Network Programming for Windows 01:

Introduction to Winsock

jintaeks@dongseo.ac.kr
Division of Digital Contents, DSU

OSI model

- ✓ The **Open Systems Interconnection model** (**OSI model**) is a conceptual model that characterizes and standardizes the communication functions of a telecommunication or computing system without regard to their underlying internal structure and technology.
- ✓ Its goal is the interoperability of diverse communication systems with standard protocols.
- ✓ The model partitions a communication system into abstraction layers.
- ✓ The original version of the model defined seven layers.
- ✓ A layer serves the layer above it and is served by the layer below it.

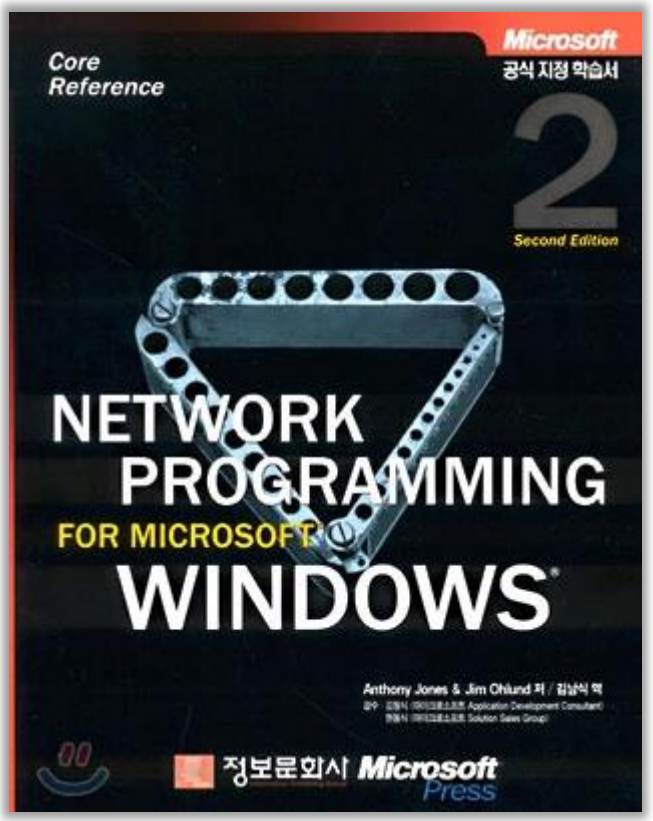
OSI Model			
Layer		Protocol data unit (PDU)	Function
Host Layers	7. Application	Data	High-level APIs, including resource sharing, remote file access
	6. Presentation		Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption
	5. Session		Managing communication sessions, i.e. continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes
Media Layers	4. Transport	Segment (TCP) / Datagram (UDP)	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing
	3. Network	Packet	Structuring and managing a multi-node network, including addressing, routing and traffic control
	2. Data link	Frame	Reliable transmission of data frames between two nodes connected by a physical layer
	1. Physical	Bit	Transmission and reception of raw bit streams over a physical medium

OSI (Open Source Interconnection) 7 Layer Model

Layer	Application/Example		Central Device/ Protocols			DOD4 Model
Application (7) Serves as the window for users and application processes to access the network services.	**End User layer** Program that opens what was sent or creates what is to be sent Resource sharing • Remote file access • Remote printer access • Directory services • Network management		**User Applications** SMTP		GATEWAY Can be used on all layers	Process
Presentation (6) Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network.	**Syntax layer** encrypt & decrypt (if needed) Character code translation • Data conversion • Data compression • Data encryption • **Character Set Translation**		JPEG/ASCII EBDIC/TIFF/GIF PICT			
Session (5) Allows session establishment between processes running on different stations.	**Synch & send to ports** (logical ports) Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc.		**Logical Ports** RPC/SQL/NFS NetBIOS names			
Transport (4) Ensures that messages are delivered error-free, in sequence, and with no losses or duplications.	**TCP** Host to Host, Flow Control Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing	PACKET FILTERING	TCP/SPX/UDP			Host to Host
Network (3) Controls the operations of the subnet, deciding which physical path the data takes.	**Packets** ("letter", contains IP address) Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting		**Routers** IP/IPX/ICMP			Internet
Data Link (2) Provides error-free transfer of data frames from one node to another over the Physical layer.	**Frames** ("envelopes", contains MAC address) [NIC card — Switch — NIC card] (end to end) Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control		**Switch Bridge WAP** PPP/SLIP	Land Based Layers		Network
Physical (1) Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium.	**Physical structure** Cables, hubs, etc. Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts		**Hub**			

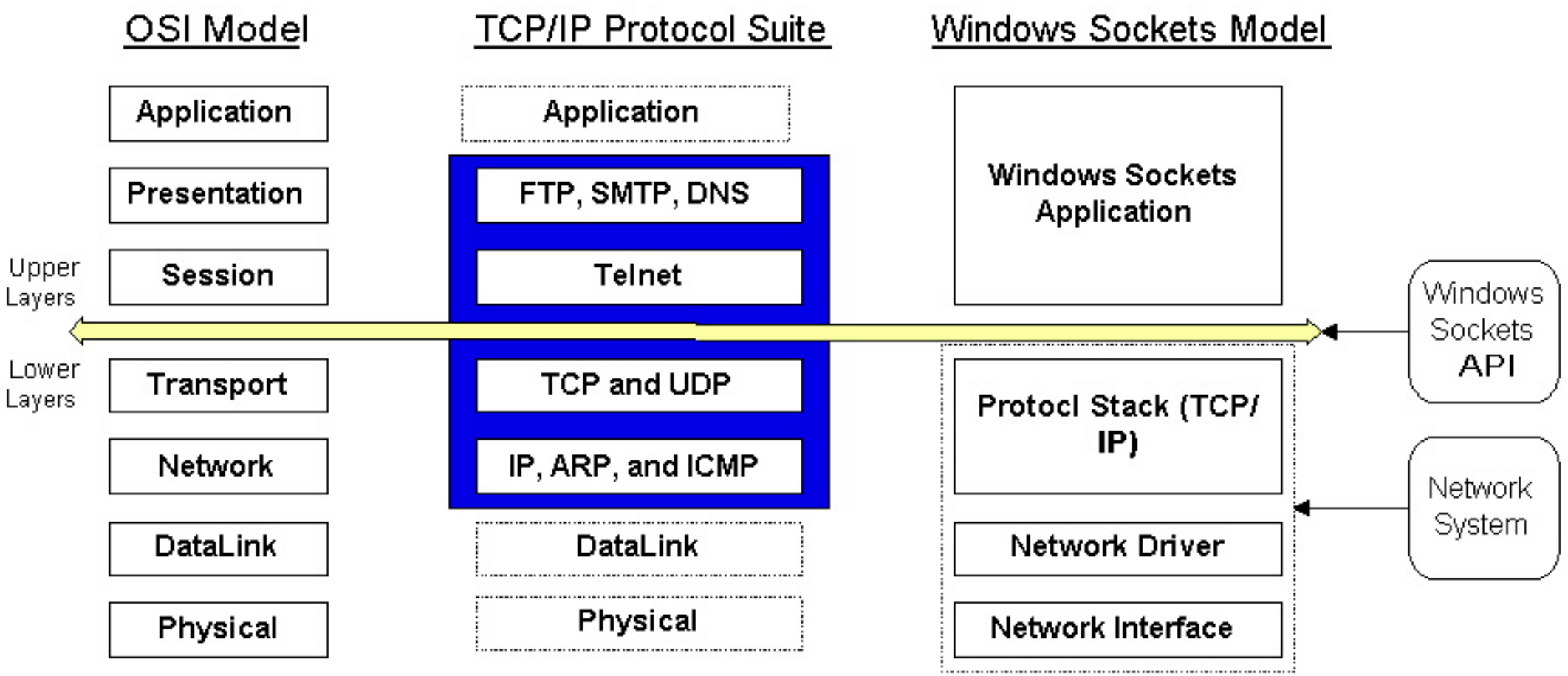
Introduction to Winsock

- ✓ Winsock Headers and Libraries
- ✓ Initializing Winsock
- ✓ Error Checking and Handling
- ✓ Addressing a Protocol
- ✓ Creating a Socket
- ✓ Connection-Oriented Communication
- ✓ Connectionless Communication
- ✓ Miscellaneous APIs
- ~~✓ Windows CE(skip)~~
- ✓ Conclusion



- ✓ **Winsock** is a standard **application programming interface (API)** that allows two or more applications to communicate across a network.
- ✓ Winsock is **a network programming interface** and not a protocol.
- ✓ Winsock provides the programming interface for applications to communicate using popular network protocols such as **Transmission Control Protocol/Internet Protocol (TCP/IP)** and Internetwork Packet Exchange (IPX).

