Developing Transport-Independent Applications Using the Windows Sockets Interface

Click to open or copy the project files

Presented by: David Treadwell

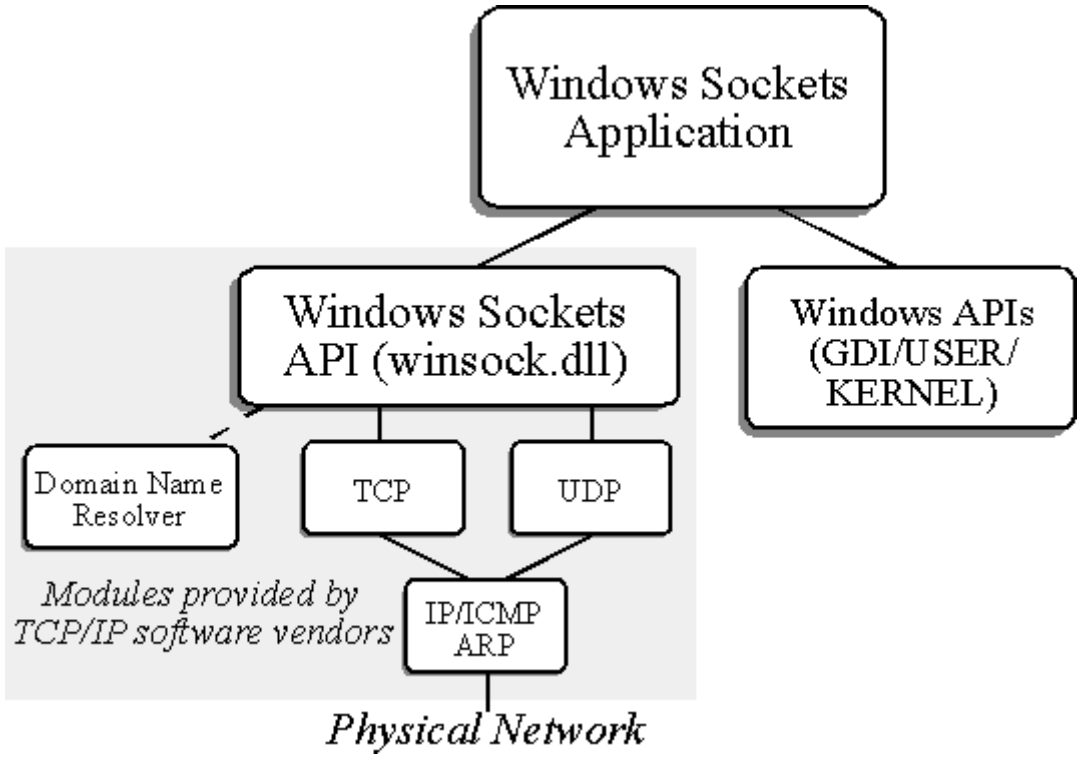
David Treadwell has been active in Microsoft Windows Sockets since its inception, acting as a coauthor of the specification and developing the Windows NT operating system/Windows Sockets interface.

Introduction

In today's heterogeneous networks, one thing is clear: closed, proprietary standards are unwelcome. As the business of networking computers has evolved over the last two decades, hardware and software providers have learned that cooperation can yield very rewarding results. The Microsoft® Windows® Sockets effort represents an extremely usable open networking standard developed by over 20 cooperating vendors in the networking community.

The Windows Sockets API (Application Programming Interface) provides applications with an abstraction of the networking software below it. The present Windows Sockets specification, version 1.1, defines this abstraction for the TCP/IP, or Internet protocol family. For many, the TCP/IP protocols represent the greatest common denominator between the many distinct systems that make up today's enterprise networks. In fact, the TCP/IP protocols were developed in a manner similar to the Windows Sockets specification: open cooperation between many interested parties with different requirements. Windows Sockets doesn't stop at TCP/IP, however. The level of abstraction is complete enough to support other protocol families as well, for example: the Xerox® Network System (XNS) protocols, Digital's DECnet protocol, or Novell®'s IPX/SPX family. For many, the attraction to Windows Sockets is the ability to develop and/or run a Windows Sockets-compatible application over any vendor's Windows Sockets-compliant TCP/IP implementation, while providing the ability to move the application easily to other networking protocols.

Windows Sockets is implemented as a DLL (dynamic link library) provided by the vendor of the given network protocol software. A Windows Sockets developer leverages the APIs from both Windows Sockets as well as the Windows operating system itself to create a network-aware Windows application. The following diagram illustrates the basic building blocks used to create a Windows Sockets application. The areas shaded in gray are provided by the network protocol vendor.



The basic building blocks of a Windows Sockets application

The goal of this article is to offer developers a taste of this powerful new API. We assume that the reader is familiar with both networking basics as well as Windows programming. To keep things simple, we will focus our discussion on using Windows Sockets to develop network applications over the TCP/IP protocol family, but will include several tips on how other transport protocols are supported with Windows Sockets.

Today, over 30 application vendors have announced the development of commercial applications to Windows Sockets such as X-Windows servers, terminal emulators, and email systems. Several commercial and public domain versions of Windows Sockets-compliant TCP/IP stacks are available, and both Microsoft Windows NT™ and Chicago include Windows Sockets implementations which are capable of supporting transport protocols in additions to TCP/IP, such as IPX/SPX. Many corporate developers are standardizing on Windows Sockets for heterogeneous client-server application development under Microsoft Windows.

Whether you're developing client-server, peer-to-peer, or distributed network applications, Windows Sockets represents a standard networking API for Microsoft Windows which will allow you to develop flexible networking applications for TCP/IP and other networking protocols.

The Birth of Windows Sockets...

The Windows Sockets specification is the result of a cooperative effort among over 20 vendors in the TCP/IP community. The charter of the group was simple: *to design a binary-compatible API for the TCP/IP protocol family under Microsoft Windows, allowing for future support of additional transport protocols.* The effort, led by Martin Hall of JSB Corporation, was kicked off at a Birds of a Feather session at the Fall '91 Interop networking conference.

Infuriated by a lack of standardization, TCP/IP application vendors like JSB were forced to develop their applications to be aware of several divergent APIs. This allowed their applications to run over multiple vendors' TCP/IP implementation, making their products available to the widest possible audience. With over 10 different TCP/IP implementations on the market, many vendors created an *abstraction layer* to the network interface, creating a common denominator which could be supported by all of their target implementations. Their application was then developed to this proprietary abstraction layer. *Providers,* or code which glued the application to a specific vendor's TCP/IP implementation, were developed for each of the TCP/IP implementations which the application desired to support.

This approach was both costly and frustrating. Application vendors were continuously updating their provider modules as TCP/IP implementors modified or updated their libraries. Moreover, new implementations were springing up quickly, and it took time before the appropriate provider could be made available to customers. Application vendors found it difficult to maintain, test, and support the multiple providers. This caused TCP/IP implementors difficulty as well, especially if a critical third-party application didn't run over their implementation. Customers were forced to choose a TCP/IP implementation based on their application needs rather than the merit of vendors' transports.

It would appear that getting the developers of TCP/IP transports and applications to work this out would make a lot of sense. Martin Hall acted as the catalyst to get things going quickly. In fact, vendors were so motivated to straighten out the TCP/IP networking API confusion that in just nine months the Windows Sockets committee published the first version of the specification. The first anniversary of the effort was christened by several technology and interoperability demonstrations at Fall Interop '92. The message was clear: Windows Sockets was *real*.

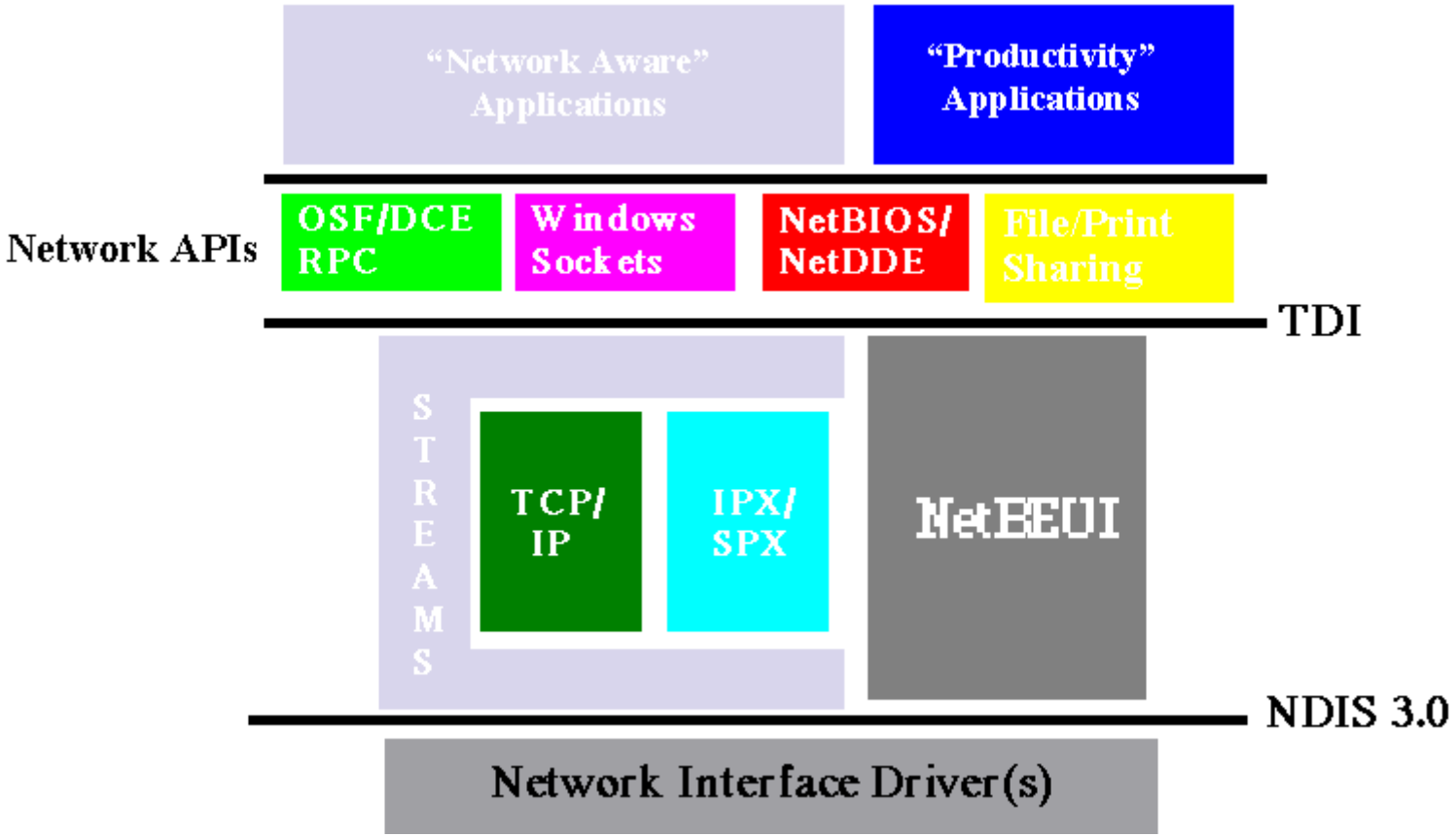
Windows Sockets Architectures

How are the Windows NT and Chicago operating systems able to support several transport protocols with a single API set? The answer lies in the carefully designed network architectures of these operating systems. Because an understanding of the underlying network architecture is always helpful and frequently mandatory in designing and implementing fast, robust networking applications, we'll briefly describe how Windows Sockets fits into these operating systems.

Windows Sockets in Windows NT

The key to transport-independent Windows Sockets support in Windows NT is a common kernel-mode transport interface called *Transport Device Interface*, or TDI for short. All of the networking components of Windows NT go through TDI to access a transport protocol's services.

TDI abstracts key differences between protocols, such as the format of transport addresses, and provides common entry points for typical transport features like sending data.



