor behavior correctly under Windows without some required modifications.

Windows-Specific Winsock Functions

- Core functions
  - WSAStartup, WSACleanup, WSAAsyncGetHostByName/ Addr, WSAAsyncSelect, WSACancelAsyncRequest, WSAConnect, WSAIoctl, WSARecv, WSARecvFrom, WSASend, WSASendTo, WSASocket
- Auxiliary functions
  - WSADuplicateSocket, WSAEnumNetworkEvents, WSAEnumProtocols, WSAGetLastError, WSAGetQOSByName, WSAHtonl, WSAHtons, WSAJoinLeaf, WSANtohl, WSANtohs, WSAProviderConfigChange, WSASetLastError
- Supporting functions
  - WSACloseEvent, WSACreateEvent, WSAEventSelect, WSAGetOverlappedResult, WSAResetEvent, WSASetEvent, WSAWaitForMultipleEvents
- These functions are not portable to Unix platforms.

MFC Socket Classes in Visual C++

- CSocket, CAsyncSocket, etc.
- These are **not part of the Winsock specification.**
- They are just C++ classes wrapping the Winsock APIs.
- These are specific to Visual C++ and may *not be portable to other* platforms/compilers.

The MFC Winsock Classes
We've tried to use MFC classes where it makes sense to use them, but the MFC developers have informed us that the CAsyncSocket and CSocket classes are not appropriate for 32-bit synchronous programming. The Visual C++ .NET online help says you can use CSocket for synchronous programming, but if you look at the source code you'll see some ugly message-based code left over from Win16.

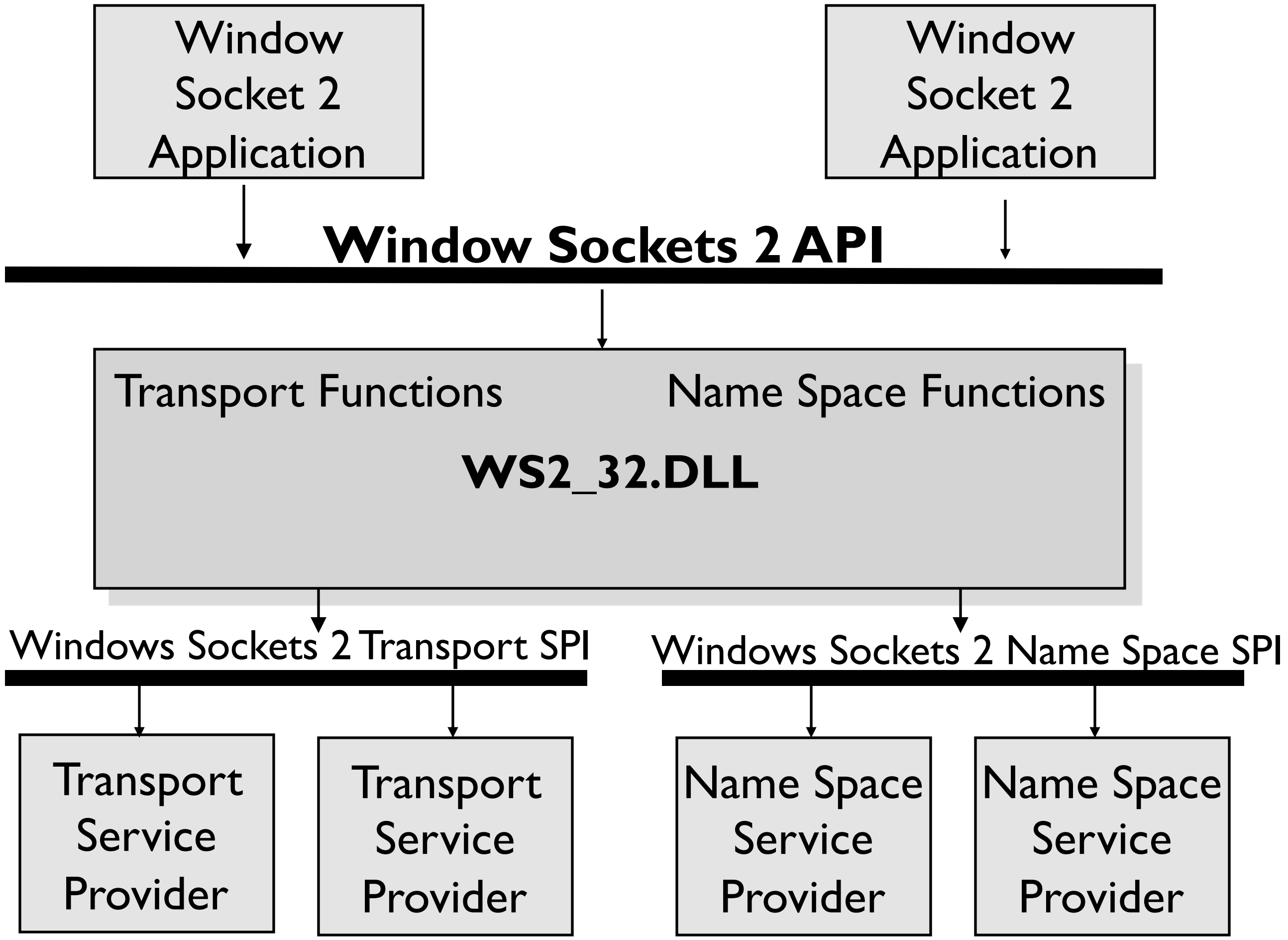
---Programming With Microsoft Visual C++ NET 6th Ed. - George/Kruglinski Shepherd. 2002