OSI Layer and Winsock



Winsock Headers and Libraries

- ✓ When developing new applications you should target the Winsock 2 specification by including **WINSOCK2.H** in your application.
- ✓ When compiling your application with WINSOCK2.H, you should link with **WS2_32.LIB** library.

```
#include <winsock2.h>
#include <stdio.h>

#pragma comment(lib, "ws2_32.lib")

void main(int argc, char **argv)
{
```

Initializing Winsock

- ✓ Every Winsock application must load the appropriate version of the Winsock DLL.

```
int WSAStartup(
    WORD wVersionRequested,
    LPWSADATA lpWSAData
    );
```

- ✓ You can use the handy macro MAKEWORD(x, y), in which x is the high byte and y is the low byte, to obtain the correct value for wVersionRequested.

- ✓ The lpWSAData parameter is a pointer to a LPWSADATA structure that WSAStartup fills with information related to the version of the library it loads.

```
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;
```

- ✓ When your application is completely finished using the Winsock interface, you should call WSACleanup, which allows Winsock to free up any resources allocated by Winsock.

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

Error Checking and Handling

- ✓ The most common return value for an unsuccessful Winsock call is SOCKET_ERROR.
- ✓ You can use the function WSAGetLastError to obtain a code that indicates specifically what happened.

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

Skeleton Winsock application.

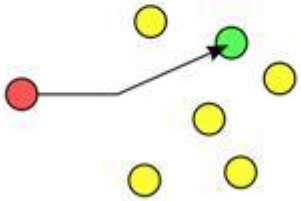
```
void main(void)
{
    WSADATA wsaData;

    if ((Ret = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)
    {
        printf("WSAStartup failed with error %d₩n", Ret);
        return;
    }

    // Setup Winsock communication code here

    // When your application is finished call WSACleanup
    if (WSACleanup() == SOCKET_ERROR)
    {
        printf("WSACleanup failed with error %d₩n", WSAGetLastError());
    }
}
```

Addressing a Protocol

- ✓ Throughout the remainder of this chapter, we will demonstrate the basics of how to set up Winsock communication using the IPv4 protocol.
- ✓ By design, **IP** is a **connectionless** protocol and doesn't guarantee data delivery. Two higher-level protocols—Transmission Control Protocol (**TCP**) and User Datagram Protocol (**UDP**)—are used for **connection-oriented** and connectionless data communication over IP.

TCP

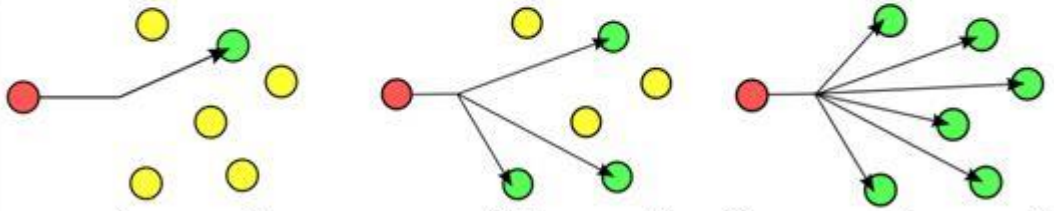
- **Slower but reliable transfers**
- **Typical applications:**
  - Email
  - Web browsing



unicast

UDP

- **Fast but non-guaranteed transfers ("best effort")**
- **Typical applications:**
  - VoIP
  - Music streaming



unicast **multicast** **broadcast**

- ✓ In IPv4, computers are assigned an address that is represented as a 32-bit quantity.

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

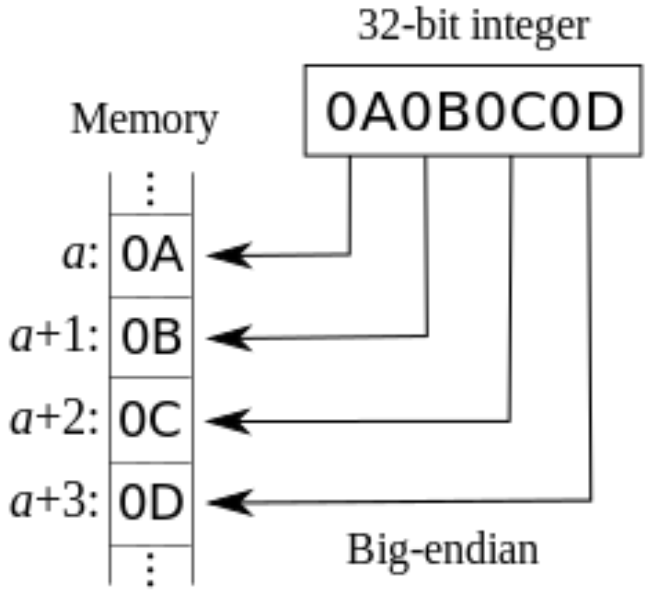
- ✓ The final field, sin_zero, functions only as **padding** to make the SOCKADDR_IN structure the same size as the SOCKADDR structure.

- ✓ inet_addr converts a dotted IP address to a 32-bit unsigned long integer quantity.

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

Byte Ordering

- ✓ Different computer processors represent numbers in **big-endian** and **little-endian** form.
- ✓ For example, on **Intel x86** processors, multibyte numbers are represented in **little-endian** form: the bytes are ordered from least significant to most significant.
  - saved from little end
- ✓ **host-byte** order.
- ✓ **network-byte** order.



convert a number from host-byte to network-byte order

```
u_long htonl(u_long hostlong);

int WSAHtonl(
    SOCKET s,
    u_long hostlong,
    u_long FAR * lpnetlong
    );

u_short htons(u_short hostshort);

int WSAHtons(
    SOCKET s,
    u_short hostshort,
    u_short FAR * lpnetshort
    );
```

convert network-byte order to host-byte order.

```
u_long ntohl(u_long netlong);

int WSANtohl(
    SOCKET s,
    u_long netlong,
    u_long FAR * lphostlong
    );

u_short ntohs(u_short netshort);

int WSANtohs(
    SOCKET s,
    u_short netshort,
    u_short FAR * lphostshort
    );
```

```
SOCKADDR_IN InternetAddr;
INT nPortId = 5150;

InternetAddr.sin_family = AF_INET;

// Convert the proposed dotted Internet address 136.149.3.29
// to a four-byte integer, and assign it to sin_addr

InternetAddr.sin_addr.s_addr = inet_addr("136.149.3.29");

// The nPortId variable is stored in host-byte order. Convert
// nPortId to network-byte order, and assign it to sin_port.

InternetAddr.sin_port = htons(nPortId);
```

Creating a Socket

- ✓ There are two functions that can be used to create a socket: socket and WSASocket.

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

- ✓ af: the protocol's address family. For IPv4 protocol, you should set this field to **AF_INET**.
- ✓ type: When you are creating a socket to use TCP/IP, set this field to **SOCK_STREAM**, for UDP/IP use **SOCK_DGRAM**.
- ✓ protocol: For TCP you should set this field to **IPPROTO_TCP**; for UDP use **IPPROTO_UDP**.

Connection-Oriented Communication

- ✓ We'll first discuss how to develop a server by **listening** for client **connections** and explore the process for **accepting** or **rejecting** a connection.
- ✓ We'll describe how to develop a client by initiating a connection to a server.
- ✓ Finally, we'll discuss how data is transferred in a **connection-oriented** session.

Server API Functions

- ✓ A **server** is a process that waits for any number of client connections with the purpose of servicing their requests.
- ✓ The first step in Winsock is to **create** a socket with either the ***socket*** or *WSASocket* call and **bind** the 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 (appropriately enough) with the ***listen*** API function.
- ✓ Finally, when a client attempts a connection, the server must **accept** the connection with either the ***accept*** or *WSAAccept* call.