Networking architecture of Windows NT

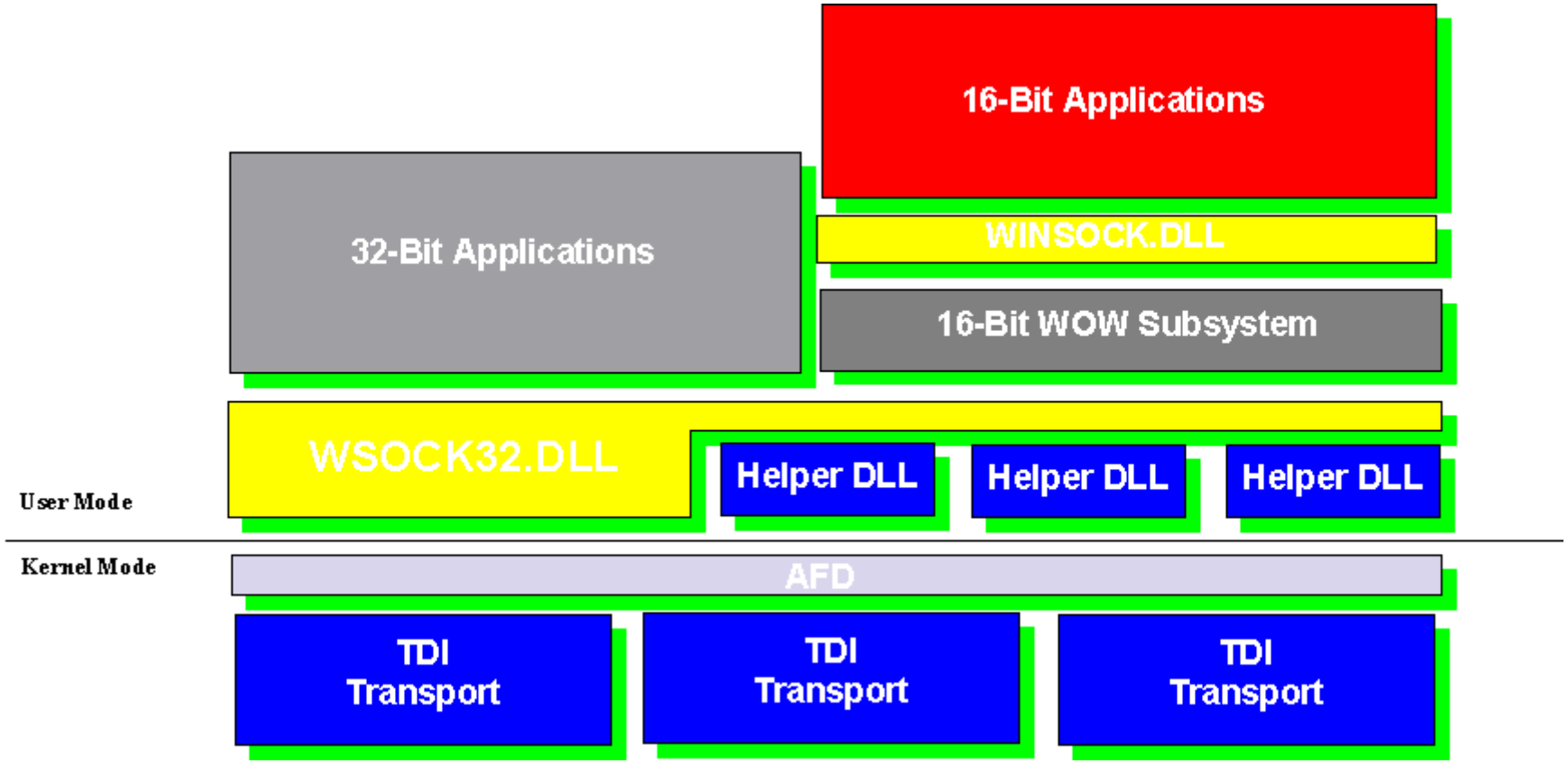
All the kernel-mode networking components of Windows NT use the TDI interface to speak to the transport layers below them. This use of a common interface allows easy addition or removal of transport protocols, since each transport protocol is completely separated from the layers which use it; in fact, each transport is packaged as a separate driver file.

However, TDI is not the only kernel-mode transport interface available in Windows NT. There is also the *Streams* interface, which is based on the AT&T® SVR4 Streams environment. Streams is useful for porting existing transport protocols to Windows NT quickly and easily. However, it does impose a performance overhead on all transactions, since there is a mapping layer between the TDI calls made by upper layers and the internal interfaces used by Streams. The TCP/IP and IPX/SPX transport protocols supplied with the original release of Windows NT existed in

the Streams environment, but Daytona, the next release of Windows NT, will include native TDI implementations of these transport protocols for improved performance.

The use of TDI as the interface underneath Windows Sockets solves most of the issues with multiple transport support, but some additional issues do remain. For example, because each transport protocol uses a different address format, how could Windows Sockets know which addresses are broadcast addresses? How can the Windows Sockets DLL know which transport device name corresponds to a given type of socket? And how can transports supply their own unique socket options like TCP/IP's SO_DONTROUTE option?

The answer is the use of user-mode "helper DLLs" which the Windows Sockets DLL (WSOCK32.DLL) uses for carefully defined functionality. Each of the TDI transport protocols exposed through Windows Sockets supplies one of these helper DLLs as well as placing information in Windows NT's registry about where to find the helper DLLs and what sorts of sockets each supports. The Windows Sockets DLL then calls into the helper DLL's entry points to learn about the format of transport addresses, to process transport-specific socket options, and more.



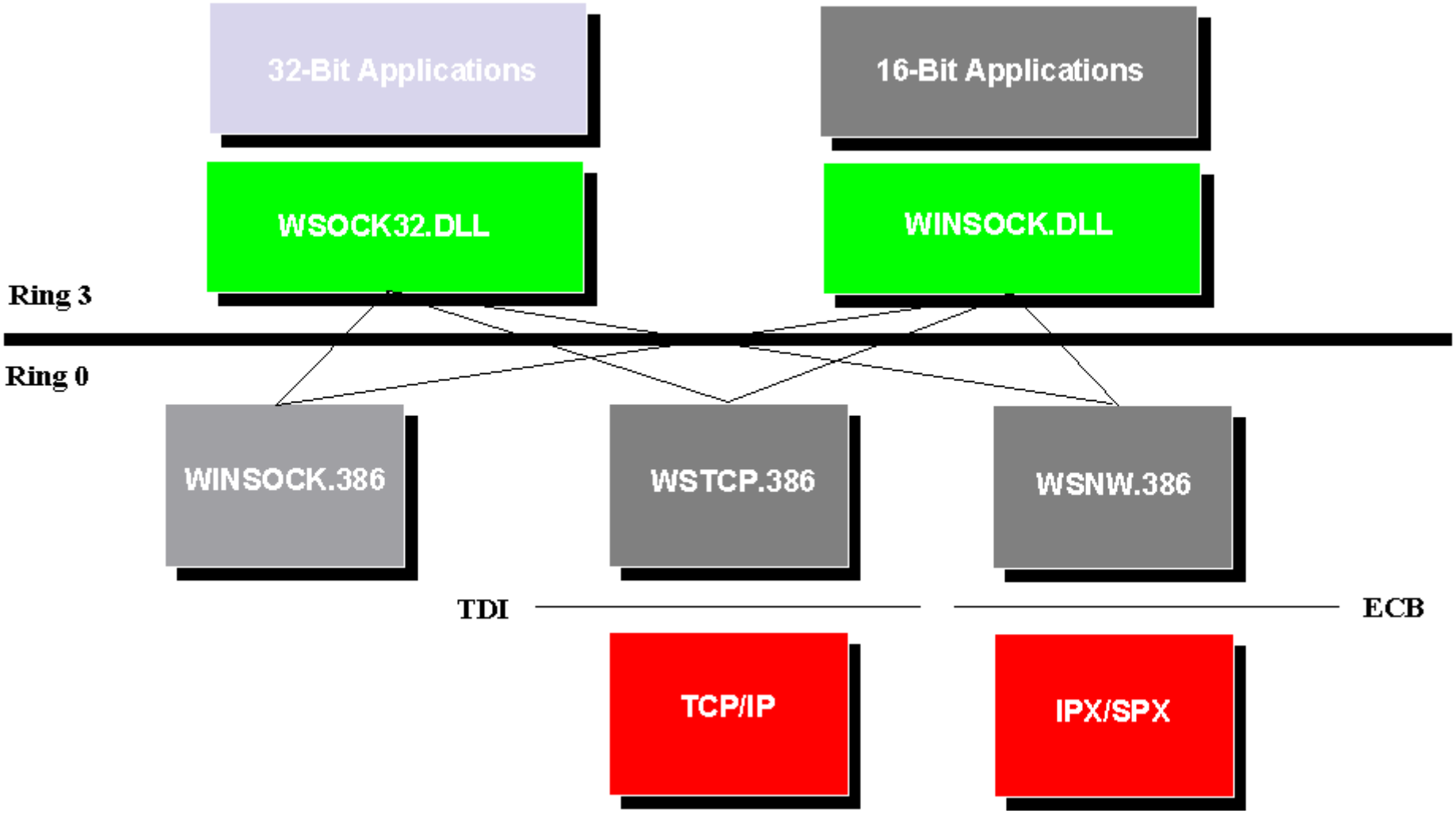
Windows Sockets architecture of Windows NT

In addition to 32-bit Windows Sockets applications supported through WSOCK32.DLL, Windows NT also supports 16-bit Windows Sockets applications through the file WINSOCK.DLL. This file is composed of minimal entry points which call into Windows NT's "WOW" (Windows-on-Windows) subsystem, which widens the parameters to 32-bits and calls into WSOCK32.DLL for the actual networking support. Thus, WINSOCK.DLL acts as a "thunking layer" between 16-bit applications and the rest of the operating system, which is all 32-bit.

Windows Sockets in Chicago

Note The information in this section pertains to the next major release of Windows, code-named Chicago. All of the information in this section is preliminary and subject to change.

Chicago also supports multiple transports under Windows Sockets, using the same names, WSOCK.DLL and WINSOCK.DLL, for the system DLLs so that application binary compatibility is preserved. However, internally the architecture of Windows Sockets in Chicago is significantly different than Windows NT's architecture.



Windows Sockets architecture of Chicago

One of the key differences between the Chicago and Windows NT Windows Sockets architectures is that there is no thunking involved for 16-bit applications. The Windows Sockets DLL is cross-compiled into two different files, WSOCK32.DLL and WINSOCK.DLL. Each of these speaks directly to the underlying VxDs (virtual device drivers) which provide Windows Sockets support, thereby eliminating an extra layer for 16-bit applications.

Next, Chicago has multiple kernel-mode transport interfaces, including both TDI and the ECB (event control block) interface. Therefore, the drivers which translate between Windows Sockets calls and the transport drivers are different for each transport. The WINSOCK.386 VxD points the user-mode DLLs at the appropriate kernel driver, then the DLLs access WSTCP.386 or WSNW.386, which in turn speak to their transports over each transport's interface.

The Sockets Paradigm

The Sockets paradigm was first introduced in Berkeley UNIX® (BSD) in the early 1980s. Initially designed as a local IPC (inter-process communication) mechanism, sockets evolved into a network IPC mechanism for the built-in TCP/IP protocol family. A *socket* simply defines a bidirectional endpoint for communication between processes. Bidirectional simply implies that Windows Sockets allow applications to transmit as well as receive data through these connections.

A socket has three primary components: the interface to which it is bound (specified by an IP address), the port number, or ID to which it will be sending or receiving data, and the type of socket (either stream or datagram). Typically, a server application *listens* on a well-known port over all installed network interfaces. On the other hand, a client generally initiates communication from a specific interface from any port that the system has available. The type of the socket (stream or datagram) depends entirely on the needs of the application. Windows Sockets is closely related to the Berkeley sockets model; many of the APIs are identical, or very close. In addition to the Berkeley-style functions, Windows Sockets offers a class of asynchronous extensions which facilitate the development of more "Windows-friendly" applications. These extensions will be discussed in detail later in this article.

Stream vs. Datagram Sockets

The Windows Sockets model offers service for both connection-oriented and connectionless protocols. In the TCP/IP protocol family, TCP provides a connection-oriented service, whereas UDP (user datagram protocol) offers connectionless service. In the Windows Sockets model, connection-oriented service is offered by *stream* sockets, connectionless service is provided by *datagram* sockets.

TCP is a reliable, connection-oriented protocol, used by applications which either plan to exchange large amounts of data at a time, or by applications which require reliability and sequencing. For example, FTP (file transfer protocol), a protocol which facilitates the binary or ASCII transfer of arbitrarily large files, represents an application written to TCP or stream sockets. In contrast, if an application is willing to manage its own sequencing or reliability, or is using the network for low-bandwidth iterative processing, UDP is often used. An application which keeps system clocks synchronized by periodically broadcasting its system time would probably be written to use UDP.

Network-byte Order

Since Windows Sockets applications can't possibly be aware of what type of remote computer system that they will be dealing with *a priori*, it is necessary to define a common data representation model for vital information. Windows Sockets chose the big-endian model for the "on-the-wire" data representation, known as *network byte order*.

The Windows Sockets interface offers APIs to application programmers to do the necessary conversion between the local system representation (or host byte order) and network byte order. There is no harm in using these routines on systems which store data in big-endian natively; in fact, it's encouraged. By religiously using the byte-ordering APIs, your code can be used on systems with different internal representations without inheriting byte-ordering problems, thereby making your code more portable.

A Guided Tour of the Windows Sockets API

Although the Windows Sockets specification defines all of the Windows Sockets functions and structures, this guided tour of the API will give you a basic understanding of the building blocks of a Windows Sockets application. Following the walkthrough, we will discuss the use of Windows Sockets by the WormHole sample application.

The Basic Structures

Although the Windows Sockets specification contains about a dozen different structures, application developers will quickly become familiar with a few that are required by nearly all Windows Sockets applications.

```
struct sockaddr {
        u_short sa_family;
        char    sa_data[14];
};
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

The *sockaddr* structure is used by Windows Sockets to specify a local or remote endpoint address to which to connect a socket. An endpoint address simply contains the appropriate information to send data between two sockets on different systems. As the contents of endpoint addresses differ between network protocol families, the *sockaddr* structure was designed to accommodate endpoint addresses of variable size, satisfying requirements of many common network protocol families. The first field of a *sockaddr* contains the family number identifying the format of the remaining part of the address.

In the Internet address family, the *sockaddr_in* structure is used to store the endpoint address information and is cast to type *sockaddr* for the functions which require it. Other address families must define their own *sockaddr_* structures as appropriate for their needs. For TCP/IP, the *sockaddr_in* structure breaks the endpoint address into its two components: port ID (sin_port) and IP address (sin_addr), and pads the remaining eight bytes of the endpoint address with a character string (sin_zero). The port and IP address values are always specified in network byte order. The value for *sin_family* under TCP/IP is always *AF_INET* (address family Internet).

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

The *hostent* structure is generally used by the Windows Sockets database routines to return *host*, or system, information about a specified system on the network. The host structure contains the primary name for a system including optional aliases for the primary name. Additionally, it contains a list of address(es) for the specified system. This information is generally sought for the remote system to which an application is connecting using the Windows Sockets database routines described later.