New features of WinSock 2

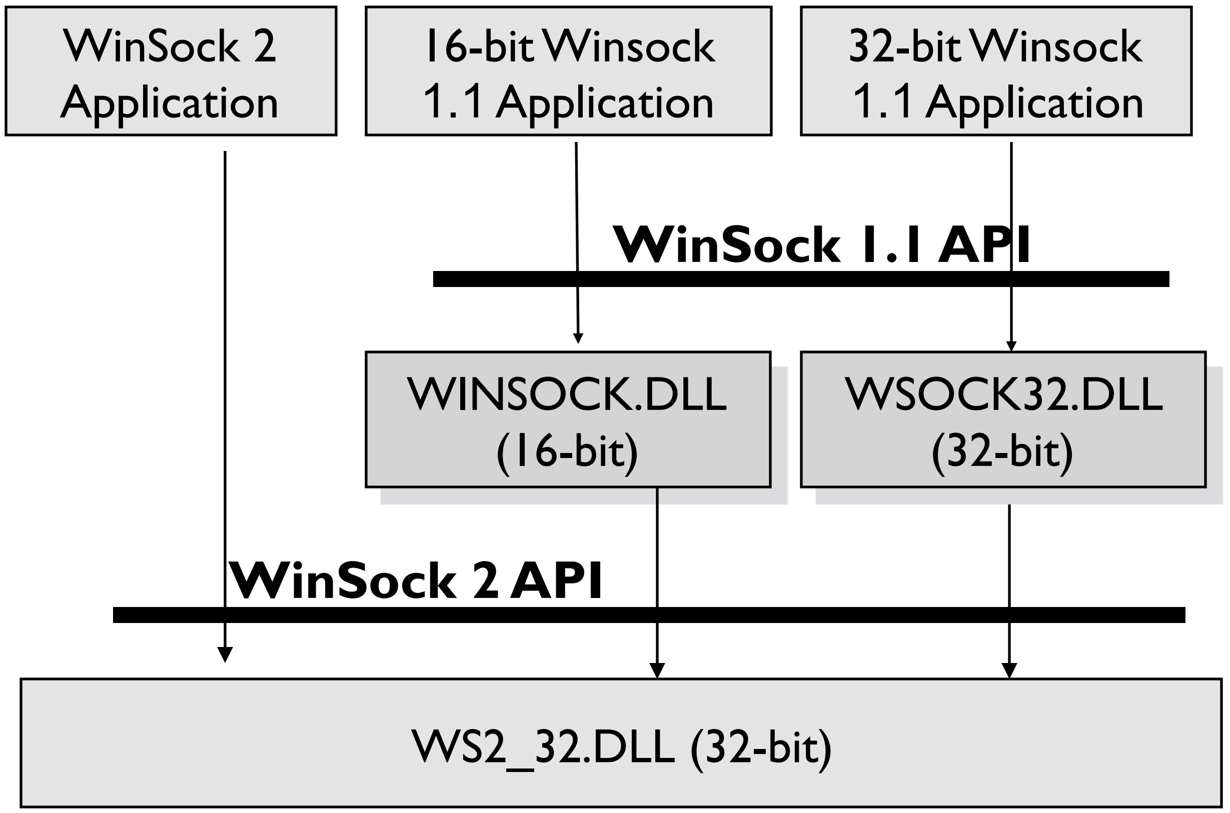
- Multiple Protocol support: WOSA architecture let's service providers "plug-in" and "pile-on"
- Transport Protocol Independence: Choose protocol by the services they provide
- Multiple Namespaces: Select the protocol you want to resolve hostnames, or locate services
- Scatter and Gather: Receive and send, to and from multiple buffers
- Overlapped I/O and Event Objects: Utilize Win32 paradigms for enhanced throughput
- Quality of Service: Negotiate and keep track of bandwidth per socket
- Multipoint and Multicast: Protocol independent APIs and protocol specific APIs
- Conditional Acceptance: Ability to reject or defer a connect request before it occurs
- Connect and Disconnect data: For transport protocols that support it (NOTE: TCP/IP does not)
- Socket Sharing: Two or more processes can share a socket handle
- Vendor IDs and a mechanism for vendor extensions: Vendor specific APIs can be added
- Layered Service Providers: The ability to add services to existing transport providers

WinSock 2 Architecture

- Windows Open Systems Architecture (WOSA) model


Window Socket 2 Application
Window Socket 2 Application
Window Sockets 2 API
Transport Functions
Name Space Functions
WS2_32.DLL
Windows Sockets 2 Transport SPI
Windows Sockets 2 Name Space SPI
Transport Service Provider
Transport Service Provider
Name Space Service Provider
Name Space Service Provider

Backward compatibility with Winsock1.1



* Figure taken from Winsock 2 specification document.

Basic Windows Socket Programming

Winsock Headers and Libraries

- Include files: winsock2.h
- Library files: ws2_32.lib

```
//winsock 2.2
#include <winsock2.h>
#pragma comment(lib, " Ws2_32.lib ")
```

```
//winsock 1.1
#include <winsock.h>
#pragma comment( lib, "wsock32.lib" )
```

Synchronous vs. Asynchronous Socket

- Synchronous
  - Blocking Socket
- Asynchronous
  - Non-Blocking Socket
- Multi-Threading
  - make Winsock calls from *worker threads* so the program's main thread can carry on with the user interface.

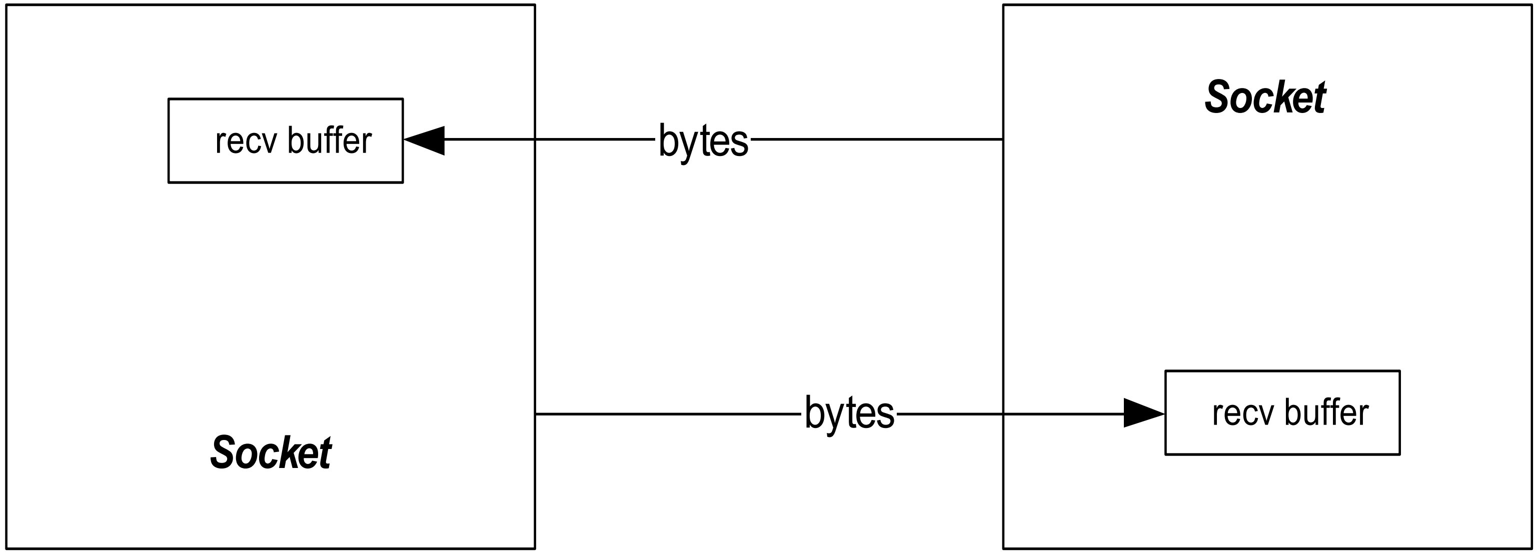
Blocking and Non-Blocking Socket

- 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.
- In non-blocking mode, the Winsock functions return immediately.