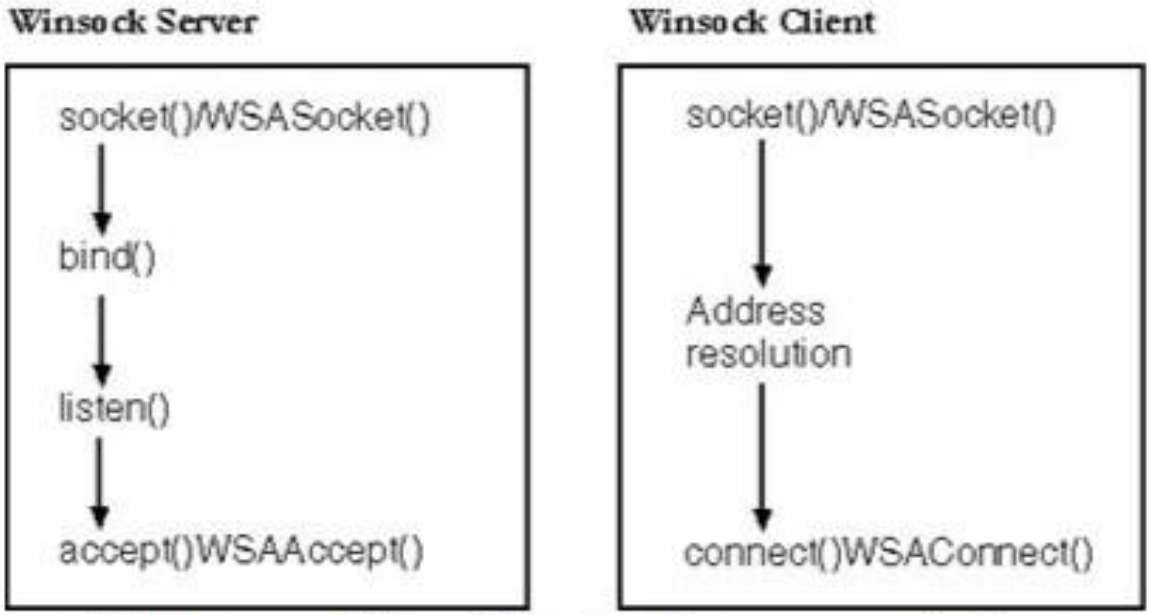


***Figure 1-1** Winsock basics for server and client*

✓ Once the socket of a particular protocol is created, you must **bind** it to a well-known address.

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

```
SOCKET              s;
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));
```

Listening

- ✓ The next piece of the equation is to put the socket into **listening** mode. The API function that tells a socket to wait for incoming connections is listen.

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

- ✓ The **backlog** parameter specifies the maximum queue length for pending connections.

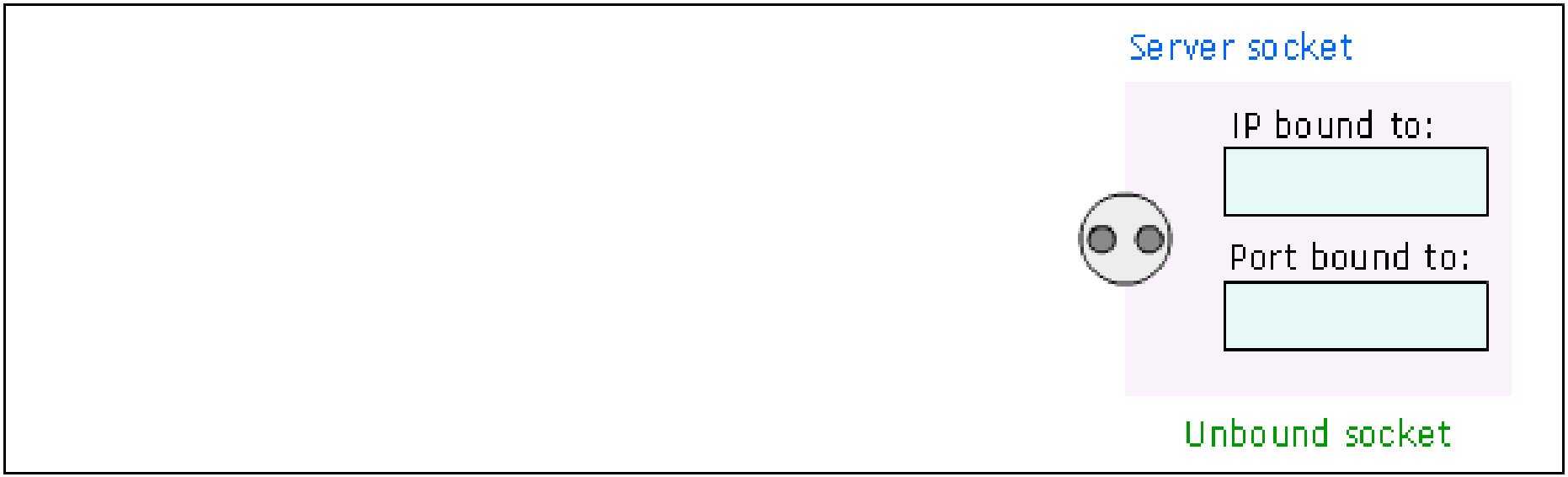
Accepting Connections

- ✓ Now you're ready to **accept** client connections. This is accomplished with the ***accept***, WSAAccept, or AcceptEx function.

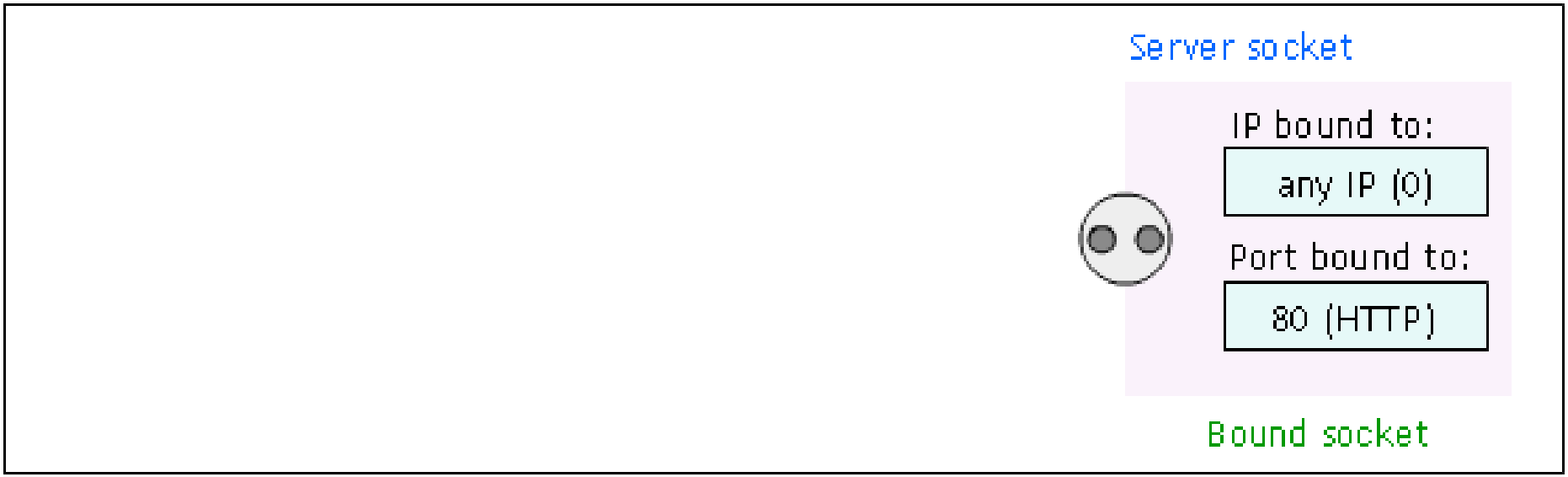
```
SOCKET accept(
    SOCKET s,
struct sockaddr FAR* addr,
    int FAR* addrlen
    );
```

  - addr: should be the address of a valid SOCKADDR_IN structure.
  - addrlen: should be a reference to the length of the SOCKADDR_IN structure.