```
struct  protoent {
        char FAR *            p_name;
        char FAR * FAR *        p_aliases;
        short                 p_proto;
};

struct  servent {
        char FAR *            s_name;
        char FAR * FAR *        s_aliases;
        short               s_port;
        char FAR *            s_proto;
};
```

The *protoent* and *servent* structures are also filled by the Windows Sockets database routines. These structures contain information about a particular protocol (TCP or UDP) or service (finger or telnet, for example) respectively. Along with the primary name and an array of aliases for the protocol or service, these structures also contain their corresponding 16-bit IDs, necessary to build a valid TCP/IP endpoint address.

```
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;
```

Finally, the *WSAData* structure is filled in by a Windows Sockets DLL when an application calls the *WSAStartup()* API. Along with Windows

Sockets version information, the structure also contains vendor-specific information, such as the maximum number of sockets available and the maximum datagram size. The *szDescription* and *szSystemStatus* members can be used by an implementation to identify itself and the current status of the DLL. For example, an implementation may return the text "Joe's ShareWare Windows Sockets implementation v1.2. 10/22/92" in *szDescription.* The specification of the *lpVendorInfo* member is completely up to an implementor and is not defined in the Windows Sockets specification.

Setting Up, and Cleaning Up Your Windows Sockets Application

As mentioned earlier, Windows Sockets offers some extensions to the Berkeley sockets paradigm to allow your application to be more "friendly" in the Windows Environment. All such functions are preceded by the characters "WSA", which is short for "Windows Sockets API." Although the use of WSA functions is strongly advised, there are two WSA functions that your application can't avoid: *WSAStartup()* and *WSACleanup()*.

WSAStartup() "attaches" your application to Windows Sockets and causes the Windows Sockets DLL to initialize any structures that it might need for operation. Additionally, *WSAStartup()* performs version negotiation and forces an internal Windows Sockets reference count to be incremented. This reference count allows Windows Sockets to maintain the number of applications on the local system requiring Windows Sockets services and structures. The version negotiation allows an application to determine whether or not the underlying Windows Sockets implementation is able to support the same version of the Windows Sockets specification that the application is written to. A Windows Sockets implementation may or may not support multiple versions of the specification. Other Windows Sockets-specific information may also be filled in such as the vendor of the implementation, the maximum datagram size supported, maximum number of sockets which an application can open, and more.

The following startup code is authored to run only under version 1.1 (the most current version of the Windows Sockets specification) or later, and requires that at least six sockets be available to the calling application.

```
#define    WS_VERSION_REQD    0x0101
#define     WS_VERSION_MAJOR    HIBYTE(WS_VERSION_REQD)
#define     WS_VERSION_MINOR    LOBYTE(WS_VERSION_REQD)
#define    MIN_SOCKETS_REQD    6

WSADATA    wsaData;
char     buf[MAX_BUF_LEN];
int        error;

.
.
.
error=WSAStartup(WS_VERSION_REQUIRED,&wsaData);
if (error !=0 ) {
    /* Report that Windows Sockets did not respond to the WSAStartup() call */
```

```
    sprintf(buf,"WINSOCK.DLL not responding.");
    MessageBox (hWnd, buf. "Windows Sockets Error",MB_OK);
    shutdown_app();
}

if (( LOBYTE (wsaData.wVersion) < WS_VERSION_MAJOR) ||
    ( LOBYTE (wsaData.wVersion) == WS_VERSION_MAJOR &&
      HIBYTE (wsaData.wVersion) < WS_VERSION_MINOR)) {

    /* Report that the application requires Windows Sockets version WS_VERSION_REQD */
    /* compliance and that the WINSOCK.DLL on the system does not support it.       */

    sprintf(buf,"Windows Sockets version %d.%d not supported by WINSOCK.DLL",
         LOBYTE (wsaData.wVersion), HIBYTE (wsaData.wVersion));
    MessageBox (hWnd, buf. "Windows Sockets Error",MB_OK);
    shutdown_app();
}

if (wsaData.iMaxSockets < MIN_SOCKETS_REQUIRED ) {

    /* Report that WINSOCK.DLL was unable to support the minimum number of  */
     /* sockets (MIN_SOCKETS_REQD) for the application                      */

    sprintf(buf,"This application requires a minimum of %d supported sockets.",
         MIN_SOCKETS_REQUIRED);
    MessageBox (hWnd, buf. "Windows Sockets Error",MB_OK);
    shutdown_app();
}
.
```

A Windows Sockets application generally calls*WSACleanup()* during its own cleanup, decrementing the internal reference count and letting Windows Sockets know that it's no longer needed by the calling application. Whatever cleanup this function forces is implementation-specific and shielded from the application. The application author should, however, check for any possible error conditions from *WSACleanup()* and report them before exiting, as this information might indicate a network layer problem in the system.

Error Handling

In order to provide a consistent mechanism for reporting errors and to ensure safety of Windows Sockets applications in multithreaded versions of Windows (like Windows NT), the *WSAGetLastError()* API was introduced as a means to get the code for the last network error on a particular thread. Under Windows 3.x, thread safety is not an issue, although *WSAGetLastError()* is still the appropriate way to check for extended error codes. Many functions in the Windows Sockets API set return an error code in the event that there was a problem, and rely on the application to call *WSAGetLastError()* to get more detailed information on the failure. The following code illustrates how an application might report an error to a user:

```
LPHOSTENT        host_info;
char         user_buf[MAX_BUF],    appl_buf[MAX_BUF];
.
.
.
/* Attempt to resolve hostname specified by  user_buf,return meaningful */
/* message to the user in the event of an error. */

host_info=gethostbyname(user_buf);
if(host_info==NULL){
    sprintf(buf,"Windows Sockets error %d: Hostname: %s couldn't be resolved.",
         WSAGetLastError(),user_buf);
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);}
.
.
.
```

In addition to the *WSAGetLastError()* API, an application may choose to use the *WSASetLastError()* API to set a network error condition which will be returned by a subsequent *WSAGetLastError()* call. Obviously, any Windows Sockets calls made between a *WSASetLastError()* and *WSAGetLastError()* pair will override the code set by the *WSASetLastError()* call.

Database Routines

The TCP/IP protocol relies on the binary representations for addresses and various other identifiers. However, end users and programmers prefer to use easy-to-remember names (such as "ftp rhino.microsoft.com"). It is therefore necessary to provide a common method to resolve both services and hostnames into their respective binary equivalents. To solve this, the Windows Sockets specification offers a set of APIs known as the *database routines.*

The database routines fall into three categories:

host resolution	Learning the IP address for a host based on system, or host name
protocol resolution	Learning the protocol ID of a specific member of a protocol family (TCP, for example)
service resolution	Learning the port ID of a service based on a service name/protocol pair

All the database routines return information in structures defined in the previous section.

Applications use the *gethostbyname()* and *gethostbyaddr()* functions to learn about the names and IP address(es) of a particular system, knowing only the name or the address of the system. Both calls return a pointer to a *hostent* structure as defined in the previous section. The

gethostbyname() call simply accepts a pointer to a null-terminated string representing the name of the system to resolve. The *gethostbyaddr()* instead accepts three parameters: a pointer to the address (in network byte order), the length of the address, and the type of address.

Generally the hostname or IP address is offered to the application by the user to specify a remote system to connect to, and the IP address is resolved by Windows Sockets by either parsing a local *hosts* file, or querying a DNS (domain name system) server. The details of the resolution however, are specific to the implementation, abstracted from the application by these APIs.

The *getservbyname()* and *getservbyport()* functions return information about well-known Windows Sockets services, or applications. Each of these system calls return a pointer to a *servent* structure, as defined in the previous section. Typically an application will use these calls to determine the port ID for a well-known service (such as FTP) to create an endpoint address.

The following code fragment demonstrates the use of the *getservbyname()* function to fill in the *sockaddr_in* structure which will be used to connect a socket to a well-known port (the FTP protocol port over TCP):

```
char            buf [MAX_BUF_LEN];
struct sockaddr_in        srv_addr;
LPSERVENT            srv_info;
LPHOSTENT            host_info;
SOCKET           s;
.
.
.
/* Get FTP service port information */

srv_info=getservbyname("ftp","tcp");

if (srv_info== NULL) {
    /* Couldn't find an entry for "ftp" over "tcp" */

    sprintf(buf,"Windows Sockets error %d: Couldn't resolve FTP service port.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

/* Set up socket */

srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = INADDR_ANY;
srv_addr.sin_port=srv_info->s_port;
```

The example uses *getservbyname()* to resolve the port number of the FTP service over TCP. The port ID is used to construct the *sockaddr_in* structure (endpoint address) for future use by the application. As we mentioned before, the address family for TCP/IP is always assigned as

AF_INET. We use the *INADDR_ANY* macro to specify any local IP interface to accept incoming connections (more on this later).

To round out the database routines, *getprotobyname()* and *getprotobynumber()* fill in a *protoent* structure, sometimes used by applications to create a socket over a particular protocol (e.g., UDP or TCP). More often, however, an application will use the SOCK_DGRAM and SOCK_STREAM macros to create either datagram or stream sockets.

Data Manipulation Routines

There are several routines that convert values between network byte order and host (or local system) byte order. Windows Sockets offers byte-ordering routines for 16- and 32-bit values from both host byte order and network byte order. The *htons()* function takes a 16-bit value (a short), and converts it from host byte order to network byte order—hence the name htons (host to network short). The other byte-ordering functions available are *htonl()*, *ntohs()*, and *ntohl()*.

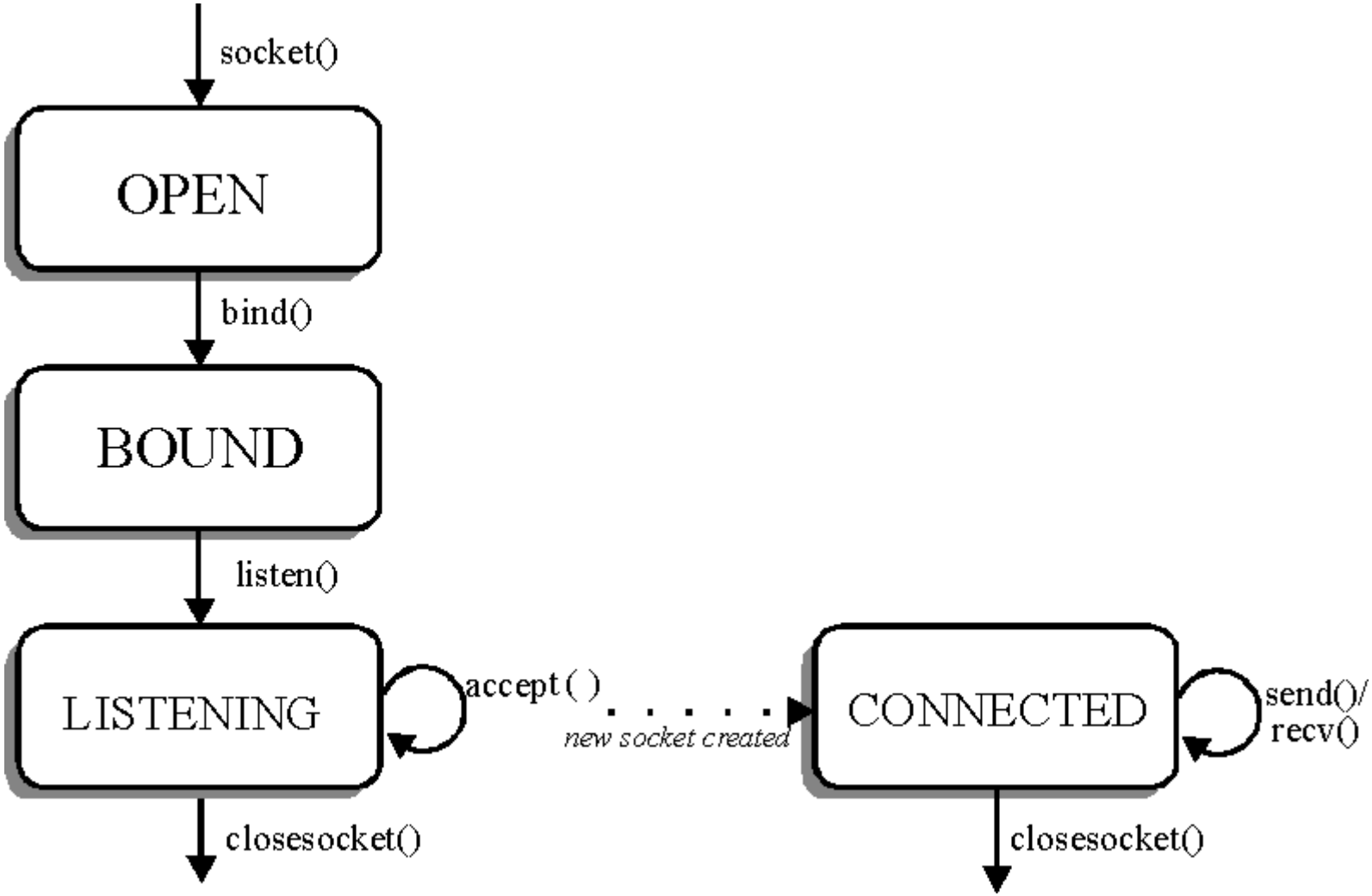
There are two other useful routines offered by Windows Sockets which convert IP addresses between strings and network byte-ordered 32-bit values. These functions are *inet_ntoa()* and *inet_addr()*. These routines are useful to convert the IP address of an endpoint user input. In the following example, a TCP-based server application uses the *inet_ntoa()* function to log incoming connection attempts:

```
SOCKET             cli_sock,    srv_sock;
LPSOCKADDR_IN        cli_addr;
char              *cli_ip,     buf[MAX_BUF];
int              len;
.
/* Accept incoming connection, create new local socket cli_sock */

cli_sock=accept(srv_sock,(LPSOCKADDR)&cli_addr,&len);

if (cli_sock==INVALID_SOCKET){
    return(ERROR);
}

/* Convert endpoint IP address from network byte order to ASCII */

cli_ip=inet_ntoa(cli_addr.sin_addr);
sprintf(buf,"Incoming connection request from: %s.\n",cli_ip);
log_event(buf);
```

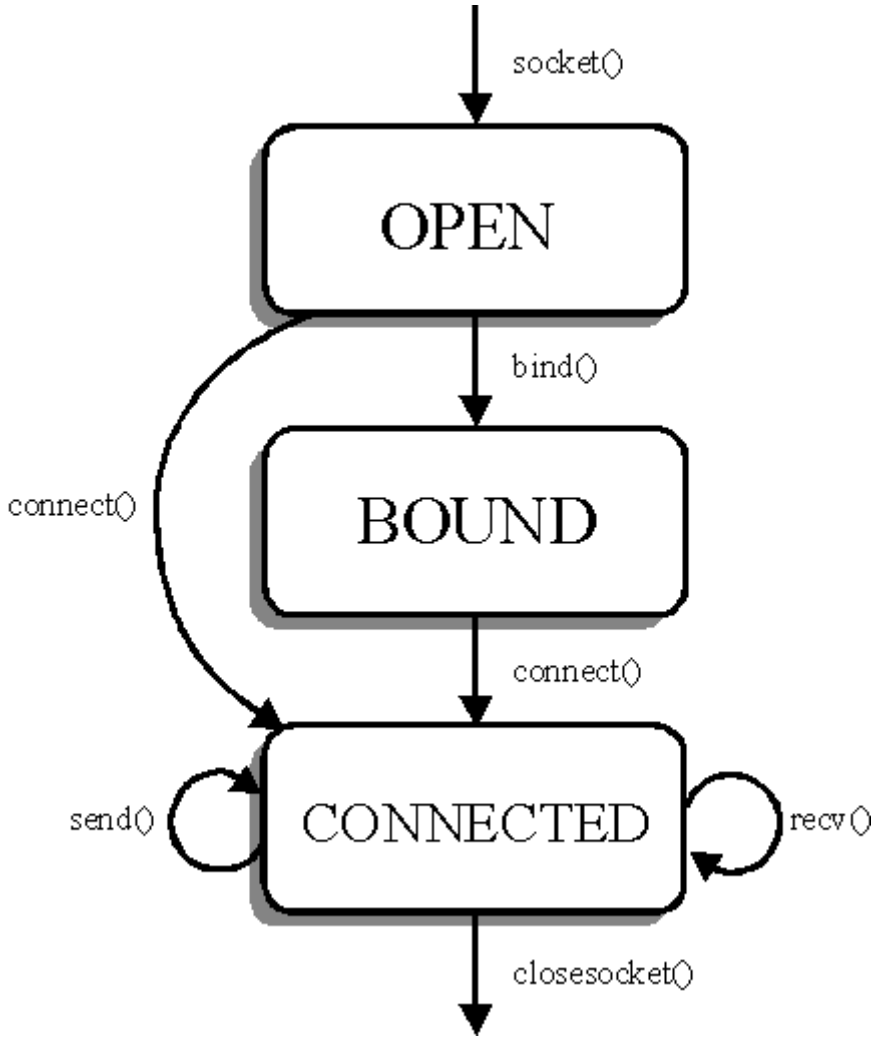
Setting Up Client and Server Sockets

Most Windows Sockets applications are asymmetrical; that is, there are generally two components to the network application—a client and a server. Frequently, these components are isolated into separate programs. Sometimes, these components are integrated into a single

application (such as our sample application). When both the client and server components of a networking application are integrated, the application is generally referred to as a "peer" application. Both the client and the server components go through different procedures to ready themselves for networking by making a number of Windows Sockets API calls. The following state diagrams illustrate the state transitions for setting up client- and server-side socket applications:



Setting up a server-side stream-based application



Setting up a client-side stream socket-based application

The *socket()* call creates an endpoint for both client- and server-side application communication. When calling *socket()*, the application specifies the protocol family and either the socket type (stream or datagram), or the specific protocol which it expects to use (for example, TCP). Both the client- and server-side of a network application use the *socket()* call to define their respective endpoints. *socket()* returns a *socket descriptor*, an integer which uniquely identifies the socket created within Windows Sockets.

Server-side connection setup

Once the socket is created, the server-side associates the freshly created socket descriptor and a *local endpoint address* via the *bind()* API. The local endpoint address is comprised of two pieces of data, the IP address and the port ID for the socket. The local IP address is used to determine which interfaces the server application will accept connection requests on; the port ID identifies the TCP or UDP port on which connections will be accepted. It is for these two values that the network byte-ordering routines (*htonl(), htons(),* etc.) were created. These values must always be represented in network byte order.

Alternatively, an application may substitute the value INADDR_ANY in place of a valid local IP address, and the system will accept incoming requests on any local interface, and will send requests on the "most appropriate" local interface. In fact, most server applications do exactly this. To associate a socket with any valid system port, provide a value of 0 for the *.sin_port* member of the *sockaddr_in* structure. This will select an unused system port between 1025 and 5000. As mentioned before, most server applications listen on a specified port, and client applications use this mechanism to obtain an unused local port. Once an application uses this mechanism to obtain a valid local port, it may call *getsockname()* to determine the port the system selected.

The *listen()* API sets up a connection queue. It accepts only two parameters, the socket descriptor and the queue length. The queue length identifies the number of outstanding connection requests that will be allowed to queue up on a particular port/address pair, before denying service to incoming connections.

The *accept()* API completes a stream-based server-side connection by accepting an incoming connection request, assigning a new socket to the connection, and returning the original socket to the *listen*ing state. The new socket is returned to the application, and the server can begin interacting with the client over the network.

Client-side connection setup

From the client's perspective, the application also creates a socket using the *socket()* call. The *bind()* command is used to bind the socket to a locally specified endpoint address which the server will use to transmit data back to the client. Once a local endpoint association is made, the *connect()* API establishes a connection with a remote endpoint. This routine initiates the network connection between the two systems. Once the connection is made, the client can begin interaction with the server on the network.

Although the client may choose to call **bind()**, it is not necessary to do so. Calling **connect()** with an unbound socket will simply force the system to choose an IP interface and unique port ID and mark the socket as bound. Most client-side applications neglect the **bind()** call as there are rarely specific requirements for a particular local interface/port ID pair.

The following code fragments create and connect a pair of stream-based sockets using the API flow outlined above:

Server-side (connection-oriented)

```
#define           SERVICE_PORT        5001

SOCKET            srv_sock,        cli_sock;
```

```
struct sockaddr_in        srv_addr,        cli_addr;
char             buf[MAX_BUF_LEN];

/* Create the server-side socket */

srv_sock=socket(AF_INET,SOCK_STREAM,0);
if (srv_sock==INVALID_SOCKET){
    sprintf(buf,"Windows Sockets error %d: Couldn't create socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

srv_addr.sin_family=AF_INET;
srv_addr.sin_addr.s_addr=INADDR_ANY;
srv_addr.sin_port=SERVICE_PORT;        /* specific port for server to listen on */

/* Bind socket to the appropriate port and interface (INADDR_ANY) */

if (bind(srv_sock,(LPSOCKADDR)&srv_addr,sizeof(srv_addr))==SOCKET_ERROR){
    sprintf(buf,"Windows Sockets error %d: Couldn't bind socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

/* Listen for incoming connections */

if (listen(srv_sock,1)==SOCKET_ERROR){

    sprintf(buf,"Windows Sockets error %d: Couldn't set up listen on socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

/* Accept and service incoming connection requests indefinitely */

for ( ; ; ) {
    cli_sock=accept(srv_sock,(LPSOCKADDR)&cli_addr,&addr_len);
    if (cli_sock==INVALID SOCKET){

        sprintf(buf,"Windows Sockets error %d: Couldn't accept incoming \
             connection on socket.",WSAGetLastError());
        MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
        shutdown_app();
```

```
    }
    .
    /* Client-server network interaction takes place here */
    .
    closesocket(cli_sock);
}
```

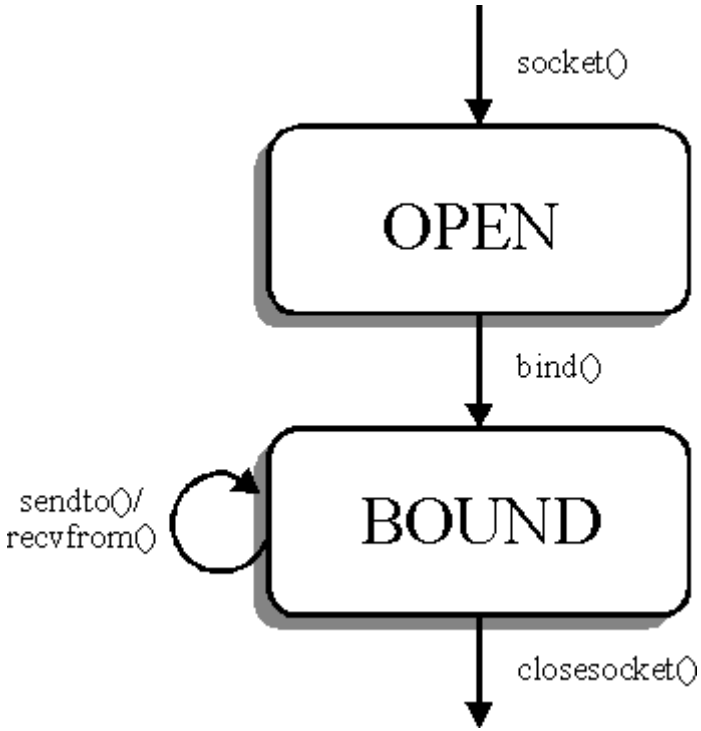
Client-side (connection-oriented)

```
/* Static IP address for remote server for example. In reality, this would be
   specified as a hostname or IP address by the user */

#define    SERVER        "131.107.1.121"

struct     sockaddr_in    srv_addr,cli_addr;
LPSERVENT            srv_info;
LPHOSTENT            host_info;
SOCKET            cli_sock;
.
/* Set up client socket */

cli_sock=socket(PF_INET,SOCK_STREAM,0);

if (cli_sock==INVALID_SOCKET){

    sprintf(buf,"Windows Sockets error %d: Couldn't create socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

cli_addr.sin_family=AF_INET;
cli_addr.sin_addr.s_addr=INADDR_ANY;
cli_addr.sin_port=0;                /* no specific port req'd */

/* Bind client socket to any local interface and port */

if (bind(cli_sock,(LPSOCKADDR)&cli_addr,sizeof(cli_addr))==SOCKET_ERROR){

    sprintf(buf,"Windows Sockets error %d: Couldn't bind socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}
```

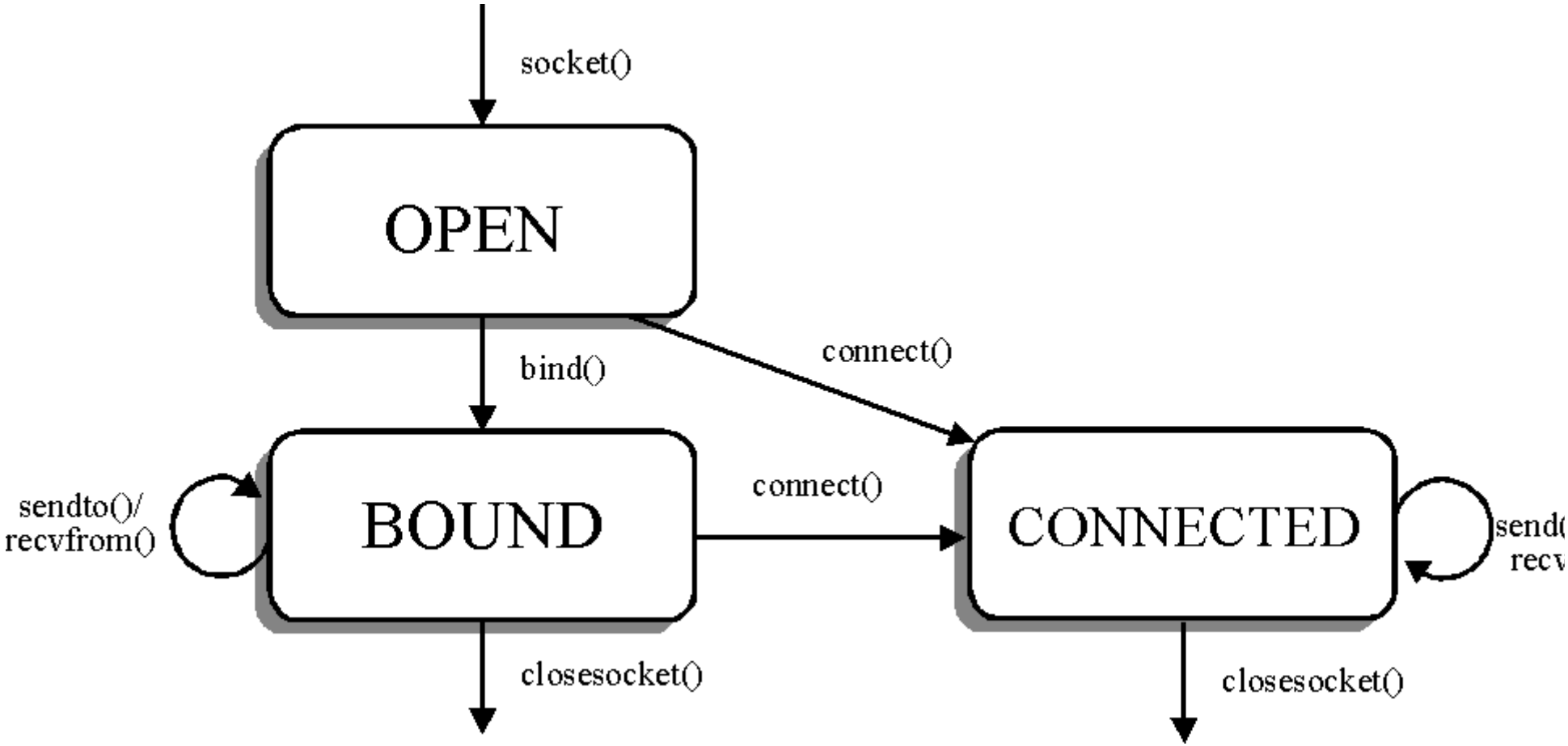
```
/* Get the remote port ID to connect to for FTP service */

srv_info=getservbyname("ftp","tcp");

if (srv_info== NULL) {

    sprintf(buf,"Windows Sockets error %d: Couldn't resolve FTP service port.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = inet_addr(SERVER);
srv_addr.sin_port=srv_info->s_port;