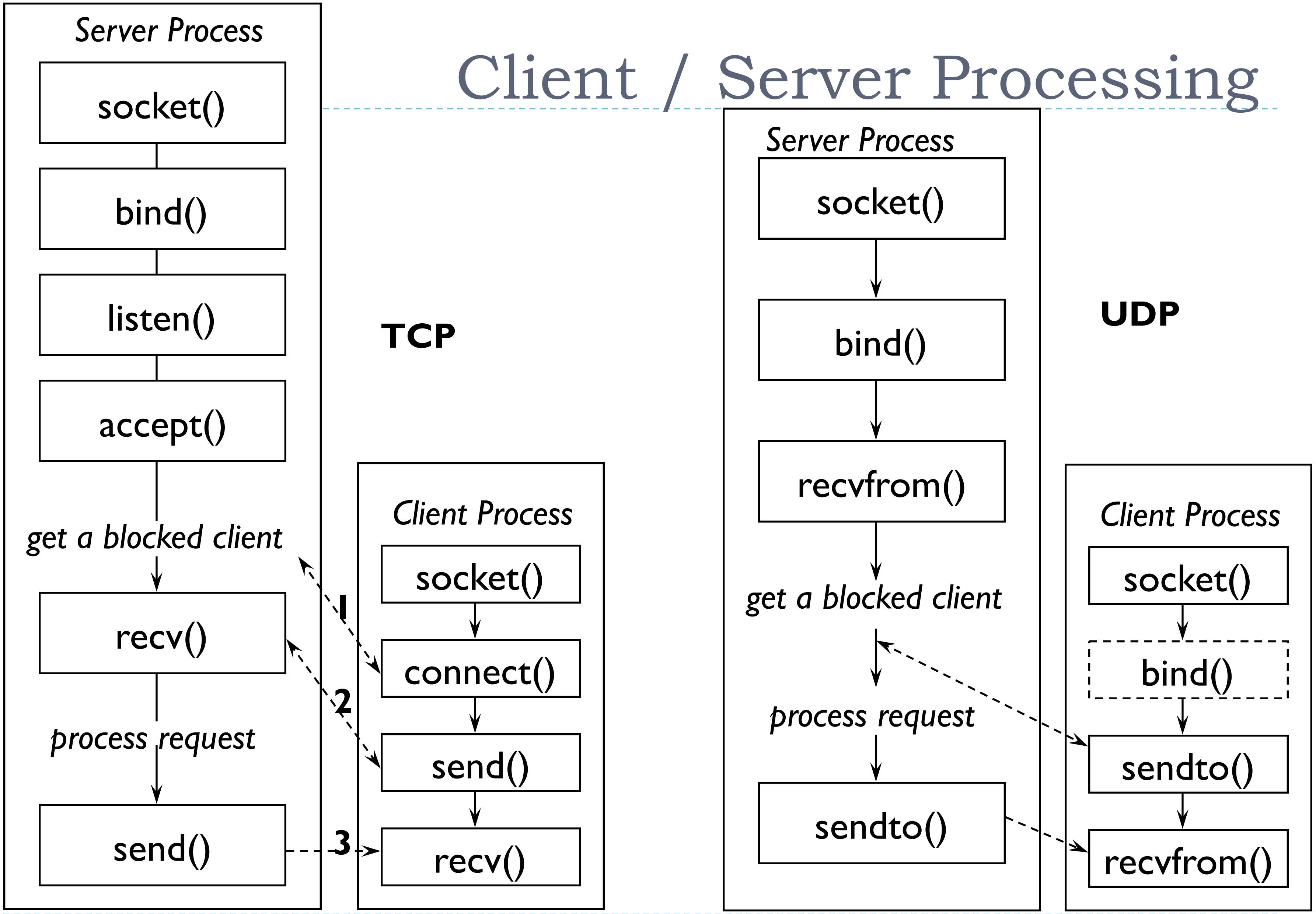
Connection-Oriented Communication

- Connection-oriented means that two communicating machines must first connect.
- In IP, connection-oriented communication is accomplished through the TCP/IP protocol. TCP provides reliable error-free data transmission between two computers.
- All data sent will be received in the same order as sent.
  - Note that IP packets may arrive in a different order than that sent.
  - This occurs because all packets in a communication do not necessarily travel the same route between sender and receiver.
- Streams mean that, as far as sockets are concerned, the only recognized structure is bytes of data.

Socket Logical Structure



Client / Server Processing



Socket API

WSAStartup	- loads WS2_32.dll
WSACleanup	- unloads dll
socket	- create socket object
connect	- connect client to server
bind	- bind server socket to address/port
listen	- request server to listen for connection requests
accept	- server accepts a client connection
send	- send data to remote socket
recv	- collect data from remote socket
Shutdown	- close connection
closesocket	- closes socket handle

Helper functions for byte ordering

- Byte Ordering: host byte --- network byte
  - u_long htonl(u_long hostlong);
  - u_short htons(u_short hostshort);

  - u_long ntohl(u_long netlong);
  - u_short ntohs(u_short netshort);

Convert IP address: Windows

- IPv4 only
  - Convert a string containing an IPv4 dotted-decimal address into/from a proper address for the **IN_ADDR** structure.

```
unsigned long inet_addr( _In_  const char *cp );
char* FAR inet_ntoa( _In_  struct in_addr in );
```

- Converts an IPv4 or IPv6 Internet network address into/ from a string in Internet standard format.

```
PCTSTR WSAAPI InetNtop(     _In_   INT Family,
                            _In_   PVOID pAddr,
                            _Out_  PTSTR pStringBuf,
                            _In_   size_t StringBufSize );

INT WSAAPI InetPton(        _In_   INT Family,
                            _In_   PCTSTR pszAddrString,
                            _Out_  PVOID pAddrBuf );
```

Convert IP address: Linux

- Convert IPv4 or IPv6 addresses to human-readable form and back.