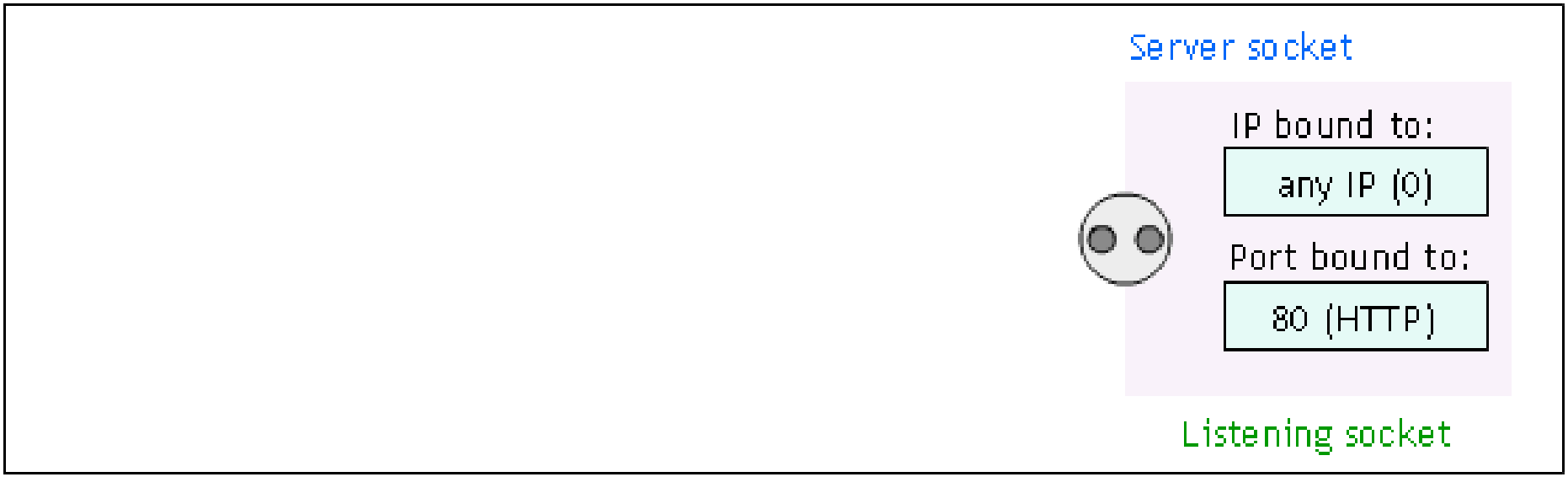
The server socket is created



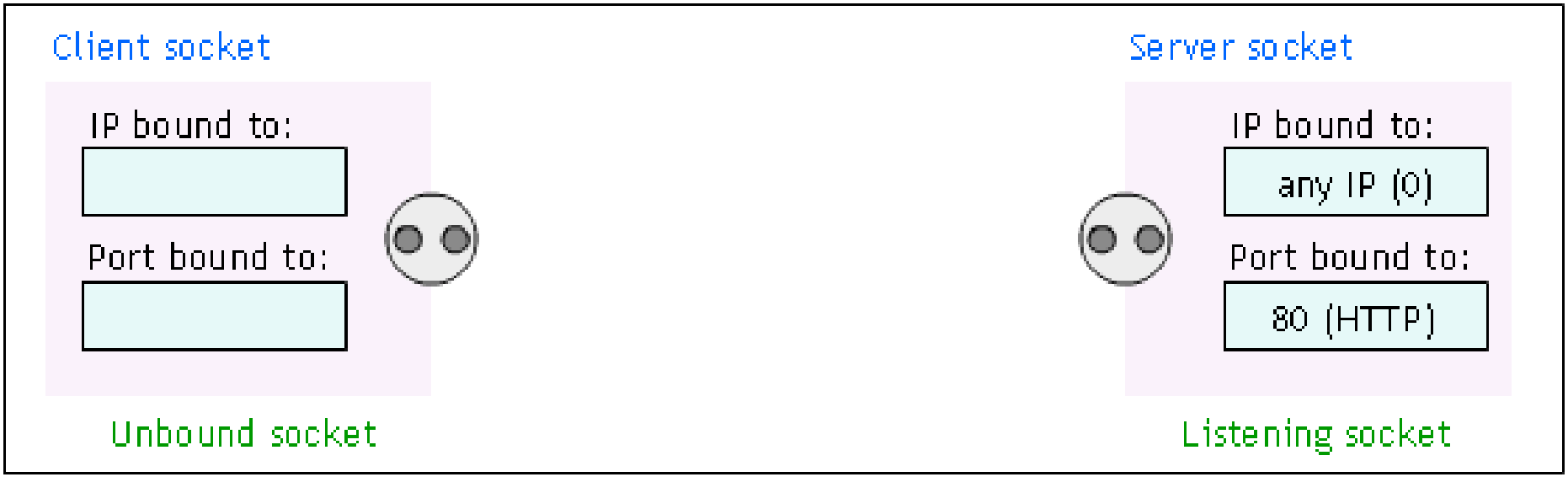
The server socket is bound



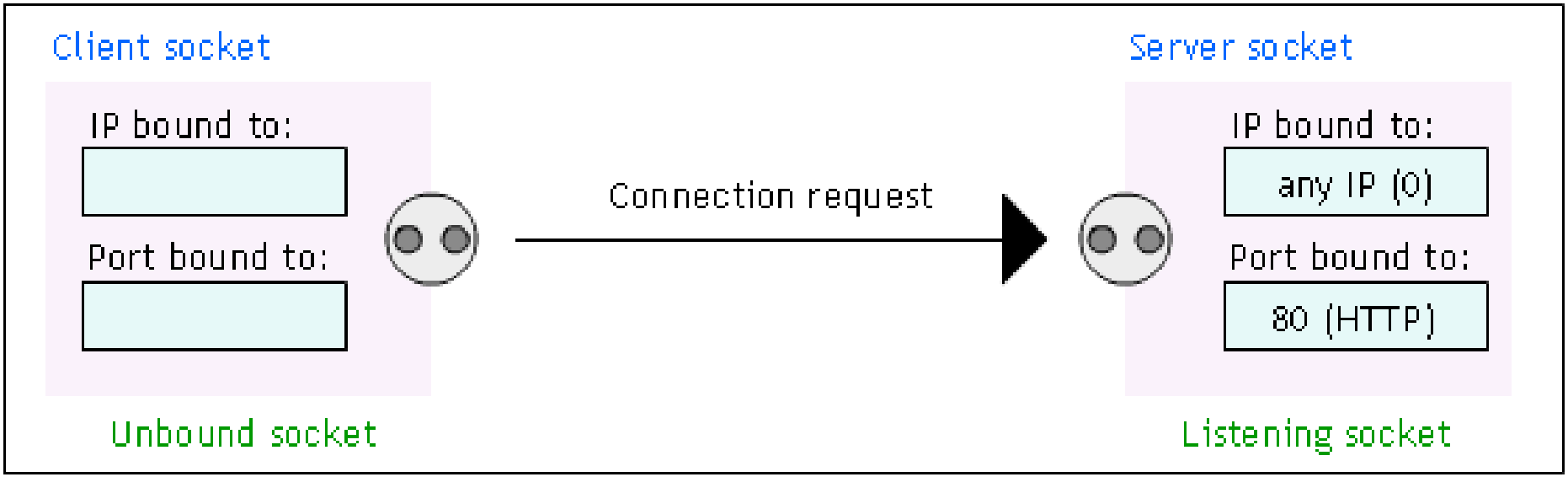
The server socket is listening



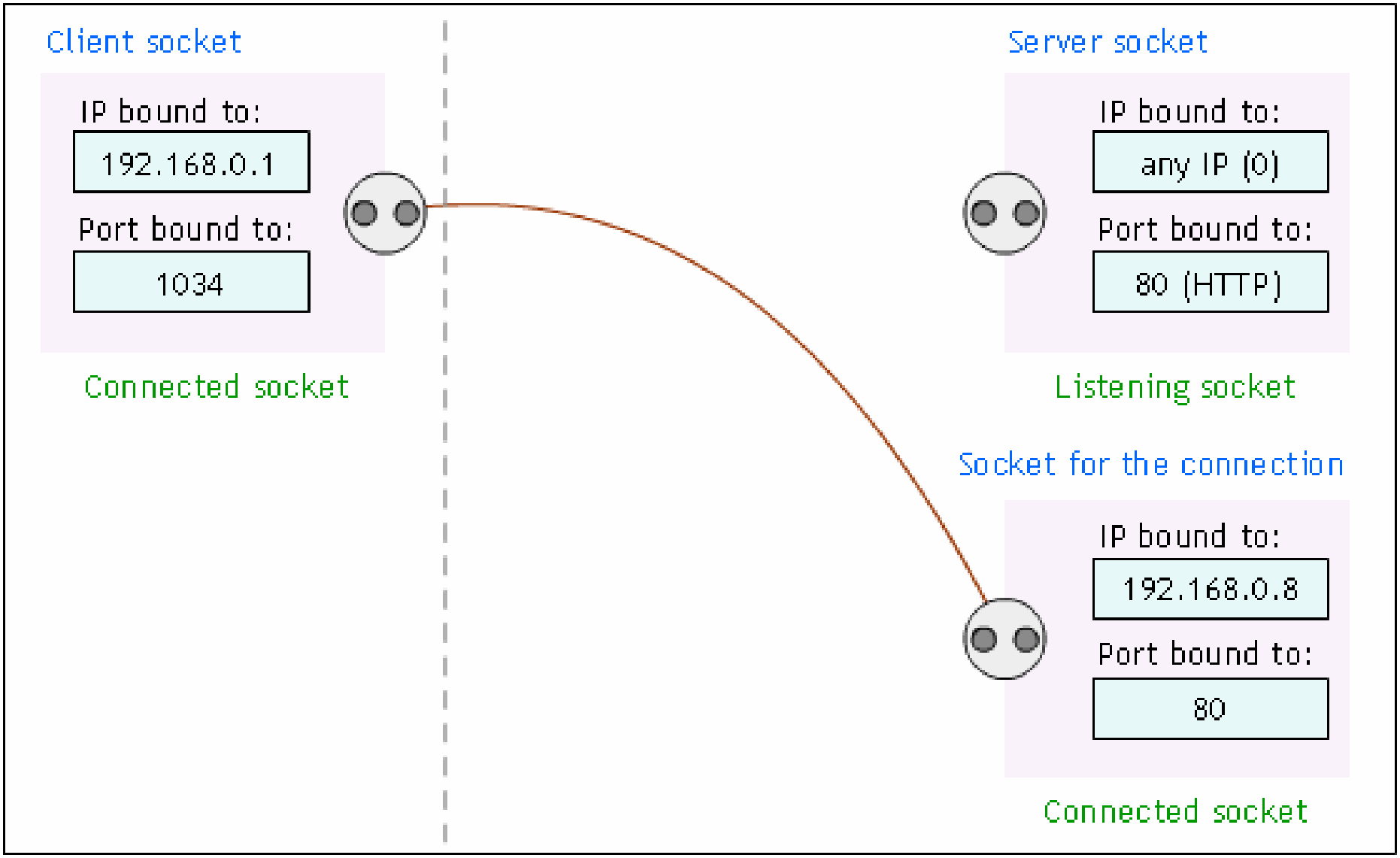
A client creates a socket



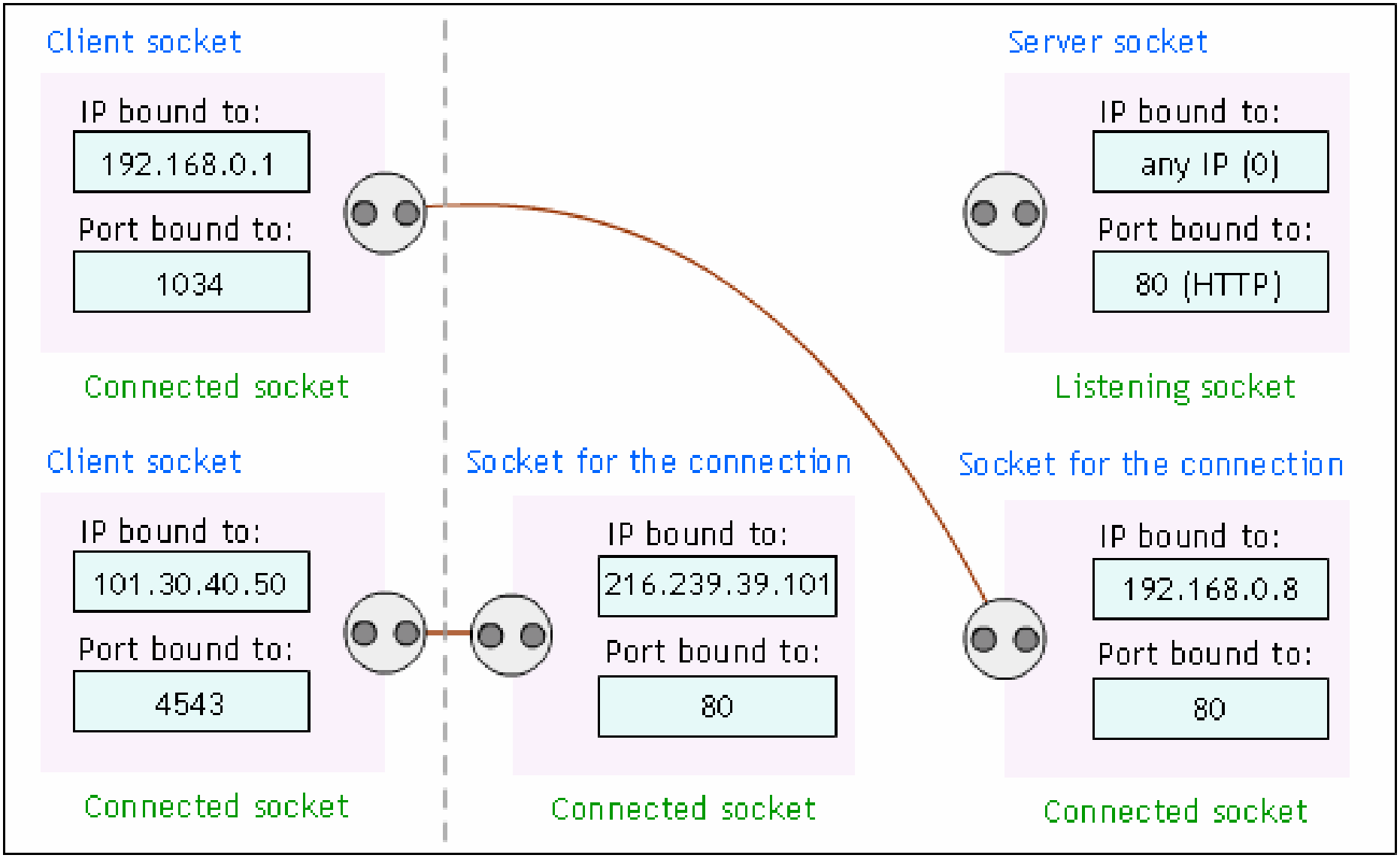
The client socket tries to connect



The server accepts the request



Another client connects



```
#include <winsock2.h>

void main(void)
{
    WSADATA             wsaData;
    SOCKET              ListeningSocket;
    SOCKET              NewConnection;
    SOCKADDR_IN         ServerAddr;
    SOCKADDR_IN         ClientAddr;
    int                 Port = 5150;

    // Initialize Winsock version 2.2
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Create a new socket to listen for client connections.
    ListeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_port = htons(Port);
    ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    // Associate the address information with the socket using bind.
    bind(ListeningSocket, (SOCKADDR *)&ServerAddr,
        sizeof(ServerAddr));
```

```
    // Listen for client connections. We used a backlog of 5, which
    // is normal for many applications.
    listen(ListeningSocket, 5);

    // Accept a new connection when one arrives.
    NewConnection = accept(ListeningSocket, (SOCKADDR *)
        &ClientAddr, &ClientAddrLen));

    // At this point you can do two things with these sockets. Wait
    // for more connections by calling accept again on ListeningSocket
    // and start sending or receiving data on NewConnection. We will
    // describe how to send and receive data later in the chapter.

    closesocket(NewConnection);
    closesocket(ListeningSocket);

    // When your application is finished handling the connections,
    // call WSACleanup.

    WSACleanup();
}
```