/* Connect to FTP server at address SERVER */

if (connect(cli_sock,(LPSOCKADDR)&srv_addr,sizeof(srv_addr))==SOCKET_ERROR){

    sprintf(buf,"Windows Sockets error %d: Couldn't connect socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}
/* Client-server network interaction takes place here */
```

Although the above fragment makes use of the **bind()** API, it would be just as effective to skip over this call as there are no specific local port ID requirements for this client. The only advantage that **bind()** offers is the accessibility of the port which the system chose via the **.sin_port** member of the **cli_addr** structure which will be set upon success of the **bind()** call.

For connectionless, or "datagram" sockets, Windows Sockets usage is a little simpler. Since the communication in datagram sockets is connectionless, it is not necessary to use the APIs necessary for creating a connection, namely: *connect()*, *listen()*, and *accept()*. The flow of Windows Sockets APIs that a typical connectionless client-server pair will generally traverse follows:



Setting up a server-side datagram-based application



Setting up a client-side stream-based application

As pictured above, a client may choose to *connect()* the datagram socket for convenience of multiple sends to the remote endpoint. Connecting a datagram socket will cause all sends to go to the connected address, and any datagrams received from a remote address different than the connected address are discarded by the system. Generally, connectionless clients use the *sendto()* API to transmit application data. The *sendto()* call requires that the destination's endpoint address be specified with every call to the API. By *connect*ing a datagram socket, a client sending a large amount of data to the same destination can simply use the *send()* API to transmit, without having to specify a remote endpoint with every call, and the client need not concern itself with the possibility of receiving unwanted data from other hosts. Depending on the type of application you are developing, and the Windows Sockets implementation below your application, *connect*ing datagram sockets may improve performance of your application.

Some sample code illustrates how the TFTP protocol (a connectionless protocol for file transfer) client and server might be implemented over Windows Sockets:

Server-side (connectionless)

```
SOCKET              srv_sock;
struct sockaddr_in        srv_addr;

.
.
/* Create server socket for connectionless service */

srv_sock=socket(PF_INET,SOCK_DGRAM,0);

if (srv_sock==INVALID_SOCKET){

    sprintf(buf,"Windows Sockets error %d: Couldn't create socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

/* Resolve TFTP service port to listen on */

srv_info=getservbyname("tftpd","udp");

if (srv_info== NULL) {

    sprintf(buf,"Windows Sockets error %d: Couldn't resolve TFTPd service port.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = INADDR_ANY;   /* Allow the server to accept connections                    /* over any inte
srv_addr.sin_port=srv_info->s_port;

/* Bind remote server's address and port */

if (bind(srv_sock,(LPSOCKADDR)&srv_addr,sizeof(srv_addr))==SOCKET_ERROR){

    sprintf(buf,"Windows Sockets error %d: Couldn't bind socket.",
         WSAGetLastError());
```

```
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

/* Client-server network interaction takes place here */
```

Client-side (connectionless)

```
/* Static IP address for remote server for example. In reality, this would be
   specified as a hostname or IP address by the user */

#define    SERVER        "131.107.1.121"

struct sockaddr_in        srv_addr,cli_addr;
LPSERVENT            srv_info;
LPHOSTENT            host_info;
SOCKET            cli_sock;

.
.
/* Create client-side datagram socket */

cli_sock=socket(PF_INET,SOCK_DGRAM);
if (cli_sock==INVALID_SOCKET){

    sprintf(buf,"Windows Sockets error %d: Couldn't create socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

cli_addr.sin_family=AF_INET;
cli_addr.sin_addr.s_addr=INADDR_ANY;
cli_addr.sin_port=0;                /* no specific local port req'd */

/* Bind local socket */
if (bind(cli_sock,(LPSOCKADDR)&cli_addr,sizeof(cli_addr))==SOCKET_ERROR){

    sprintf(buf,"Windows Sockets error %d: Couldn't bind socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}
```

```
/* Resolve port information for TFTP service */

srv_info=getservbyname("tftp","udp");
if (srv_info== NULL) {

    sprintf(buf,"Windows Sockets error %d: Couldn't resolve TFTP service port.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}

srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = inet_addr(SERVER);
srv_addr.sin_port=srv_info->s_port;

if (connect(cli_sock,(LPSOCKADDR)&srv_addr,sizeof(srv_addr))==SOCKET_ERROR){
    sprintf(buf,"Windows Sockets error %d: Couldn't connect socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}
/* Client-server network interaction takes place here */
```

Since a connectionless client (such as TFTP) will undoubtedly be doing successive transmissions with the server (especially during long transfers), we have chosen to connect the socket, allowing the use of the **send()** and **recv()** APIs rather than **sendto()** and **recvfrom()**. The use of **connect()** with datagram sockets is purely optional.

Sending and Receiving Data (stream sockets)

Once an application successfully establishes a socket connection, it is ready to start transferring data over the connection. With stream (TCP) sockets, data transfer is said to be "reliable," meaning that the application may assume that the underlying transport will ensure that the data gets to the remote host without duplication or corruption. When a connection is established on a stream socket, a the TCP transport creates a "virtual circuit" between the two machines. This circuit remains open until both applications decide that they are done sending data on the circuit (typically a "graceful" close), or until a network error occurs which causes the circuit to be terminated abnormally.

An application sends data using the *send()* API. This API takes a socket descriptor, a pointer to a buffer to send, the length of the buffer, and an integer which specifies flags which can modify the behavior of *send()*. To receive data, an application uses the *recv()* API, which takes a pointer to a buffer to fill with data, the length of the specified buffer, and a flags integer. *recv()* returns the number of bytes actually received, and *send()* returns the number of bytes actually sent. Note that applications should always check the return codes of *send()*and *recv()* for the number of bytes actually transferred, since it may be different from the number requested. Because of the stream-oriented nature of TCP sockets, there isn't necessarily a one-to-one correspondence between *send()* and *recv()* calls. For example, a client application may perform ten calls to *send()*, each for 100 bytes. The system may combine or "coalesce" these sends into a single network packet, so that if the server

application did a *recv()* with a buffer of 1000 bytes, it would get all the data at once.

Therefore, an application must not make any assumptions about how data will arrive. A server which expects to receive 1000 bytes should call *recv()*in a loop until it has received all of the data:

```
SOCKET         s;
int          bytes_received;
char          buffer[1000];
char          *buffer_ptr;
int          buffer_length;

.
.
buffer_ptr = buffer;
buffer_length = sizeof(buffer);

/* Receive all outstanding data on socket s */

do {
    bytes_received = recv(s, buffer_ptr, buffer_length, 0);
    if (bytes_received == SOCKET_ERROR) {
        sprintf(buf,"Windows Sockets error %d: Error while receiving data.",
             WSAGetLastError());
        MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
        shutdown_app();
    }
    buffer_ptr += bytes_received;
    buffer_length -= bytes_received;
} while (buffer_length > 0);
```

Likewise, an application which wants to send 1000 bytes should *send()* in a loop until all of the data has been sent:

```
SOCKET         s;
int          bytes_sent;
char          buffer[1000];
char          *buffer_ptr;
int          buffer_length;

buffer_ptr = buffer;
buffer_length = sizeof(buffer);

/* Enter send loop until all data in buffer is sent */
```

```
do {
    bytes_sent = send(s, buffer_ptr, buffer_length, 0);
    if (bytes_sent == SOCKET_ERROR) {
        sprintf(buf,"Windows Sockets error %d: Error while sending data.",
             WSAGetLastError());
        MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
        shutdown_app();
    }
    buffer_ptr += bytes_sent;
    buffer_length -= bytes_sent;
} while (buffer_length > 0);
```

Sending and Receiving Data (datagram sockets)

Data transmission on datagram sockets is significantly different from stream sockets. The most important difference is that transmission is not reliable on datagram sockets. This means that if an application attempts to send data to another application, the system does nothing to guarantee that the data will actually be delivered to the remote application. Reliability will tend to be good on LANs (local-area networks, such as a network connecting several computers within a single building), but can be very poor on WANs (wide-area networks, such as the Internet which connects hundreds of thousands of computers worldwide).

The next important difference in datagram sockets is that they are connectionless. This means that there is no default remote address assigned to them. For this reason, an application which wants to send data must specify the address to which the data is destined with the *sendto()* API. In addition, an application typically wants to know where data came from, so it receives data with the *recvfrom()* API, which returns the address of the sender of the data.

The final significant difference in data transmission for datagram sockets is that it is "message oriented." This means that there is a one-to-one correspondence between *sendto()* and *recvfrom()* calls, and that the system does not coalesce data when sending it. For example, if an application makes 10 calls to *sendto()* with a buffer of two bytes, the remote application will need to perform 10 calls to *recvfrom()* with a buffer of at least two bytes to receive all the data.

Below is a code fragment from a datagram sockets application which echos datagrams back to the sender.

```
SOCKET             s;
SOCKADDR_IN         remoteAddr;
int             remoteAddrLength = sizeof(remoteAddr);
BYTE             buffer[1024];
int             bytesReceived;

for ( ; ; ) {
    /* Receive a datagram on socket s */
```

```
    bytesReceived = recvfrom( s, buffer, sizeof(buffer),0,
                         (PSOCKADDR)&remoteAddr, &remoteAddrLength );

     /* Echo back to the server as long as bytes were received */

    if ( bytesReceived != SOCKET_ERROR ||
         sendto( s, buffer, bytesRecieved, 0,
                 (PSOCKADDR)&remoteAddr, remoteAddrLength ) ==
         SOCKET_ERROR ){
         sprintf(buf,"Windows Sockets error %d: Error while sending data.",
              WSAGetLastError());
         MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
         shutdown_appl();
     }
}
```

As mentioned earlier, it is possible to *connect()* a datagram socket. Once a datagram socket is connected, all data is sent and received from the remote address to which the datagram is connected, so it is possible to use the *send()* and *recv()* APIs on connected datagram sockets.

Terminating a Connection

An application has several options for terminating a connection. The simplest is to call the *closesocket()* API, which takes only a socket descriptor as input. This API frees resources associated with a socket and initiates the graceful close sequence. This sequence is completed when the remote application also closes its socket.

If an application determines that it is done sending data, but may want to receive more data, it can call the *shutdown()* API. This API notifies the remote end that the local application won't be sending any more data, but may continue to receive data.

Lastly, an application may cause an "abortive" or "hard" close on a connection with the SO_LINGER socket option in conjunction with *closesocket()*. Setting the linger timeout to 0 causes the circuit to be terminated immediately, regardless of whether the remote end has completed its data transfer, and any unreceived or unsent data is dropped. Therefore, this option should be used with caution and only if the results are understood and intended. The following code fragment demonstrates how to perform a hard close on a connection:

```
LINGER         lingerInfo;
INT         err;
SOCKET         s;

/* First set the linger timeout on the socket to 0.  This will */
/* cause the connection to be reset. */

lingerInfo.l_onoff = 1;
```

```
lingerInfo.l_linger = 0;

setsockopt( s, SOL_SOCKET, SO_LINGER, (char *)&lingerInfo,sizeof(lingerInfo) );
closesocket(s);
```

How can an application know that the remote end has terminated the connection? The answer depends on whether the remote end terminated the connection gracefully or abortively. If the termination was abortive, then *send()* and *recv()* calls will fail with the error WSAECONNRESET. This indicates to an application that data may have been lost, and the error condition should be reported to the user.

If the termination was graceful, then any *recv()* calls made after all data has been received will return the value zero as the number of bytes received. This indicates that the remote end has gracefully terminated its end of the connection and the local end may close the socket without fear of data loss. The following code fragment illustrates how an application initiates a graceful close and then waits for the remote end to close gracefully before closing the socket.

```
SOCKET         s;
INT         err;
BYTE         buffer[1024];

err = shutdown( s , 1 );
if ( err == SOCKET_ERROR ) {
    sprintf(buf,"Windows Sockets error %d: Error during shutdown.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_appl();
}

/* Receive the rest of the pending data */

while ( (err = recv( s, buffer, sizeof(buffer), 0 ) != 0 ) {

   if ( err == SOCKET_ERROR ) {
        sprintf(buf,"Windows Sockets error %d: Error while receiving data.",
             WSAGetLastError());
        MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
        shutdown_appl();
    }
    .
    .
    /* do something with the data we received. */
    .
    .