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
int inet_pton(int af, const char *src, void *dst);
```

- demo

```
// IPv4 demo of inet_ntop() and inet_pton()
struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));

// now get it back and print it
inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN);

printf("%s\n", str); // prints "192.0.2.33"
```

Sequence of Server Calls

- WSAStartup
- socket (create listener socket)
- bind
- listen
- accept
  - create new socket so listener can continue listening
  - create new thread for socket
  - send and recv
  - closesocket (on new socket)
  - terminate thread
- shutdown
- closesocket (on listener socket)
- WSACleanup

Initializing Winsock: WSAStartup()

- WSAStartup()

```
wVersionRequested = MAKEWORD(2,2);
WSADATA wsaData;
// Initialize Winsock version 2.2
if ((WSAStartup(wVersionRequested, &wsaData)!= 0)
 {
            return;
 }
```

```
//Initializing Winsock
int WSAStartup(
   WORD wVersionRequested,
   LPWSADATA lpWSAData
);

//Clean up Winsock
int WSACleanup(void);
```

```
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, * LPWSADATA;
```

Prototypes for end address:IPv4

- 16-byte *sockaddr_in* structure

```
struct sockaddr_in {
    short   sin_family;              //type of address (always AF_INET)
    u_short sin_port;                //protocol port number
    struct  in_addr sin_addr;    //IP address, it is stored in a structure instead of an
                            //unsigned long (with 32 bits) because of a historical reason
    char    sin_zero[8];           //unused, set to zero
}
```

- 4-byte *in_addr* structure

```
// Internet address (a structure for historical reasons)
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

- Setting sin_addr.s_addr = INADDR_ANY allows a server application to listen for client activity on every network interface on a host computer.

examples

```
// IPv4:
struct sockaddr_in ip4addr;
int s;

ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(3490);
ip4Addr.sin_addr.s_addr = inet_addr("10.0.0.1");
//InetPton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof(ip4addr));
```

Prototypes for end address:IPv6

- IPv6 AF_INET6 sockets

```
struct sockaddr_in6 {
        short sin6_family;                    /* AF_INET6 */
        u_short sin6_port;                    /* Transport level port number */
        u_long sin6_flowinfo;                 /* IPv6 flow information */
        struct in_addr6 sin6_addr;            /* IPv6 address */
        u_long sin6_scope_id;                 /* set of interfaces for a scope */
};


struct in_addr6 {
        u_char s6_addr[16];                   /* IPv6 address */
};
```

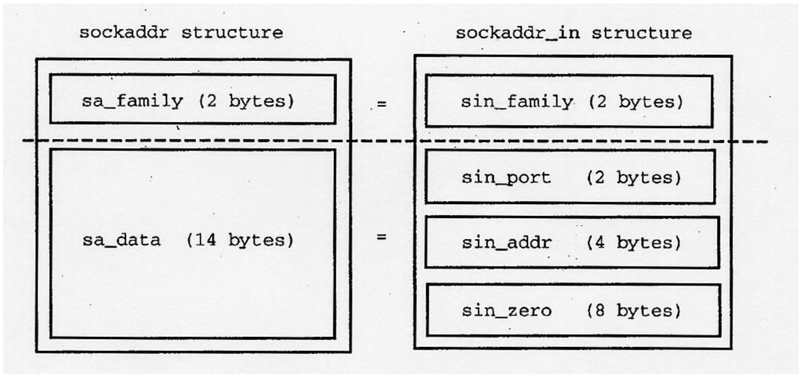
examples

```
// IPv6:
struct sockaddr_in6 ip6addr;
int s;

ip6addr.sin6_family = AF_INET6;
ip6addr.sin6_port = htons(4950);
InetPton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);

s = socket(PF_INET6, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip6addr, sizeof(ip6addr));
```

Generic Endpoint Address

- The socket abstraction accommodates many protocol families.
  - It supports many address families(AF_INET (IPv4) or AF_INET6 (IPv6)).
  - It defines the following generic endpoint address:
    (address family, endpoint address in that family)
- Data type for generic endpoint address:

```
//cast to switch between the two types
struct sockaddr {
   unsigned short sa_family;    //type of address
   char sa_data[14];            //value of address
};
```



- *sockaddr* and *sockaddr_in* are compatible

Examples

- The IP address of a server is 136.149.3.29.

```
struct sockaddr_in ServerAddr;
              …
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_addr.s_addr =  inet_addr("136.149.3.29");
ServerAddr.sin_port = htons(2000);
//convert to string
char *strAddr = inet_ntoa(ServerAddr.sin_addr);
```

```
struct sockaddr_in ServerAddr;
              …
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);
ServerAddr.sin_port = htons(2000);
```

where the symbolic constant **INADDR_ANY** represents a wildcard address that matches any of the computer's IP address(es).

Creating a Socket

- A socket is a handle to a transport provider.

```
SOCKET socket (
    int af,                 //address family
    int type,               // socket type
    int protocol            // default 0
);
```

- Address Family: AF_INET for IPv4 protocol ; AF_INET6 for IPv6 protocol
- Type – stream, datagram, raw IP packets, …
  - SOCK_STREAM → TCP packets
  - SOCK_DGRAM → UDP packets
  - SOCK_RAW → RAW packets

```
SOCKET m_hSocket;
m_hSocket = socket(AF_INET,SOCK_STREAM,0);
if (m_hSocket == INVALID_SOCKET)  {
      printf( "Socket failed with error %d\n",WSAGetLastError());
}
```

Error Checking and Handling

- The most common return value for an unsuccessful Winsock call is *SOCKET_ERROR*

```
#define SOCKET_ERROR        -1
```

- If you make a call to a Winsock function and an error condition occurs, you can use the function WSAGetLastError to obtain a code that indicates specifically what happened. This function is defined as

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

```
if (WSACleanup() == SOCKET_ERROR) {
		printf("WSACleanup failed with error %d\n",WSAGetLastError());
}
```

Bind socket

- Assign an Endpoint Address to a Socket
  - After creating a socket, a server must assign its endpoint address to this socket.
    - Then the client can identify the server's socket and send requests to this server.
- The server calls *bind()* to bind its endpoint address to the newly created socket.

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

```
struct sockaddr_in ServerAddr;
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);
ServerAddr.sin_port = htons(2000);
bind (m_hSocket ,(struct sockaddr *) &ServerAddr, sizeof(ServerAddr));
```

Listen to incoming requests

- Server Listens to Connection Requests
  - After creating a socket, the server calls *listen()* to place this socket in passive mode.
    - Then the socket listens and accepts incoming connection requests from the clients.
  - Multiple clients may send requests to a server.
    - The function *listen()* tells the OS to queue the connection requests for the server's socket.
  - The function *listen()* has two arguments:
    - the server's socket;
    - the maximum size of the queue.
  - The function *listen()* applies only to sockets used with TCP.