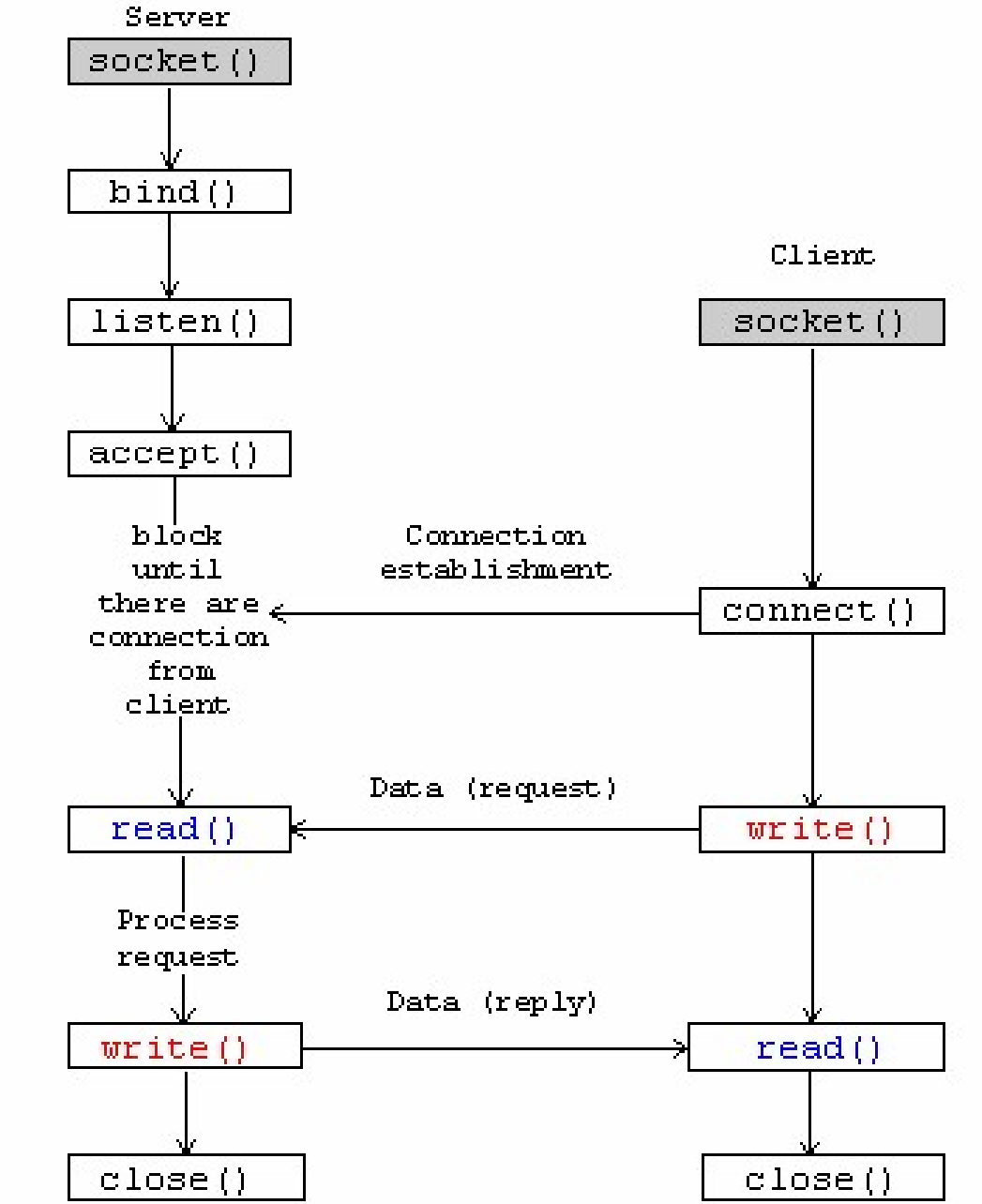
Client API Functions

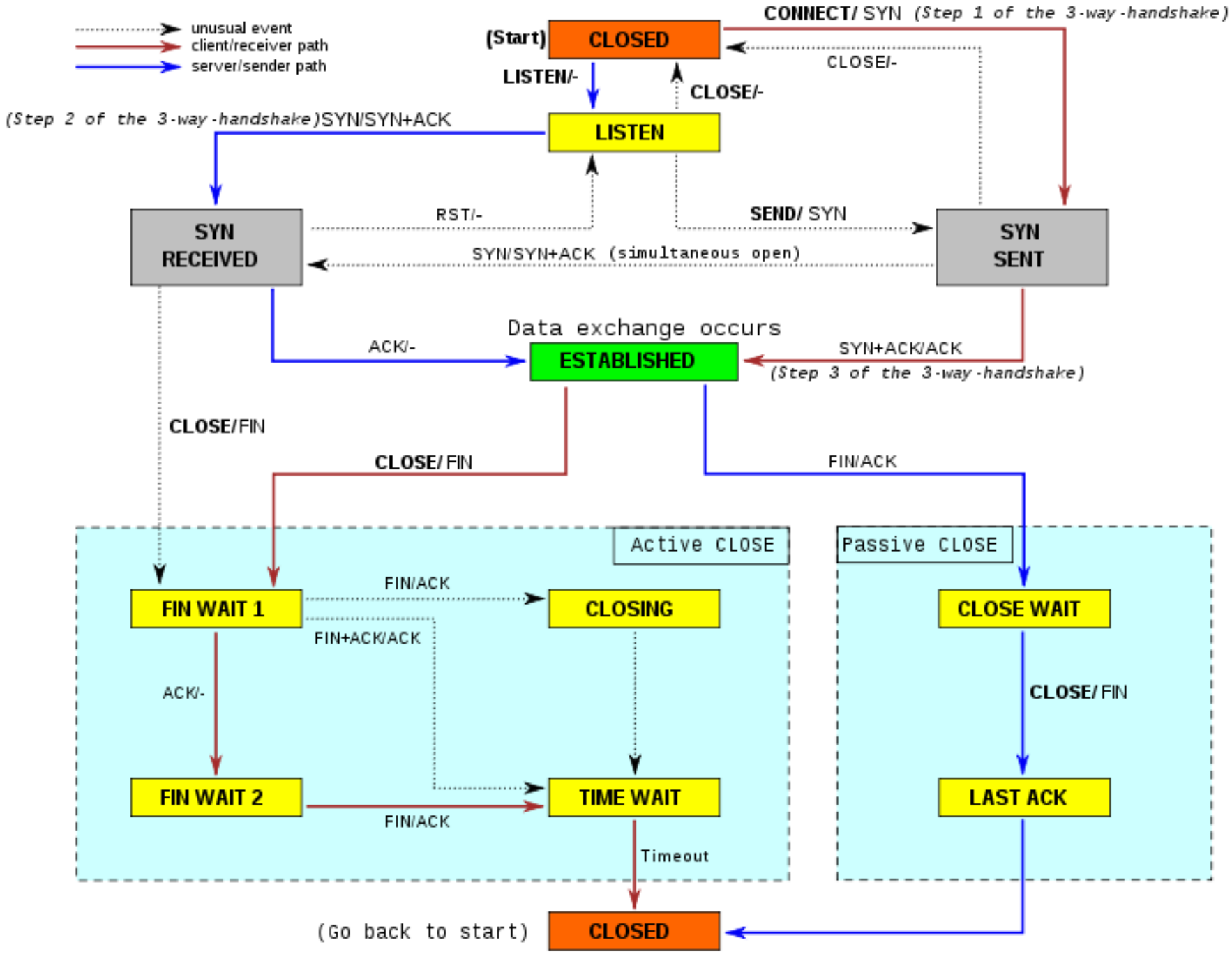
- ✓ **Create** a socket.
- ✓ Set up a SOCKADDR address structure with the name of server you are going to connect to (dependent on underlying protocol). For TCP/IP, this is the server's IP address and port number its application is listening on.
- ✓ Initiate the connection with ***connect*** or WSAConnect.

```
void main(void)
{
    WSADATA             wsaData;
    SOCKET              s;
    SOCKADDR_IN         ServerAddr;
    int                 Port = 5150;

    // Initialize Winsock version 2.2
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Create a new socket to make a client connection.
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_port = htons(Port);
    ServerAddr.sin_addr.s_addr = inet_addr("136.149.3.29");

    // Make a connection to the server with socket s.
    connect(s, (SOCKADDR *)&ServerAddr, sizeof(ServerAddr));
```