}
```

```
/* The other side has also terminated.  we can safely close now */

closesocket( s );
```

Note that in this example the application keeps calling *recv()* until it returns zero bytes received. If an application closes a socket and there is data available to be received, or data later arrives, then the system will abort the connection and throw out the data, since there is nobody to give the data to. Well-behaved applications should ensure that they receive all data before closing a socket.

Asynchronous Windows Sockets Calls

By default, an application's socket calls will block until the requested operation can be completed. For example, if an application wishes to receive data from another application, its call to the *recv()* API will not complete until the other application has sent data which can be returned to the calling application.

This model is sufficient for simple applications, but more sophisticated applications may not wish to block for an arbitrarily long period for a network event. In fact, in Windows 3.1, blocking operations are considered poor programming practice because applications are expected to call *PeekMessage()* or *GetMessage()* regularly in order to allow other applications to run and to receive user input.

To support the sophisticated applications which fit better within the Windows programming paradigm, Windows Sockets supports the concept of "nonblocking sockets." If an application sets a socket to nonblocking, then any operation which may block for an extended period will fail with the error code WSAEWOULDBLOCK. This error indicates to the application that the system was unable to perform the requested operation immediately.

How, then, does an application know when it can successfully perform certain operations? Polling would be one (poor) solution. The optimal mechanism is to use the asynchronous notification mechanism provided by the *WSAAsyncSelect()* API. This routine allows an application to notify a Windows Sockets implementation of certain events which are of interest, and to receive a Windows message when the events occur. For example, an application may indicate interest in data arrival with the FD_READ message, and when data arrives the Windows Sockets DLL posts a message to the application's window handle. The application receives this message in a *GetMessage()* or *PeekMessage()* call and can then perform the corresponding operation.

The following code fragment demonstrates how an application opens and connects a TCP socket and indicates that it is interested in being notified when one of three network events occurs:

- data arrives on the socket
- it is possible to send data on the socket
- the remote end has closed the socket

```
/* Static IP address for remote server for example. In reality, this would be
   specified as a hostname or IP address by the user */

#define    SERVER            "131.107.1.121"

#define     SOCKET_MESSAGE     WM_USER+1
#define     SERVER_PORT         4000

struct sockaddr_in        srv_addr;
SOCKET            cli_sock;

.
.
.
/* Create client-side socket */

cli_sock=socket(PF_INET,SOCK_STREAM,0);

if (cli_sock==INVALID_SOCKET){
    sprintf(buf, "Windows Sockets error %d: couldn't open socket.",
         WSAGetLastError());
    MessageBox(hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_appl();
}

srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = inet_addr(SERVER);
srv_addr.sin_port=SERVER_PORT;

/* Connect to server */

if (connect(cli_sock,(LPSOCKADDR)&srv_addr,sizeof(srv_addr))==SOCKET_ERROR){

    sprintf(buf,"Windows Sockets error %d: Couldn't connect socket.",
         WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_appl();
}

/* Set up async select on FD_READ, FD_WRITE, and FD_CLOSE events */

err = WSAAsyncSelect(cli_sock, hWnd, SOCKET_MESSAGE, FD_READ|FD_WRITE|FD_CLOSE);
if (err == SOCKET_ERROR) {
    sprintf(buf,"Windows Sockets error %d: WSAAsyncSelect failure.",
         WSAGetLastError());
```

```
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_appl();
}
```

When one of the specified network events occurs, the specified window handle *hWnd* receives a message containing a *wMsg* of SOCKET_MESSAGE. The *wParam* field of the message will have the socket handle, and *lParam* contains two pieces of information: the low word contains the event that occurred (FD_READ, FD_WRITE, or FD_CLOSE) and the high word contains an error code, or 0 if there was no error. Code similar to the following fragment, which would belong in an application's main window procedure, may be used to interpret a message from *WSAAsyncSelect()*:

```
long FAR PASCAL _export WndProc(HWND hWnd, UINT message, UINT wParam, LONG lParam)
{
    INT err;

    switch (message) {

    case ...:
        /* handle Windows messages */
    .
    .
    .
    case SOCKET_MESSAGE:

        /* A network event has occurred on our socket.
           Determine which network event occurred. */

        switch (WSAGETSELECTEVENT(lParam)) {
        case FD_READ:
            /* Data arrived.  Receive it. */
            err = recv(cli_sock, Buffer, BufferLength, 0);
            if (WSAGetLastError() == WSAEWOULDBLOCK) {
                /* We have already received the data. */
                break;
            }
            if (err == SOCKET_ERROR) {
                sprintf(buf, "Windows Sockets error %d: receive error.", WSAGetLastError());
                MessageBox(hWnd,buf,"Windows Sockets Error",MB_OK);
                shutdown_appl();
            }
            .
            .
            /* Do something useful with the data. */
            .
            .
```

```
            break;

        case FD_WRITE:
            /* We can send data. */

            err = send(cli_sock, Buffer, BufferLength, 0);
            if (err == SOCKET_ERROR) {
                if (WSAGetLastError() == WSAEWOULDBLOCK) {
                    /* Send buffers overflowed. */
                    break;
                }
                sprintf(buf, "Windows Sockets error %d: send failed.",
                     SAGetLastError());
                MessageBox(hWnd,buf,"Windows Sockets Error",MB_OK);
                shutdown_appl();
            }
            break;

        case FD_CLOSE:
            /* The remote closed the socket. */

            closesocket(cli_sock);
            break;
        }

        break;
    }
.
```

An important part of the behavior of *WSAAsyncSelect()* is that after the Windows Sockets DLL has posted a message for a particular network event, the DLL refrains from posting another message for that event until the application has called the "reenabling function" for that event. For example, once an FD_READ has been posted, no more FD_READ messages will be posted until the application calls *recv()*. This prevents an application's message queue from being overflowed with messages for a single network event.

Using *WSAAsyncSelect()* can simplify and improve organization in Windows applications by allowing them to be fully event-driven. Such an application responds to network events in much the same way it responds to user events such as a mouse click. In addition, applications which make use of WSAAsyncSelect() are better behaved Windows applications since they must frequently call *PeekMessae()* or *GetMessage()* in order to receive the network messages.

Asynchronous Database Routines

Just as simple network operations like *recv()* can block for extended periods of time, so can the database routines like *gethostbyname()*, especially is they go through DNS which can require considerable time due to the network activity involved. In order to allow well-behaved

Windows Applications to use the database routines, the Windows Sockets API supplies asynchronous versions of the database routines. Their use is similar to the synchronous database routines with two important exceptions:

- Completion of the operation is indicated via a Windows message, as with *WSAAsyncSelect().*
- The application is required to include a buffer which is filled in by the Windows Sockets DLL with the information requested by the application. This is in contrast to the synchronous database routines which return a pointer to space owned by the Windows Sockets DLL.

In order to make an asynchronous database call, an application passes in information about the call, a window handle to receive the completion message, a message code, and a buffer for the output information. The Windows Sockets API defines a constant, MAXGETHOSTSTRUCT, which is the maximum size of a hostent structure. An application should use a buffer of this size to pass to the asynchronous database routines in order to be guaranteed that all the output information will fit in its buffer. A call to *WSAAsyncGetHostByName()* may be as follows:

```
#define GETHOST_MESSAGE         WM_USER+2

BYTE                HostBuffer[MAXGETHOSTSTUCT];
HANDLE TaskHandle;

/* Resolve hostname "hostname" asynchronously */

TaskHandle = WSAAsyncGetHostByName(hWnd, GETHOST_MESSAGE, "hostname", HostBuffer,                                MAXGETHOSTS

if (TaskHandle == SOCKET_ERROR) {

    sprintf(buf, "Windows Sockets error %d: Hostname couldn't be resolved.", WSAGetLastError());
    MessageBox(hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_appl();
}
```

WSAAsyncGetHostByName() and the other asynchronous database routines return a "task handle" which uniquely refers to the operation in progress. This task handle allows an application which may have multiple outstanding asynchronous database routines to associate completion messages with the request, since the task handle is returned in the completion message as *wParam.*

To receive and process the completion message, an application will do the following in its window procedure:

```
long FAR PASCAL _export WndProc(HWND hWnd, UINT message, UINT wParam, LONG lParam)
{
    INT err;
```

```
    switch (message) {
    case ...:
        /* handle Windows messages */
    .
    .
    .

    case GETHOST_MESSAGE:

        /* An asynchronous database routine completed. */
        if (WSAGETASYNCERROR(lParam) != 0) {
            sprintf(buf, "Windows Sockets error %d: Hostname couldn't be resolved.", WSAGetLastError());
            MessageBox(hWnd,buf,"Windows Sockets Error",MB_OK);
            shutdown_appl();
        }
        .
        .
        /* HostBuffer now contains a host buffer, use info from it. */
        .
        .
}
```

Windows Sockets on Windows NT

Although the initial focus of Windows Sockets was for Windows 3.1 16-bit applications, Windows NT supports Windows Sockets as well. To run existing 16-bit Windows Sockets applications, Windows NT supplies WINSOCK.DLL. In addition, Windows NT offers 32-bit Windows Sockets support in the DLL called WSOCK32.DLL. In general, all of the Windows Sockets routines in the 32-bit DLL are identical to their 16-bit counterparts, although their parameters are widened to 32-bits.

The most significant difference in programming Windows Sockets applications for Windows NT is that Windows NT is a fully preemptive, multithreaded operating system. Therefore, if an application blocks on a Windows Sockets call, the rest of the system is not negatively impacted. In addition, it is feasible to write a multithreaded application which uses one thread to process user input and another to block on sockets calls. Such an application could use the blocking sockets calls and still be responsive to user input.

The asynchronous Windows Sockets calls are still advantageous in Windows NT. The most significant advantage is that they allow an application to be fully event-driven, fitting better within the Windows programming paradigm. In addition, 32-bit versions of Windows Sockets will soon be available for Win32s®. An application written to use the asynchronous routines can be easily ported to Win32s without the negative system impacts of blocking calls.

Transport Independence

We've focused on TCP/IP for most of this article, but we promised to describe how an application can use Windows Sockets other transport protocols in Windows NT and Chicago. We'll focus on the highlights and the key features that differentiate different transport protocols, and on the mechanisms for opening sockets for use with different transport protocols.

Although these are not explicitly addressed in the Windows Sockets specification, the fact that the fundamental job of a transport protocol is data transfer, along with the fact that Windows Sockets provides a rich interface for low-level data transfer in applications, make the interface to Windows Sockets over different transport protocols quite feasible. In fact, having a common, well-known API interface allows applications to be ported quickly between different transport protocols, and there are several successful examples of this.

Because of the architecture of the Windows Sockets components of Windows NT and Chicago, all supported transports are accessed through the same DLL, WSOCK32.DLL. In fact, a single application may simultaneously use sockets of different transport protocols.

An application uses the parameters to the **socket()** API to specify the protocol it desires. For example, a TCP socket is opened with

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

while an SPX socket is opened with