Function listen( )

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

- Backlog* is the number of incoming connections queued (pending) for acceptance
- Puts socket in listening mode, waiting for requests for service from remote clients.

```
listen(m_hSocket, 5);
m_hListenSocket = m_hSocket;
```

- Ch.18.11 TCP Server Design, TCP/IP illustrated, volume 1
- Ch. 4.5 listen Function. Unix Network Programming Volume 1
- 4.14 - What is the connection backlog? Winsock Programmer's FAQ

Accept Incoming Connection

- Server Accepts a Connection Request
  - The server calls *accept()* to
    - extract the next incoming connection request from the queue.
    - Then the server creates a new socket for this connection request and returns the descriptor of this new socket.
  - Remarks
    - The server uses the new socket for the new connection only. When this connection ends, the server closes this socket.
    - The server still uses the original socket to accept additional connection requests.
    - The function *accept()* applies only to stream (TCP) sockets.

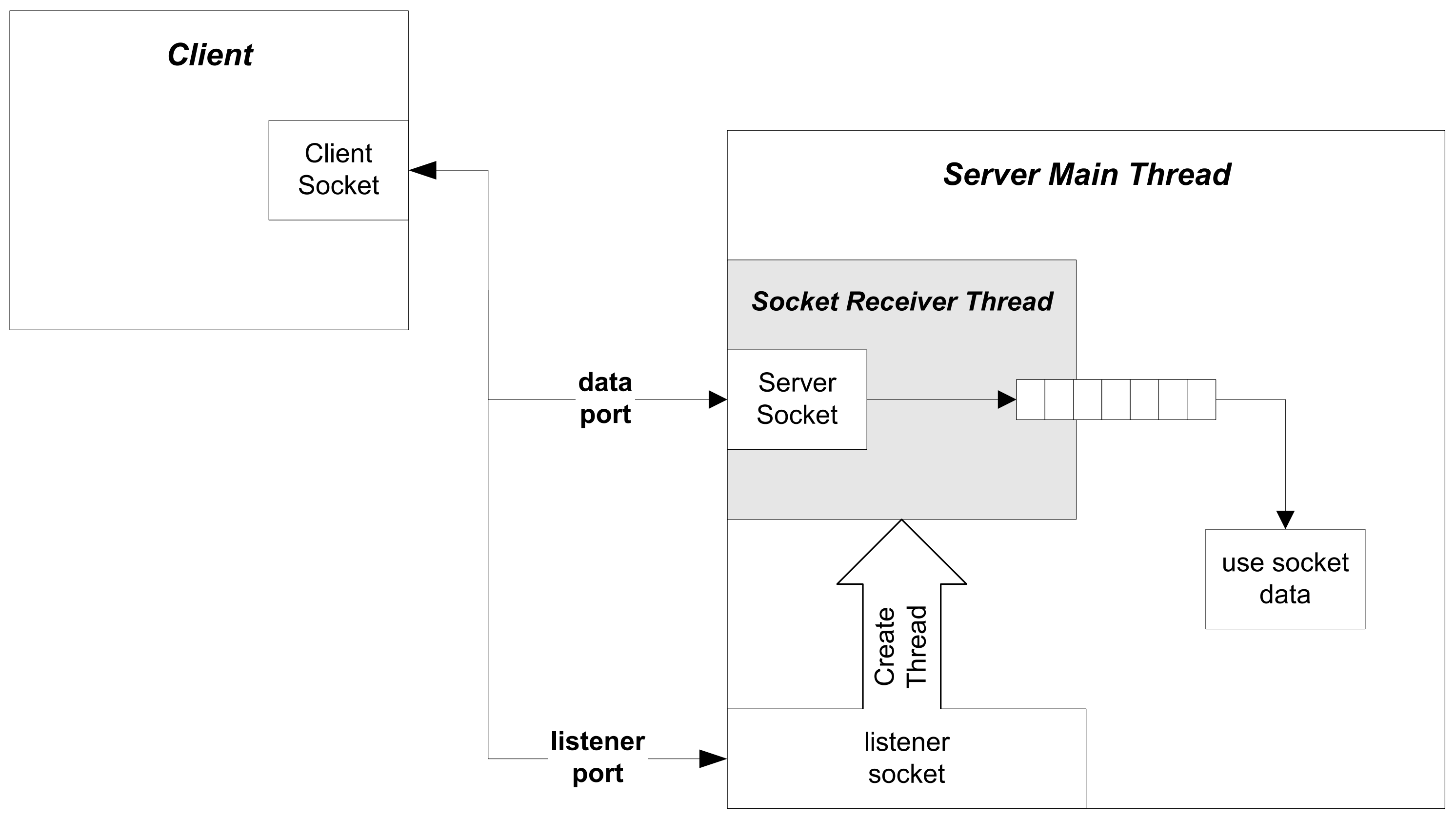
Accept( )

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

- Parameters
  - *S[in]*: A 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] :An optional pointer to a buffer that receives the address of the connecting entity, as known to the communications layer.
  - *addrlen [in, out]* :An optional pointer to an integer that contains the length of structure pointed to by the *addr* parameter.
- Return Value
  - **accept** returns a value of type **SOCKET** that is a descriptor for the new socket. This returned value is a handle for the socket on which the actual connection is made.

```
struct sockaddr_in ClientAddr; Size=sizeof(ClientAddr);
m_hSocket = accept(m_hListenSocket, (struct sockaddr *) &ClientAddr, &Size);
```

Client/Server Configuration



Receiving data

- Both the client and server call *recv()* to receive data through a TCP connection.
- Before calling *recv()*, the application must allocate a buffer for storing the data to be received.
- The function *recv()* extracts the data that has arrived at the specified socket, and copies them to the application's buffer. Two possible cases:
- <u>Case 1: No data has arrived</u>
  - The call to *recv()* blocks until data arrives.
- <u>Case 2: Data has arrived</u>
  - If the incoming data can fit into the application's buffer, *recv()* extracts all the data and returns the number of bytes received.
  - Otherwise, *recv()* only extracts enough to fill the buffer. Then it is necessary to call *recv()* a number of times in order to receive the entire message.

Recv()

```
int recv(
  SOCKET s,
  char *buff,
  int len,    //Receive data in buff up to len bytes.
  int flags   //flags should normally be zero.
) //Returns actual number of bytes read.
```

```
//Example:  receive a small message with at most 5 characters
char  buf[5],  *bptr;
int   buflen;
…
bptr = buf;
buflen = 5;
recv(s, bptr, buflen, 0);
```

If no incoming data is available at the socket, the recv call **blocks** and waits for data to arrive according to the blocking rules defined for WSARecv with the MSG_PARTIAL flag not set unless the socket is nonblocking. In this case, a value of SOCKET_ERROR is returned with the error code set to WSAEWOULDBLOCK. The select, WSAAsyncSelect, or WSAEventSelect functions can be used to determine when more data arrives.
If the socket is connection oriented and the remote side has shut down the connection gracefully, and all data has been received, a recv will complete immediately with zero bytes received. If the connection has been reset, a recv will fail with the error WSAECONNRESET.