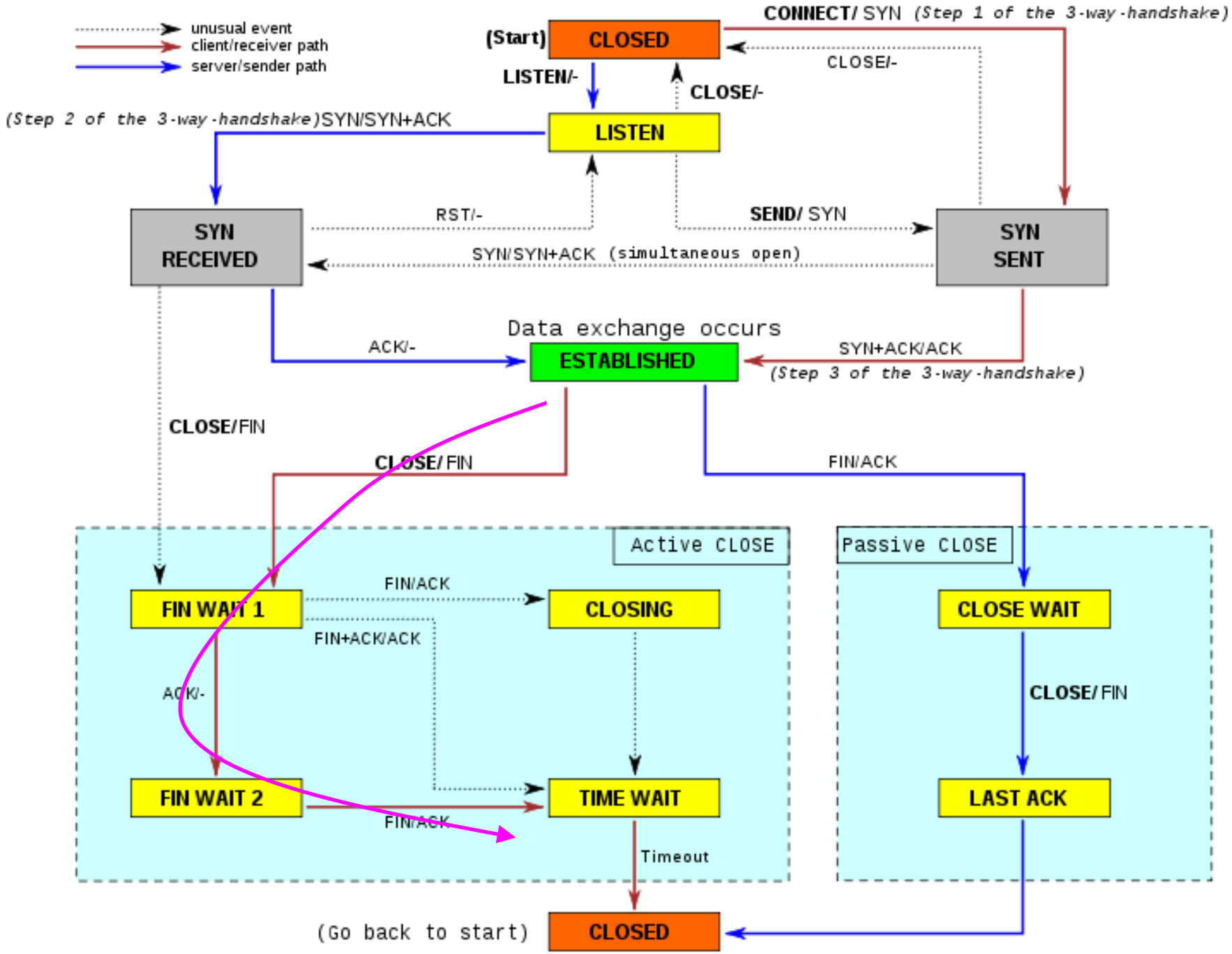
```
    // At this point you can start sending or receiving data on
    // the socket s.

    closesocket(s);

    WSACleanup();
}
```

TCP States

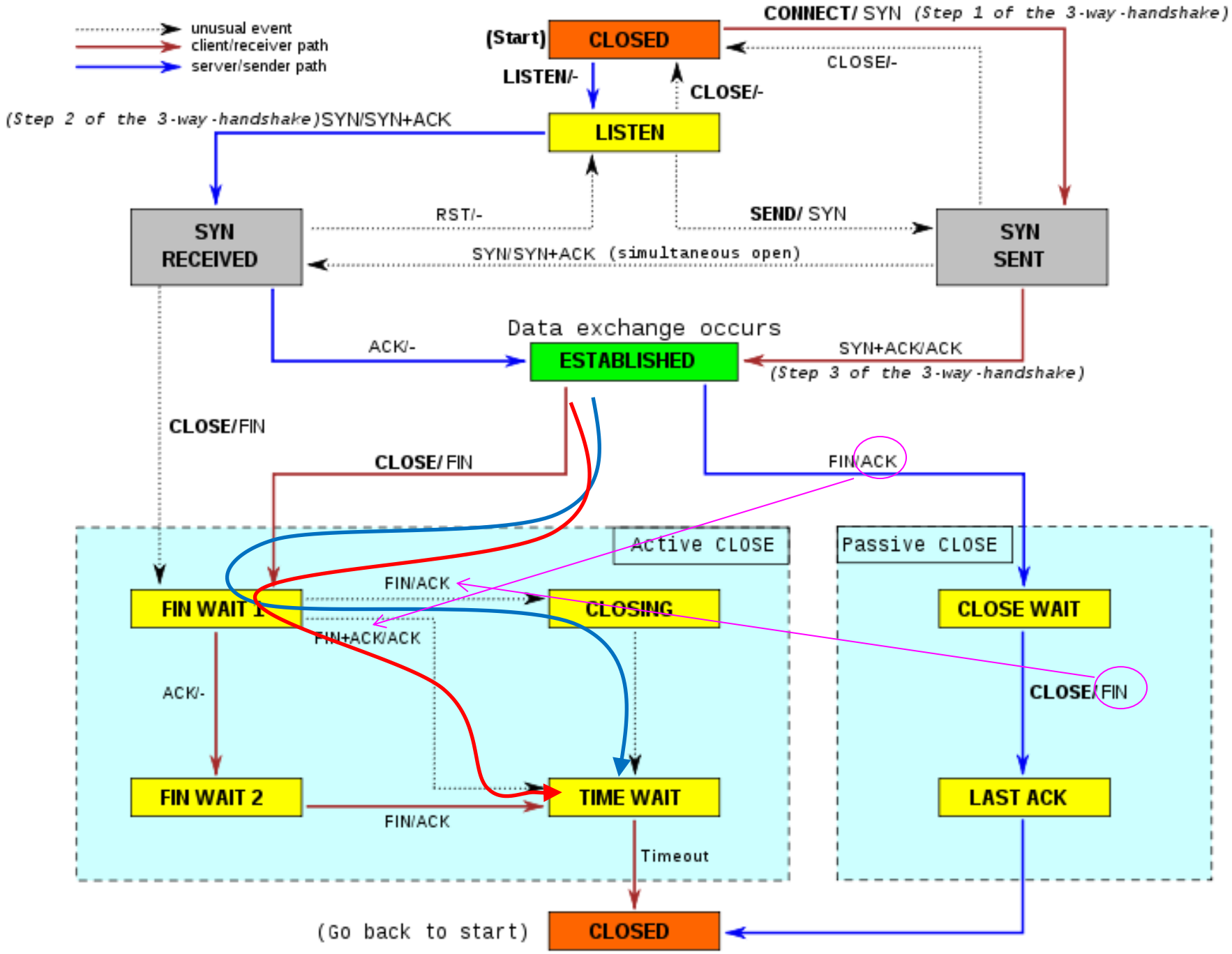

Server
socket()
bind()
listen()
accept()
block until there are connection from client
Client
socket()
Connection establishment
connect()
Data (request)
read()
write()
Process request
Data (reply)
write()
read()
close()
close()


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/ SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake)SYN/SYN+ACK
LISTEN
SYN RECEIVED
RST/-
SEND/ SYN
SYN SENT
SYN/SYN+ACK (simultaneous open)
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/ FIN
CLOSE/ FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
FIN+ACK/ACK
CLOSE WAIT
ACK/-
CLOSE/ FIN
FIN WAIT 2
TIME WAIT
FIN/ACK
LAST ACK
Timeout
(Go back to start)
CLOSED

- ✓ **CLOSED**: There is no connection.
- ✓ **LISTEN**: The local end-point is waiting for a connection request from a remote end-point i.e. a passive open was performed.
- ✓ **SYN-SENT**: The first step of the **three-way connection handshake** was performed. A connection request has been sent to a remote end-point i.e. an active open was performed.
- ✓ **SYN-RECEIVED**: The second step of the three-way connection handshake was performed. An acknowledgement for the received connection request as well as a connection request has been sent to the remote end-point.
- ✓ **ESTABLISHED**: The third step of the three-way connection handshake was performed. The connection is open.


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/ SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake)SYN/SYN+ACK
LISTEN
SYN RECEIVED
RST/-
SEND/ SYN
SYN SENT
SYN/SYN+ACK (simultaneous open)
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
FIN+ACK/ACK
CLOSE WAIT
ACK/-
CLOSE/FIN
FIN WAIT 2
TIME WAIT
FIN/ACK
LAST ACK
Timeout
(Go back to start)
CLOSED

- ✓ **FIN-WAIT-1**: The first step of an active close (four-way handshake) was performed. The local end-point has sent a connection termination request to the remote end-point.
- ✓ **CLOSE-WAIT**: The local end-point has received a connection termination request and acknowledged it e.g. a passive close has been performed and the local end-point needs to perform an active close to leave this state.
- ✓ **FIN-WAIT-2**: The remote end-point has sent an acknowledgement for the previously sent connection termination request. The local end-point waits for an active connection termination request from the remote end-point.
- ✓ **LAST-ACK**: The local end-point has performed a passive close and has initiated an active close by sending a connection termination request to the remote end-point.