```
s_spx = socket( AF_IPX, SOCK_SEQPACKET, NSPROTO_SPX );
```

After opening a socket for a specific protocol, all actions performed on the socket go through the specified protocol. In the above example, if the application did a **listen()** on the s_tcp socket but not on the s_spx socket, it would receive incoming connects through TCP but not receive incoming connects through SPX.

Protocol Characteristics

An application which uses different transport protocols must be aware of the different characteristics of transport protocols. While all transport protocols share the same fundamental purpose of bidirectional data transfer, many details of the data transfer differ. Following is a list of key characteristics of transport protocols of which Windows Sockets applications need to be aware.

Addressing

The most obvious difference between different transport protocols is the way in which each uses transport addresses, or "sockaddrs" in Windows Sockets terms. Virtually all transport protocols expose different address formats to applications, the one exception being NetBIOS protocols which all use the same address format.

A number of Windows Sockets APIs take sockaddrs as input or output parameters, including **bind(), ccept(), sendto(), recvfrom(), getsockname(),** and **getpeername().** For all these APIs a sockaddr is really two arguments: a pointer to the sockaddfr structure itself and a

length integer which gives the number of bytes in the sockaddr. For many transport protocols the length of a sockaddr is always the same, but sockaddr lengths may differ between transport protocols. For example, a TCP/IP sockaddr has 16 bytes and an IPX/SPX sockaddr has 14 bytes.

The format of sockaddr structures used by a transport protocol is specified in the *af* or "address family" argument to the **socket()** API. All TCP/IP sockets (including both TCP and UDP sockets) are opened with the AF_INET address family, while IPX/SPX sockets are opened with AF_IPX. Other transport protocols use different address family values as defined in the header file *winsock.h*.

It is up to an application to fill in a sockaddr when **bind()**ing to a local address or **connect()**ing to a remote address, and also to interpret the sockaddrs returned from the **accept()**, **sendto(), recvfrom, getpeername(),** and **getsockname()** APIs.

Connection-Oriented Vs. Connectionless

In a connection-oriented transport, applications are required to establish a virtual circuit (sometimes abbreviated to Visual C) before data transfer can take place. Virtual circuit establishment is asymmetric: one side, the server, must make known to the transport its willingness to receive incoming connections via the **listen()** API. The other side, the client, initiates the circuit with the **connect()** API, and the server can obtain a socket for the circuit with the **accept()** API. Once the circuit is established, data transfer takes place with the **send()** and **recv()** APIs. There is protocol-level activity which results from this circuit establishment, and more protocol activity tears down the circuit when the sockets are closed. TCP and SPX are examples of connection-oriented transport protocols.

In a connectionless transport, there is no circuit establishment required for data transfer. An application only needs to open and bind a socket, after which it may use the **sendto()** and **recvfrom()** APIs to send and receive data. Of course, in order to specify the remote address for sending data or the address from which received data was sent, an application must specify a sockaddr to these routines. UDP and IPX are examples of connectionless transport protocols.

It is possible to use the **connect()** API on sockets opened for connectionless protocols. This is merely an application convenience, allowing the application to use the **send()** and **recv()** APIs, and this does not result in any protocol activity. If a socket is connected in this manner, it will only receive packets sent from the connected address; other packets destined for the socket are silently discarded.

Reliable vs. Unreliable Data Delivery

Most physical networks are inherently unreliable. They can drop data, corrupt it, deliver it out of order, and deliver multiple copies of it. Some transport protocols hide this unreliability from applications by using protocol-level mechanisms to ensure that data is delivered in the correct order, without loss, corruption, or duplication. Other transport protocols do not make these guarantees; they simply make a best effort to get the data to its intended recipient, but physical network failures can lead to data integrity problems.

Because they have less inherent overhead, unreliable protocols often provide better performance, especially for applications which send small amounts of data in a request-response (transaction) format. However, connectionless protocols impose the burden on the application of doing whatever reliability guarantees are necessary, and therefore lead to more complicated applications. Reliable protocols are usually simpler for the application at the expense of a higher cost in system resources and performance.

Typically, connection-oriented protocols are reliable, while connectionless protocols are unreliable. While this is true of all the transport protocols supported by Windows NT, it is not required that these always go hand-in-hand.

Orderly Release

For a connection-oriented transport, there are two ways to terminate a virtual circuit: orderly and abortive. In an orderly release, both sides get a chance to indicate that they have sent all the data they intend to send, and only when both sides are done is the actual circuit terminated. For such a release it is possible to have one side indicate that it is done sending with the **shutdown()** API while the other side continues sending. In an abortive termination of a virtual circuit, one side decides that it is time to terminate the circuit and unilaterally ends the connection. If the other side attempts to use the connected socket, the request fails.

By default, **closesocket()** also attempts an orderly release of the connection. However, under several circumstances, such as pending unreceived, outstanding **send()** calls, and more, a **closesocket()** will result in an abortive close.

All transport protocols support the concept of an abortive release, but not all protocols support orderly release. If an application attempts an orderly release on a transport which does not support orderly release, the connection is terminated abortively. TCP supports orderly release, while SPX supports only abortive release.

Stream-Oriented vs. Message-Oriented

Connection-oriented transport protocols can be either stream-oriented or message-oriented. In a message-oriented transport protocol, each block of data is sent and received as a single block of data. An individual message is never broken down when received, although a transport protocol may be forced to break down a message if it is larger than the physical network's maximum transmission unit (MTU). However, this is transparent to the application.

In a stream-oriented transport, data is sent and received as a continuous stream of data without any message boundaries. The transport protocol is free to coalesce data from discrete **send()** calls and deliver it to a single **recv()** call as well as to break down the data buffer given to a single **send()** call and deliver it to several **recv()** calls. An application has no control over how these breakdowns occur, and an application gets no notification of how data was sent. If an application needs to communicate in discrete messages, it is the responsibility of the application to do message framing, for example by proceeding message data by the length of the message on send and receiving until the specified length has arrived.

Some message-oriented protocols support stream semantics in addition to message semantics. If an application opens a socket with SOCK_STREAM on such a transport, then the transport ignores all message boundaries when transmitting and receiving data. It simply delivers data as it becomes available without regard to whether the data comprises a complete message. This feature is handy for maximizing application portability. Note, however, that stream-oriented transport protocols cannot implement message-oriented semantics due to lack of information.

Expedited Data

Some connection-oriented transport protocols support the concept of expedited data, also referred to as urgent data or out-of-band data. Data sent with the MSG_OOB flag in the **send()** API is sent as expedited data, and the transport protocol makes every effort to deliver the data as quickly as possible. Often the data is delivered out of order so that it gets to the other application as quickly as possible. Applications interested in maximum portability should avoid expedited data since it is not supported by all transport protocols.

Connect and Disconnect Data

A few transport protocols support connect and disconnect data, where a small amount of data is sent by each side when a virtual circuit is established or terminated. This data can be anything, but is often negotiation information about the version of software connecting, etc. Again, applications desiring protocol portability should avoid this feature since it is not universally supported.

Broadcasts

Most connectionless transport protocols support broadcast in the same fashion, where any bound socket can send a broadcast if the SO_BROADCAST option is set, and broadcasts sent to the appropriate local endpoint are received without any additional work on the part of the application. NetBIOS transports, however, handle broadcasts somewhat differently. In order to receive broadcasts, an application must bind to the NetBIOS broadcast address, which is an asterisk (*) followed by 15 spaces (ASCII character 0x20). This means two things: a socket must be specially bound to receive broadcasts, and applications can't depend on receiving broadcasts intended only for a specific application, since *all* NetBIOS broadcasts are delivered to this address. In other protocols such as UDP/IP and IPX, broadcasts are delivered to a socket only if the broadcast was sent to the same port to which the socket was bound.

Guidelines for Transport Independence

The Windows Sockets interfaces of Windows NT and Chicago are designed to fully support all of the transport protocol characteristics listed above. This allows transport-specific applications to fully utilize all the idiosyncrasies of a particular protocol. However, when writing transport-independent applications, it is important to minimize or, preferably, eliminate all uses of features that are not supported by all transport protocols.

Many of the resulting guidelines are fairly obvious:

- Don't send or attempt to receive expedited data. If you must be able to receive expedited data to be compatible with older software, use the SO_OOBINLINE socket option and treat expedited data as normal data in your data reception code.
- Never use connect or disconnect data. If you need to include information at the start or end of a transmission, put it in the normal data stream.

- If you control your application's on-the-wire data format, perform your own message framing for both stream and message protocols. Message framing means to make the length of the message self-describing, for example, by including the message length as the first two bytes of the message. The application can then read two bytes, know the total message size, and read the remainder of the message. This abstracts the distinction between message-oriented and stream-oriented transport protocols.
- Assume that the transport protocol doesn't support a graceful close. It is always possible to do an abortive close, but a graceful close is not always supported. Use an application-level protocol to ensure that all data has been correctly transmitted.
- Typically, both connectionless and connection-oriented transport protocols will be available on any given machine. Therefore, it is reasonable to use whichever is most convenient for your application, which will typically be connection-oriented since it frees the application from having to worry about reliable data delivery.

Performance and Windows Sockets

Because Windows Sockets is just a transport protocol *interface* and not a protocol itself, it is able to achieve a very high level of performance. Well-written and optimized Windows Sockets applications and services typically perform much better than applications written to Named Pipes, NetBIOS, or other communication mechanisms. However, because Windows Sockets is so "close to the metal" of the transport protocol, it is also possible to write Windows Sockets applications and services which perform very poorly.

Excellent performance does not come free. Developers must spend some time understanding the performance characteristics of their applications, then making modifications to enhance performance. A good network analyzer is usually essential for network performance work, as the analyzer will indicate where the bottlenecks are and how efficiently the application is using the physical network. Also important are good benchmarks which both evaluate the important performance cases and provide consistent, reproducible results.

Effect of the Network Card

On an Ethernet network, Windows NT 3.5 TCP/IP Windows Sockets application running on a 486/33 with a fast network card can saturate the wire (>1100KB/sec) using approximately 45% of the system CPU. However, with a slow network card, the same system and application may achieve only 500KB/sec while using 100% of the CPU.

Why the difference? Slower network cards, including all 8-bit and most 16-bit cards, cause the CPU to do additional buffer copies and put the CPU into numerous wait states where the CPU is just spinning, waiting for the card to do something. However, the fast network cards, which are typically 32-bit bus mastering EISA or PCI cards, do not impose these CPU costs, instead handling themselves many of the chores required to send and receive data.

Since a developer cannot control the network cards that will be in use by end users, it is usually best to do performance testing with faster network cards where the CPU or physical network, not the card itself , is the bottleneck. When the card is the bottleneck, it can mask

significant performance problems and large improvements in the CPU utilization of the application may have a negligible effect. With a fast card, however, the bottleneck will usually be the application itself, so performance problems and improvements show quickly.

Threading Models

For a client application, the chosen threading model typically does not have a significant effect on performance. However, services that handle tens or hundreds or remote users need to select carefully the threading model they use because a poor choice can lead to poor performance.

The simplest thread model is a single process with a single thread that handles all the connected clients, typically using the **select()** API to multiplex between the clients. A single loop calls **select()** repeatedly, calling **send()**, **recv(),** or **accept()** when the **select()** call indicates that an action can be performed on one of the sockets. While simple to implement, the performance of services using this model can suffer because every network I/O call passes through **select()** which incurs significant CPU overhead for each I/O. This is acceptable when CPU use is not an issue, but presents a problem when the service requires high performance.

The next most complicated model is a single thread for each client. Every time a client connects, the service calls **CreateThread()** to create a thread which handles the client for the duration of the connection. This model can achieve very high performance when the number of connected clients is small, but at around 40 clients the thread context switching and resource overhead begins to present a significant burden for the system.

A similar model to thread per client is process per client. In process per client, an entire new process is created using **CreateProcess()** for each client that connects. This is the model typically used in UNIX Daemons. This model is discouraged for Windows NT and Windows '95 because processes are much more expensive than threads, both in terms of resources used and in the CPU overhead of context switching.

The most efficient threading models use a *worker thread* pool. In this model, a pool of threads services all the connected clients. In worker thread models, the service will usually use overlapped I/O with the Win32® **WriteFile()** and **ReadFile()** APIs to facilitate multiplexing and minimize the number of threads required by the service. In Windows NT 3.5, I/O completion ports may be used for additional efficiency. While this threading model is the most efficient for large numbers of clients, it is also the most complex. Therefore, it should only be used where it is required to have high performance with large numbers of connected clients.

I/O Buffer Size

Every time an application asks to send or receive data, the system must undergo a certain amount of overhead to check the parameters and buffers specified to the call. These checks cost cycles, so it is best to do as few I/O calls as possible to send the necessary amount of data.

The easiest way to minimize the number of I/O calls is to give each call a big chunk of data. For example, if you need to send 128KB of data, don't do 128 calls to **send()** with 1K in each call. Instead, do just a couple of calls with 32K or 64K of data. This way you get the data transferred with fewer I/O calls.