Sending Data

- Both client and server calls *send()* to send data across a connection.
- The function *send()* copies the outgoing data into buffers in the OS kernel, and allows the application to continue execution while the data is being sent across the network.
- If the buffers become full, the call to *send()* may block temporarily until free buffer space is available for the new outgoing data.
- Send() return number of characters sent if successful.

Send( )

```
int send(
   SOCKET s,
   char *buff,
   int len,    //Send data in buff up to len bytes.
   int flags   //should normally be zero.
)//Returns actual number of bytes sent.
```

```
char   *message="Hello world!";
...
If (send ( m_hSocket, message, strlen(message), 0) < 0)
{
         printf("SOCKET END ERROR: %d\n", GetLastError());
}
```

If no buffer space is available within the transport system to hold the data to be transmitted, **send** will block unless the socket has been placed in nonblocking mode. On nonblocking stream oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both the client and server computers. The **select**, **WSAAsyncSelect** or **WSAEventSelect** functions can be used to determine when it is possible to send more data.

shutdown

```
int shutdown(SOCKET s, int how)
```

- `how = SD_SEND or SD_RECEIVE or SD_BOTH`

- Disables new sends, receives, or both, respectively. Sends a FIN to server causing thread for this client to terminate (server will continue to listen for new clients).

Close a Socket

- Once a client or server finishes using a socket, it calls *closesocket()* to
  - terminate the TCP connection associated with this socket, and
  - deallocate this socket.

Close a socket

When an application finishes using sockets, it must call *WSACleanup()* to deallocate all data structures and socket bindings.

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

- **Closes socket handle s, returning heap allocation for that data structure back to system.**
- **Note: To assure that all data is sent and received on a connection, an application should call shutdown before calling closesocket**

WSACleanup

When an application finishes using sockets, it must call *WSACleanup()* to deallocate all data structures and socket bindings.

```
int WSACleanup( )
```

- Unloads `W2_32.dll` if no other users. Must call this once for each call to `WSAStartup`.

Sequence of Client Calls

- WSAStartup
- socket
- address resolution - set address and port of intended receiver
- connect - send and recv
- shutdown
- closesocket
- WSACleanup

Name Resolution- HOSTENT

- The hostent structure is used by functions to store information about a given host, such as host name, IPv4 address, and so forth.

```
typedef struct hostent {
  char FAR *    h_name;   //host name
  char FAR  FAR **h_aliases;  // a list of alternative names
  short         h_addrtype;       //type of address
  short         h_length;         //The length of each address
  char FAR  FAR **h_addr_list; // A NULL-terminated list of addresses for the host.
                                  //Addresses are returned in network byte order.
}HOSTENT, *PHOSTENT, FAR *LPHOSTENT;
```

- An application should never attempt to modify this structure or to free any of its components.
- Furthermore, only one copy of the hostent structure is allocated per thread, and an application should therefore copy any information that it needs before issuing any other Windows Sockets API calls.

Name Resolution- gethostbyname/Addr

- gethostbyname retrieves host information corresponding to a host name

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

- **gethostbyaddr** retrieves the host information corresponding to a network address.

```
struct hostent* FAR gethostbyaddr(
            const char *addr,
            int len,
            int type
);
```

Note The two functions has been deprecated by the introduction of the getaddrinfo function. Developers creating Windows Sockets 2 applications are urged to use the getaddrinfo function.

example

```
struct hostent *remoteHost;
char *host_name;
//… fill host_name
struct in_addr addr;

 if (isalpha(host_name[0])) {        /* host address is a name */
        remoteHost = gethostbyname(host_name);
  } else {
     addr.s_addr = inet_addr(host_name);
     if (addr.s_addr == INADDR_NONE) {
            printf("The IPv4 address entered must be a legal address\n");
            return 1;
        } else
        remoteHost = gethostbyaddr((char *) &addr, 4, AF_INET);
     }
int i = 0;
 while (remoteHost->h_addr_list[i] != 0) {
            addr.s_addr = *(u_long *) remoteHost->h_addr_list[i++];
            printf("\tIP Address #%d: %s\n", i, inet_ntoa(addr));
}
```

Service Resolution- SERVENT

- The **servent** structure is used to store or return the name and service number for a given service name.

```
typedef struct servent {
  char FAR *      s_name;    //official name of the service
  char FAR  FAR **s_aliases;//array of alternate names.
  short           s_port;  //Port numbers are returned in network byte
order.
  char FAR *      s_proto;  //name of the protocol to use
}SERVENT, *PSERVENT, FAR *LPSERVENT;
```

Service Resolution

- **getservbyname()** retrieves service information corresponding to a service name and protocol.

```
struct servent* FAR getservbyname(
            const char *name,  //service name.
            const char *proto   // Optional pointer to a protocol name
);
```

- **getservbyport()** retrieves service information corresponding to a port and protocol.

```
struct servent* FAR getservbyport(
            int port,   //Port for a service, in network byte order.
            const char *proto   //Optional pointer to a protocol name.
);
```

If the optional pointer is null, **getservbyport/getservbyname** returns the first service entry for which the name or *port* matches the **s_name/s_port** of the **servent** structure. Otherwise, **getservbyport/getservbynam** matches both the *port* and the *proto* parameters.

example

```
char        *port="http";
struct      servent     *se;
se = getservbyname(port, NULL);

se >s_port has the port number.
```

```
struct servent * se;
..
se = getservbyname("smtp","tcp");

se >s_port has the port number.
```

Connecting

- Client Initiates a TCP Connection
  - After creating a socket, a client calls *connect()* to establish a TCP connection to a server.
  - The function *connect()* :

```
int connect(
   SOCKET s,                //the descriptor of the client's socket;
   const struct sockaddr *addr,  //endpoint address of the server;
   int addrlen              //length of the 2nd argument.
)
```

```
hostent * remoteHost = gethostbyname(host_name);
struct sockaddr_in ServerAddr;
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_addr = *(struct in_addr *) remoteHost ->h_addr_list[0];
ServerAddr.sin_port = htons(2000);
connect ( m_hSocket, (struct sockaddr *)&ServerAddr, sizeof(ServerAddr) );
```

Example

```
// Create a connection-oriented socket
SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
// Check to see if we have a valid socket
if(s == INVALID_SOCKET) {
   int iSocketError = WSAGetLastError();   return FALSE;
}
//Get the host information
HOSTENT *hostServer = gethostbyname("www.microsoft.com");
if(hostServer == NULL) {
   int iSocketError = WSAGetLastError();
   return FALSE;
}
// Set up the target device address structure
SOCKADDR_IN sinServer;
memset(&sinServer, 0, sizeof(SOCKADDR_IN));
sinServer.sin_family = AF_INET;
sinServer.sin_port = htons(80);
sinServer.sin_addr =   *((IN_ADDR *)hostServer>h_addr_list[0]);

// Connect with a valid socket
if(connect(s, (SOCKADDR *)&sinServer, sizeof(sinServer)) ==
   SOCKET_ERROR) {
   int iSocketError = WSAGetLastError();
   return FALSE;
}
// Do something with the socket
closesocket(s);
```

TCP States

- As a Winsock programmer, you are not required to know the actual TCP states, but by knowing them you will gain a better understanding of how the Winsock API calls affect change in the underlying protocol.
- In addition, many programmers run into a common problem when closing sockets:
  - the TCP states surrounding a socket closure are of the most interest.