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/ SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake)SYN/SYN+ACK
LISTEN
SYN RECEIVED
RST/-
SEND/ SYN
SYN SENT
SYN/SYN+ACK (simultaneous open)
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
FIN+ACK/ACK
ACK/-
CLOSE/FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
(Go back to start)
CLOSED

- ✓ **CLOSING**: The local end-point is waiting for an acknowledgement for a connection termination request before going to the TIME-WAIT state.
- ✓ **TIME-WAIT**: The local end-point waits for twice the maximum segment lifetime (MSL) to pass before going to CLOSED to be sure that the remote end-point received the acknowledgement.

Data Transmission

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

- ✓ flags: can be either 0, MSG_DONTROUTE, or MSG_OOB. Alternatively, the flags parameter can be a bitwise OR any of those flags.
- ✓ On a good return, send **returns the number of bytes** sent; otherwise, if an error occurs, SOCKET_ERROR will be returned.

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

- ✓ The last two parameters, **lpOverlapped** and lpCompletionRoutine, are used for overlapped I/O.
- ✓ **Overlapped I/O is one of the asynchronous I/O models** that Winsock supports

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

- ✓ flags: can be one of the following values: **0**, **MSG_PEEK**, or **MSG_OOB**.
- ✓ MSG_PEEK causes the data that is available to be copied into the supplied receive buffer, but this data is **not removed from the system's buffer**.

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

- ✓ The **lpOverlapped** and lpCompletionRoutine parameters are used in overlapped I/O operations

Stream Protocols

- ✓ Most connection-oriented communication, such as TCP, is **streaming protocols**.
- ✓ A streaming protocol is one that the sender and receiver may break up or coalesce data into smaller or larger groups.
- ✓ You are not guaranteed to read or write the amount of data you request.

```
char sendbuff[2048];
int  nBytes = 2048;

// Fill sendbuff with 2048 bytes of data

// Assume s is a valid, connected stream socket
ret = send(s, sendbuff, nBytes, 0);
```

- ✓ It is possible for send to return having sent less than 2048 bytes. The ret variable will be set to the number of bytes sent.

- ✓ In the case of our send call, there might be buffer space to hold only 1024 bytes, in which case you would have to resubmit the remaining 1024 bytes.

```
char sendbuff[2048];
int  nBytes = 2048,nLeft,idx;

// Fill sendbuff with 2048 bytes of data

// Assume s is a valid, connected stream socket
nLeft = nBytes;
idx = 0;

while (nLeft > 0)
{
    ret = send(s, &sendbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Error
    }
    nLeft -= ret;
    idx += ret;
}
```

- ✓ If all the messages are the same size, life is pretty simple, and the code for reading, say, 512-byte messages would look like this.
- ✓ Things get a little complicated if your message sizes vary.
  - How?

```
char    recvbuff[1024];
int     ret,nLeft,idx;

nLeft = 512;
idx = 0;

while (nLeft > 0)
{
    ret = recv(s, &recvbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Error
    }
    idx += ret;
    nLeft -= ret;
}
```

Breaking the Connection

- ✓ Once you are finished with a socket connection, you must close it and release any resources associated with that socket handle.
- ✓ To actually release the resources associated with an open socket handle, use the **closesocket** call.
- ✓ a connection should be **gracefully terminated**.

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

- ✓ how: can be SD_RECEIVE, SD_SEND, or SD_BOTH. For SD_RECEIVE, subsequent calls to any receive function on the socket are disallowed.

closesocket

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

- ✓ Calling closesocket releases the socket descriptor and any further calls using the socket fail with WSAENOTSOCK.
- ✓ If there are no other references to this socket, all resources associated with the descriptor are released.
- ✓ Pending synchronous calls issued by any thread in this process are canceled without posting any notification messages.
- ✓ Pending overlapped operations are also canceled.

Practice: Client

- ✓ Modify the client behavior as follows:
- ✓ 1) Get input from the user.
- ✓ 2) Send the input string to the server.
- ✓ 3) Print the string received from the server.
- ✓ 4) If the message received from the server is bye, terminate the client program.
- ✓ 5) Go to step 1).

Practice: Server

- ✓ Modify the server behavior as follows:
- ✓ 1) Get data from the client.
- ✓ 2) Print the string received from the client.
- ✓ 3) Echo back the received data
  - Send received string to the client
- ✓ 4) If the message received from the client is "bye", terminate the server.
- ✓ 5) Go to step 1).

Connectionless Communication

- ✓ In IP, connectionless communication is accomplished through **UDP/IP**.
- ✓ UDP doesn't guarantee reliable data transmission and is capable of sending data to multiple destinations and receiving it from multiple sources.
- ✓ Data is transmitted using **datagrams**.

Receiver

- ✓ First, create the socket with either **socket** or WSASocket.
- ✓ Next, **bind** the socket to the interface on which you wish to receive data.
- ✓ The difference with connectionless sockets is that you do not call **listen** or **accept**.

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

```
int WSARecvFrom(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    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. You can supply **one or more WSABUF buffers** to WSARecvFrom with **dwBufferCount** indicating this.

```
#include <winsock2.h>

void main(void)
{
    WSADATA             wsaData;
    SOCKET              ReceivingSocket;
    SOCKADDR_IN         ReceiverAddr;
    int                 Port = 5150;
    char                ReceiveBuf[1024];
    int                 BufLength = 1024;
    SOCKADDR_IN         SenderAddr;
    int                 SenderAddrSize = sizeof(SenderAddr);

    // Initialize Winsock version 2.2

    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Create a new socket to receive datagrams on.

    ReceivingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    // Set up a SOCKADDR_IN structure that will tell bind that we
    // want to receive datagrams from all interfaces using port
    // 5150.
```

```
    ReceiverAddr.sin_family = AF_INET;
    ReceiverAddr.sin_port = htons(Port);
    ReceiverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    // Associate the address information with the socket using bind.

    bind(ReceivingSocket, (SOCKADDR *)&SenderAddr, sizeof(SenderAddr));

    // At this point you can receive datagrams on your bound socket.
    recvfrom(ReceivingSocket, ReceiveBuf, BufLength, 0,
        (SOCKADDR *)&SenderAddr, &SenderAddrSize);

    // When your application is finished receiving datagrams close
    // the socket.

    closesocket(ReceivingSocket);

    // When your application is finished call WSACleanup.

    WSACleanup();
}
```

Sender

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

- ✓ to: a pointer to a SOCKADDR structure with the destination address of the workstation to receive the data.

```
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
    );
```

```
#include <winsock2.h>

void main(void)
{
    WSADATA             wsaData;
    SOCKET              SendingSocket;
    SOCKADDR_IN         ReceiverAddr;
    int                 Port = 5150;
    char                SendBuf[1024];
    int                 BufLength = 1024;

    // Initialize Winsock version 2.2

    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Create a new socket to receive datagrams on.

    SendingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

```
    // Set up a SOCKADDR_IN structure that will identify who we
    // will send datagrams to. For demonstration purposes, let's
    // assume our receiver's IP address is 136.149.3.29 and waits
    // for datagrams on port 5150.

    ReceiverAddr.sin_family = AF_INET;
    ReceiverAddr.sin_port = htons(Port);
    ReceiverAddr.sin_addr.s_addr = inet_addr("136.149.3.29");

    // Send a datagram to the receiver.

    sendto(SendingSocket, SendBuf, BufLength, 0,
        (SOCKADDR *)&ReceiverAddr, sizeof(RecieverAddr));

    // When your application is finished sending datagrams close
    // the socket.

    closesocket(SendingSocket);

    // When your application is finished call WSACleanup.

    WSACleanup();
}
```

Releasing Socket Resources

- ✓ Because there is no connection with connectionless protocols, there is no formal shutdown or graceful closing of the connection.
- ✓ When the sender or the receiver is finished sending or receiving data, it simply calls the **closesocket** function on the socket handle.

Miscellaneous APIs

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

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

```
int WSADuplicateSocket(
    SOCKET s,
    DWORD dwProcessId,
    LPWSAPROTOCOL_INFO lpProtocolInfo
    );
```

Conclusion

- ✓ For **connection-oriented** communication, we demonstrated how to accept a client connection and how to establish a client connection to a server.
- ✓ We covered the semantics for **session-oriented** data-send operations and data-receive operations.

Practice

- ✓ Samples → chapter01 projects

References

- ✓ http://www.winsocketdotnetworkprogramming.com/winsock2programming/
- ✓ http://www.madwizard.org/programming/tutorials/netcpp/4
- ✓ https://en.wikipedia.org/wiki/OSI_model
- ✓ https://benohead.com/tcp-about-fin_wait_2-time_wait-and-close_wait/

MY **BRIGHT** FUTURE

DSU Dongseo University 동서대학교

DSU Dongseo Univ 동서대학교