UDP Server/Client

- sendto() and recvfrom()
- socket()
- bind()
- connect()/close()

sendto() and recvfrom() for DGRAM

- **sendto()** returns the number of bytes actually sent

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

- **recvfrom()** returns the number of bytes received

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

Note: to/from is a pointer to a remote/local struct sockaddr_storage that will be filled with the IP address and port of the originating machine. Tolen/fromlen is a pointer to a remote/local int that should be initialized to sizeof *to/from or sizeof(struct sockaddr_storage). When the function returns, tolen/fromlen will contain the length of the address actually stored in from.

Server: Create socket and bind

```
SOCKET m_hSocket;
if((m_hSocket = socket(AF_INET, SOCK_DGRAM, 0)) < 0) { // create the socket
   printf( "Socket failed with error %d\n",WSAGetLastError());
}
struct sockaddr_in srv;
srv.sin_family = AF_INET;
srv.sin_port = htons(80); /* bind: socket 'fd' to port 80*/
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if ( bind(m_hSocket, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
    printf( "Bind failed with error %d\n",WSAGetLastError());
}
```

Server: Receiving UDP Datagrams

```
struct sockaddr_in cli;                    /* used by recvfrom() */
char buf[512];                             /* used by recvfrom() */
int cli_len = sizeof(cli);                 /* used by recvfrom() */
int nbytes;                                /* used by recvfrom() */

// 1) create the socket m_hSocket
// 2) bind to the socket , struct sockaddr_in srv

nbytes = recvfrom(m_hSocket, buf, sizeof(buf), 0   (struct sockaddr*) &cli, &cli_len);
if(nbytes < 0) {
            printf( “receive failed with error %d\n",WSAGetLastError());
}
```

Client: Sending UDP Datagrams

```
SOCKET m_hSocket;
struct sockaddr_in srv;                    /* used by sendto() */

if((m_hSocket = socket(AF_INET, SOCK_DGRAM, 0)) < 0) { // create the socket
   printf( "Socket failed with error %d\n",WSAGetLastError());
}

/* sendto: send data to IP Address "192.168.3.153" port 80 */
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("192.168.3.153");

nbytes = sendto(m_hSocket, buf, sizeof(buf), 0 ,  (struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0) {
               printf( "Send failed with error %d\n",WSAGetLastError());
}
```

Client: bind

- The bind function is required on an unconnected socket before subsequent calls to the listen function. It is normally used to bind to either connection-oriented (stream) or connectionless (datagram) sockets.
- typically done by server only, No call to bind() is necessary before sending request to the server.
- But you may bind the UDP socket to a designated port before sendto.

UDP: Connected mode

- UDP sockets can be connected, making it convenient to interact with a specific server, or they can be unconnected, making it necessary for the application to specify the server's address each time it sends a message.
- However, Calling connect():
  - This simply tells the O.S. the address of the peer.
  - *No* handshake is made to establish that the peer exists.
  - *No* data of any kind is sent on the network as a result of calling **connect() on a UDP** socket.

Connected UDP

- Once a UDP socket is *connected:*
  - *a default destination address that can be used on subsequent send and recv calls.*
  - Any datagrams received from an address other than the destination address specified will be discarded.
    - can use **send()**
    - can use **recv():** only datagrams from the peer will be returned.
  - can use **sendto() with a null dest.** Address
  - Asynchronous errors will be returned to the process.

Asynchronous errors

- What happens if a client sends data to a server that is not running?
  - ICMP "port unreachable" error is generated by receiving host and sent to sending host.
  - The ICMP error may reach the sending host after sendto() has already returned!
  - The next call dealing with the socket could return the error.

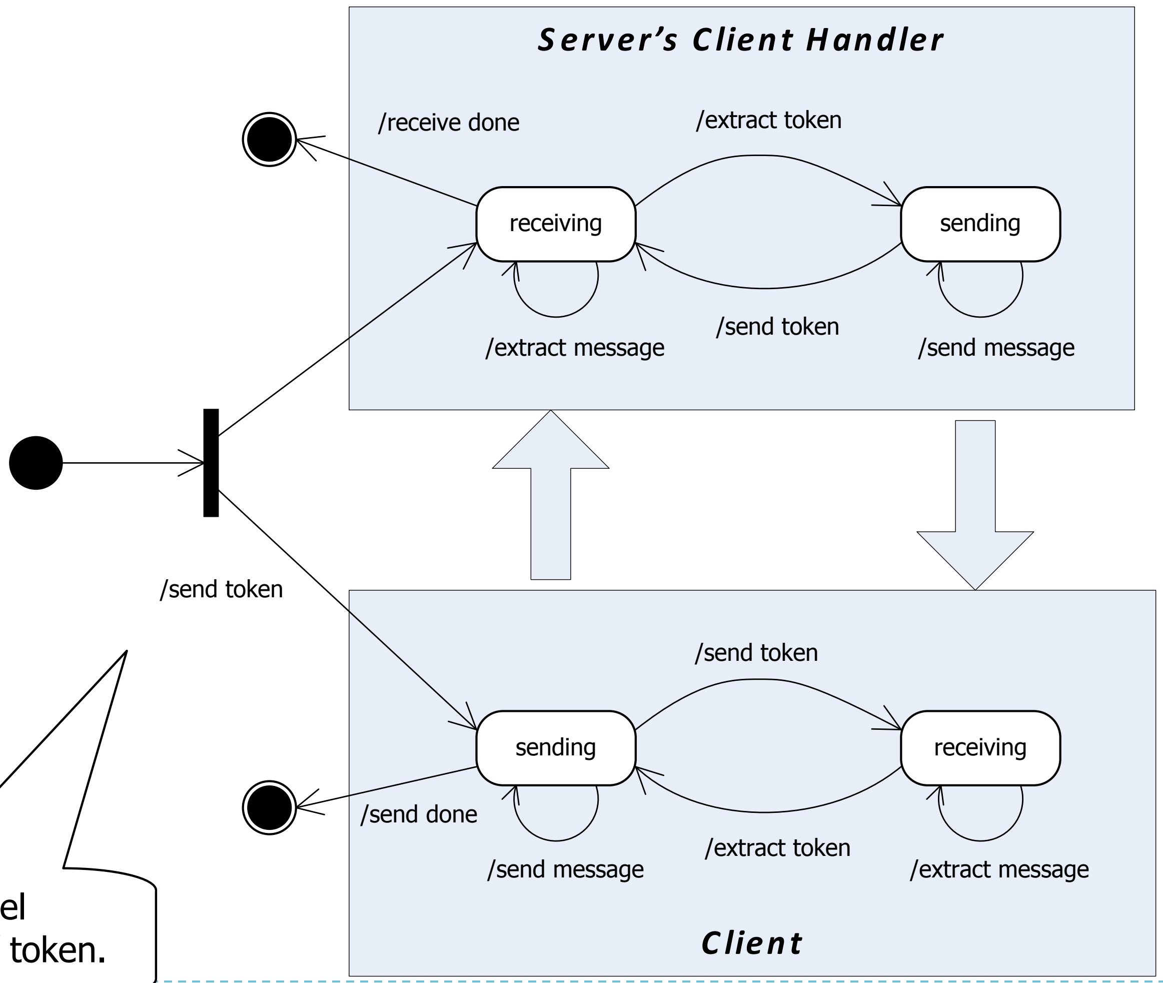
Back to UDP connect()

- Connect() is typically used with UDP when communication is with a single peer only.
- Many UDP clients use connect().
- Some servers (TFTP).
- It is possible to disconnect and connect the same socket to a new peer.

Talk Protocol

- The hardest part of a client/server socket communication design is to control the active participant
  - If single-threaded client and server both talk at the same time, their socket buffers will fill up and they both will block, e.g., deadlock.
  - If they both listen at the same time, again there is deadlock.
  - Often the best approach is to use separate send and receive threads

State Chart - Socket
Bilateral Communication Protocol



Each connection channel contains one "sending" token.

Message Length

- Another vexing issue is that the receiver may not know how long a sent message is.
  - so the receiver doesn't know how many bytes to pull from the stream to compose a message.
  - Often, the communication design will arrange to use message delimiters, fixed length messages, or message headers that carry the message length as a parameter.

Message Framing

- Sockets only understand arrays of bytes
  - Don’t know about strings, messages, or objects
- In order to send messages you simply build the message string, probably with XML
  - string msg = “<msg>message text goes here</msg>”
  - Then send(sock,msg,strlen(msg),flags)
- Receiving messages requires more work
  - Read socket one byte at a time and append to message string:
  - recv(sock,&ch,1,flags); msg.append(ch);
  - Search string msg from the back for </
  - Then collect the msg>

They're Everywhere

- Virtually every network and internet communication method uses sockets, often in a way that is invisible to an application designer.
  - Browser/server
  - ftp
  - SOAP
  - Network applications

References

- Ch.28, Programming With Microsoft Visual C++ NET 6th Ed. - George/ Kruglinski Shepherd. 2002
- Ch5, Network Programming for Microsoft Windows , 2nd Ed. - Anthony Jones, Jim Ohlund. 2002.
- Windows Sockets API Specification, http://www.sockets.com/winsock2.htm
- MSDN: Windows Socket 2, http://msdn.microsoft.com/en-us/library/ ms740673(VS.85).aspx
- The Winsock Programmer's FAQ, http://tangentsoft.net/wskfaq/
- Beej's Guide to Network Programming, http://beej.us/guide/bgnet/ output/html/multipage/index.html
- Ch.18.11, TCP Server Design, Richard Stevens, TCP/IP illustrated, volume 1
- Ch. 4.5 listen Function. Richard Stevens, Unix Network Programming, Volume 1
- Ch6,Ch7, Douglas Comer,David Stevens, *Internetworking With TCP/IP, Volume III*
- University of Cyprus, Course EPL420
- Syracuse University, Course CSE775

Appendix

TCP/UDP Server/Client Algorithm

Douglas E. Comer and David L. Stevens, *Internetworking With TCP/IP, Volume III*

Ch6,Ch7

TCP Client Algorithm
Comer and Stevens, Algorithm 6.1

- Find IP address and protocol port number on server
- Allocate a socket
- Allow TCP to allocate an arbitrary local port
- Connect the socket to the server
- Send requests and receive replies
- Close the connection

TCP Iterative Server Algorithm
Comer and Stevens, Algorithm 8.1

- Create a socket and bind to the well known address for the service offered
- Place socket in passive mode
- Accept next connection request and obtain a new socket
- Repeatedly receive requests and send replies
- When client is done, close the connection and return to waiting for connection requests

TCP Concurrent Server Algorithm
Comer and Stevens, Algorithm 8.4

- Master:
  - Create a socket and bind to the well known address for the service offered. Leave socket unconnected
  - Place socket in passive mode
  - Repeatedly call *accept* to get requests and create a new slave thread
- Slave:
  - Receive connection request and socket
  - Receive requests and send responses to client
  - Close connection and exit

UDP Client Algorithm
Comer and Stevens, Algorithm 6.2

- Find IP address and protocol port number on server
- Allocate a socket
- Allow UDP to allocate an arbitrary local port
- Specify the server
- Send requests and receive replies
- Close the socket

UDP Iterative Server Algorithm
Comer and Stevens, Algorithm 8.2

- Create a socket and bind to the well known address for the service offered
- Repeatedly receive requests and send replies

UDP Concurrent Server Algorithm
Comer and Stevens, Algorithm 8.3

- Master:
  - Create a socket and bind to the well known address for the service offered. Leave socket unconnected
  - Repeatedly call *recvfrom* to get requests and create a new slave thread
- Slave:
  - Receive request and access to socket
  - Form reply and send to client with *sendto*
  - Exit

SDK Advanced Winsock Samples

- Windows SDK
  - C:\Program Files\Microsoft SDKs\Windows\v7.1\Samples\netds\winsock
- iocp
  - This directory contains three sample programs that use I/O completion ports.
- overlap
  - This directory contains a sample server program that uses overlapped I/O.
- WSAPoll
  - This directory contains a basic sample program that demonstrates the use of the WSAPoll function.
- simple
  - This directory contains three basic sample programs that demonstrate the use of multiple threads by a server.
- accept
  - This directory contains a basic sample server and client program.