However, it is best not to specify too much data to any one I/O call because the data must be locked down in physical memory in order to be

used. High performance applications and services use buffer sizes in the range of 8K to 64K to balance between the physical memory cost of the locked-down buffer and the CPU overhead of each I/O call. The best way to determine a specific number for your application is experimentation.

Another problem with small I/O is the effect of Nagling. Nagling, which occurs only with TCP/IP, is an attempt by the transport protocol to coalesce small buffers into larger buffers so that there are not a lot of tiny packets on the physical network. While Nagling helps to reduce small packets on the network, it does it by delaying sends until there is more data available to send. With small buffers passed to **send()** these delays can *significantly* impact performance, slowing down the application by as much as 10,000%.

Minimizing Buffer Copies

Under most UNIX implementations, every sockets I/O request results in a buffer copy to or from a system buffer. This generally simplifies things for applications, but at the cost of extra CPU overhead to perform the buffer copy.

Under Windows NT and Windows '95 operating systems, Windows Sockets attempts to avoid buffer copies whenever possible. However, application actions play a big role in how often buffer copies must be performed.

On the sending side, the rules for buffer copies are simple. By default, Windows Sockets *always* copies the user buffer into an intermediate buffer before passing it on to the transport protocol. This is because the transport protocol must hang on to the data until it receives an acknowledgment from the remote end that the data has been successfully received. However, most applications want their **send()** call to complete immediately so that the thread can proceed with other work.

Applications and services which require extremely high performance can, at the cost of some complexity, avoid the send-side buffer copies. To avoid the buffer copies, set the SO_SNDBUF socket option to 0, which tells Windows Sockets not to do send buffering. The application must then use an overlapped **WriteFile()** call to send data. These **WriteFile()** calls will not complete until the transport protocol is done with the data, so it is important to keep data available to the transport protocol by pending multiple I/O calls at any one time. For example, the application calls **WriteFile()** which returns ERROR_IO_PENDING. Before I/O completion is signaled, the application should, assuming that there is more data to be sent, can **WriteFile()** a second time. As soon as the first **WriteFile()** completes, the application calls **WriteFile()** again, giving yet more data. This "double-buffering" ensures that there is always data available in the transport protocol, so that the pipe never empties.

On the receiving side, the rules are a little different. Windows Sockets will copy incoming data directly into a user buffer if one is available. If there is no available user buffer, Windows Sockets is forced to copy into a system buffer, then into a user buffer when one eventually comes in.

A user buffer is available whenever a **recv()** or **ReadFile()** is outstanding on a socket. Therefore, if an application calls **recv()** in advance of data arriving, the data will be placed directly into the user buffer. This is another reason to avoid the **select()** call: applications which call **select()** usually do not have any user buffers available to the system, so Windows Sockets is forced to perform buffer copies whenever data comes in.

Double buffering can also be used on the receive side to avoid buffer copies. On the receive side, the application should pend two **ReadFile()**

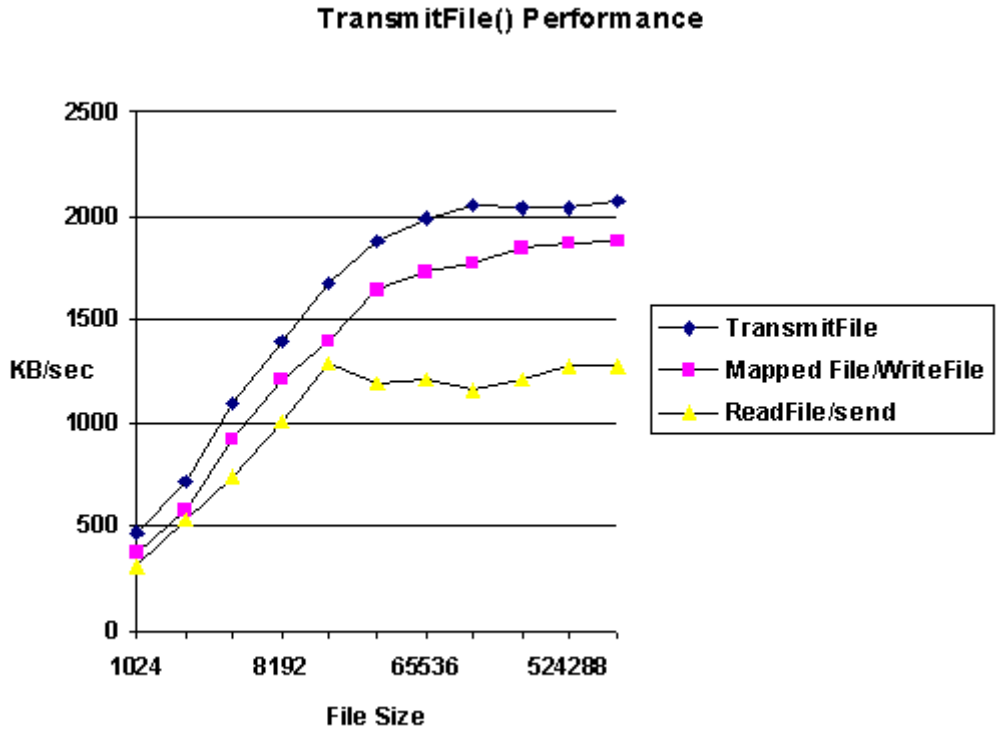
calls so that two user buffers are available to the system. Then, when data arrives, it does directly into one of the buffers. While the application is processing that data, there is still another buffer to receive incoming data, thereby avoiding the window where two chunks of data arrive back-to-back, the first going into a user buffer and the second into a system buffer.

The TransmitFile() API

Windows NT 3.51 adds a new Windows Sockets API called **TransmitFile().** This API is designed to allow applications and services to send file data very quickly over the network. It works by taking handles to a connected socket and to an open file. Then, in kernel mode, it reads data directly from the system cache and passes it off to the transport protocol. This avoids all the buffer copies, context switches, and kernel transitions associated with the typical methods of sending file data.

Another advantage of **TransmitFile()** is that it allows "head" and "tail" buffers to be specified along with the file handle. These buffers are sent before and after the file data, respectively. This is very efficient because it allows the transport protocol to combine the head buffer with the first chunk of file data and the tail buffer with the last chunk of file data.

The performance effect of **TransmitFile()** is significant, as shown in the following chart:



This chart compares three common mechanisms for sending file data: with the **ReadFile()** and **send()** APIs, by memory-mapping the file, setting SO_SNDBUF to 0 and using overlapped **WriteFile()** calls (thereby avoiding the buffer copies), and by using **TransmitFile()**. These tests were run with a 486/33 on two Ethernets with Compaq Netflex cards.

The chart shows that the cost of buffer copies, kernel transitions, and context switches make **ReadFile()** and **send()** the least fast way to send file data. This is because there are two buffer copies for every I/O (into the user buffer for **ReadFile()** and from the user buffer for **send()**) as well as kernel transitions and context switches.

Using a memory-mapped file avoids a buffer copy into the user buffer when retrieving data, and using **WriteFile()** with SO_SNDBUF set to 0 avoids the buffer copy when sending the data. Therefore, this mechanism achieves much better performance, especially for larger files. However, it must still incur the costs of the kernel transitions and context switches.

The fastest method is the **TransmitFile()** which simply sits in kernel mode, using highly optimized cache manager functions to retrieve the data and tight code paths to give this data to the transport protocol. Using **TransmitFile()**, the 486/33 is nearly able to saturate the full

bandwidth of two Ethernets with file data.

Another note on **TrasnmitFile()** for high-performance file transfer is that it is relatively simple to use. The mapped file mechanism used above took 213 lines of source code, while **TransmitFile()** used only 41.

Where is Windows Sockets Headed?

Although Windows Sockets is a new technology, much evolution has already taken place in the TCP/IP community. Microsoft's new 32-bit operating system, Windows NT, will offer Windows Sockets support for both 16- and 32-bit Windows applications. Microsoft is encouraging third-parties and corporate developers to use Windows Sockets as the client-server and distributed application API by including the Windows Sockets API as part of WOSA, the Windows Open Services Architecture.

Several PC-based TCP/IP implementations are offering Windows Sockets support, and over two dozen application vendors are shipping Windows Sockets-compatible applications. Moreover, many corporations are standardizing on Windows Sockets as the API of choice for client-server networking application development.

The Windows Sockets waters may remain calm for a short while, probably only long enough to allow application vendors and TCP/IP implementors to produce 1.1-compatible offerings. There are already some ideas of what features future revisions of the specification may incorporate: transparent transport independence, access to raw sockets for lower-level network functions, and the ability to share a connected socket between different applications to name a few.

The success of the Windows Sockets 1.1 effort has spawned significant interest in the next version, called Windows Sockets 2.0. This new standard will remain backward-compatible with Windows Sockets 1.1 while solving several of the key problems application and service writers face today in using Windows Sockets 1.1: transport independence, Win32 integration, and service registration and name resolution.

If Windows Sockets is a sign of things to come in Windows networking, application development under Microsoft Windows will continue to become easier, more flexible, and more powerful.

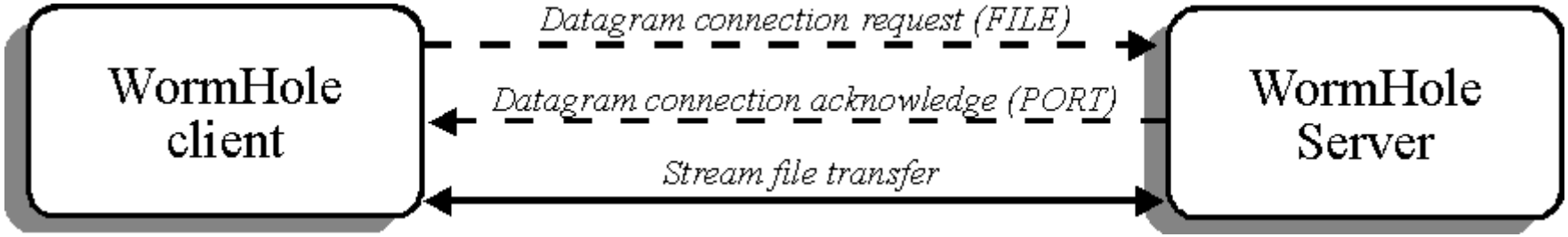
WormHole - A Sample Application

In order to demonstrate how Windows Sockets might *really* work in a real application, we developed WormHole. WormHole is a peer Windows Sockets application which allows users to establish network "wormholes" between systems and then drag and drop files to one another. The WormHole program is a simple MDI (multiple document interface) application which runs as a 16-bit application on Windows 3.1 systems with a Windows Sockets compliant TCP/IP implementations. WormHole can be conditionally compiled as a 32-bit Windows NT application which will run over Windows NT's 32-bit TCP/IP transport and 32-bit Windows Sockets interface. The makefile explains how to compile this application for Windows NT.

In order to demonstrate as many Windows Sockets concepts as possible, WormHole utilizes both stream and datagram sockets, and is

completely event-driven (asynchronous). This allows a WormHole to simultaneously service multiple client connections *and* act as a WormHole client. "Host windows" are created when a user specifies a destination system (either by IP address or hostname) to connect with. Specifying a remote system does *not* establish a network connection, it simply creates a host window on the client to provide feedback during file transfers.

"Wormholes" (connections) are established every time a user initiates a file transfer by dragging a file from the file manager into a host window. We refer to the system on which this happens as the WormHole client. The WormHole application implements a very simple protocol which would not be entirely uncommon in a production environment. The wormhole setup takes place over a simple pair of datagram frame transactions, followed by stream connection establishment which facilitates the file transfer. The following diagram illustrates the transactions for connection establishment. Remember that since WormHole is a peer application, every instance of WormHole on the network is capable of acting as both a WormHole client and a WormHole server simultaneously.



The WormHole protocol

The wormhole establishment begins on the client side by sending a FILE frame datagram which specifies to the server the name of the file to transfer, the length of the file, and a unique transaction identifier (*xid*). The *xid* allows the server to distinguish between multiple outstanding FILE requests. The server acknowledges the FILE request with either a PORT frame or a NACK frame. A PORT frame informs the client that the server will accept an incoming file, returning a specified stream socket port for the client to connect to as well as the *xid* specified in the original request. If the server wishes to refuse the FILE request, it may respond with a NACK (negative acknowledgment) frame which contains the *xid* it is refusing and optionally an error code (for example, "insufficient disk space", "duplicate filename").

Upon receipt of a PORT frame, the client establishes a stream socket connection to the specified port and begins to transfer the file. Since the server is listening on a port it created in association with a particular transaction, *and* it knows the file name and length, it is not necessary to transfer anything aside from the actual file data. Once the entire file has been received by the server, the stream connection is shut down gracefully.

WormHole Protocol Frame Types and Contents

Datagram Connection Request	Datagram Connection Acknowledge	Datagram Connection Refuse
FILE	PORT	NACK
Transaction ID	Transaction ID	Transaction ID
File length	Port Number	Error code (optional)

File name		

Because the connection request (FILE), acknowledgment (PORT) and refuse (NACK) frames are all transmitted on the network using an unreliable datagram protocol, it is quite possible for these frames to be lost in the shuffle of network activity. This said, the WormHole protocol also implements some retry timers to allow a client to retry a failed connection request. A WormHole client will retry a connection request by retransmitting up to four FILE frames to the server. To keep things simple, the server does not implement a retry timer on possible lost PORT or NACK frames. We instead rely on the client to resend a FILE request in the event that a PORT frame gets lost.

Since the server creates a local window, transaction association, and the like upon receipt of a FILE frame, it does maintain a timer per file transaction to allow for the cleanup of acknowledgment (PORT) frames which go unanswered. Finally, since the file transfer itself takes place over reliable stream sockets, it is not necessary to implement these types of precautions for the actual file transfer. WormHole simply relies on the failure of the *send()* and *recv()* APIs in the event of network or connection problems.

Windows Sockets API Function Overview

Socket Functions

accept()	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state.
bind()	Assign a local name to an unnamed socket.
closesocket()	Remove a socket descriptor from the per-process object reference table. Only blocks if SO_LINGER is set.
connect()	Initiate a connection on the specified socket.
getpeername()	Retrieve the name of the peer connected to the specified socket descriptor.
getsockname()	Retrieve the current name for the specified socket.
getsockopt()	Retrieve options associated with the specified socket descriptor.
htonl()	Convert a 32-bit quantity from host byte order to network byte order.
htons()	Convert a 16-bit quantity from host byte order to network byte order.
inet_addr()	Converts a character string representing a number in the Internet standard "." notation to an Internet address value.
inet_ntoa()	Converts an Internet address value to an ASCII string in "." notation i.e. "a.b.c.d".
ioctlsocket()	Provide control for descriptors.
listen()	Listen for incoming connections on a specified socket.
ntohl()	Convert a 32-bit quantity from network byte order to host byte order.
ntohs()	Convert a 16-bit quantity from network byte order to host byte order.

recv()*	Receive data from a connected socket.
recvfrom()*	Receive data from either a connected or unconnected socket.
select()*	Perform synchronous I/O multiplexing.
send()*	Send data to a connected socket.
sendto()*	Send data to either a connected or unconnected socket.
setsockopt()	Store options associated with the specified socket descriptor.
shutdown()	Shut down part of a full-duplex connection.
socket()	Create an endpoint for communication and return a socket descriptor.

**These functions may block if acting on a blocking socket*

Database Functions

gethostbyaddr()*	Retrieve the name(s) and address corresponding to a network address.
gethostname()	Retrieve the name of the local host.
gethostbyname()*	Retrieve the name(s) and address corresponding to a host name.
getprotobyname()*	Retrieve the protocol name and number corresponding to a protocol name.
getprotobynumber()*	Retrieve the protocol name and number corresponding to a protocol number.
getservbyname()*	Retrieve the service name and port corresponding to a service name.
getservbyport()*	Retrieve the service name and port corresponding to a port.

**These functions may block if acting on a blocking socket*

Windows Sockets Asynchronous Extensions

WSAAsyncGetHostByAddr() WSAAsyncGetHostByName() WSAAsyncGetProtoByName() WSAAsyncGetProtoByNumber() WSAAsyncGetServByName() WSAAsyncGetServByPort()	A set of functions which provide asynchronous versions of the standard Berkeley getXbyY() functions. For example, the WSAAsyncGetHostByName() function provides an asynchronous message based implementation of the standard Berkeley gethostbyname() function.
WSAAsyncSelect()	Perform asynchronous version of select().
WSACancelAsyncRequest()	Cancel an outstanding instance of a WSAAsyncGetXByY() function.

WSACancelBlockingCall()	Cancel an outstanding "blocking" API call.
WSACleanup()	Sign off from the underlying Windows Sockets DLL.
WSAGetLastError()	Obtain details of last Windows Sockets API error
WSAIsBlocking()	Determine if the underlying Windows Sockets DLL is already blocking an existing call for this thread.
WSASetBlockingHook()	"Hook" the blocking method used by the underlying Windows Sockets implementation.
WSASetLastError()	Set the error to be returned by a subsequent WSAGetLastError().
WSAStartup()	Initialize the underlying Windows Sockets DLL.
WSAUnhookBlockingHook()	Restore the original blocking function.

Obtaining the Specification

Before you get too excited about writing a Windows Sockets application, it's probably a good idea to get your hands on the specification itself. The current version of the specification is 1.1 and it is distributed electronically on the Internet and CompuServe®. If you're not able to obtain the specification electronically, ask a Windows Sockets vendor—they are generally willing to provide you with a copy on disk.

The most recent versions of the specification are available via anonymous FTPon *rhino.microsoft.com* or *sunsite.unc.edu* (in the directory */pub/micro/pc-stuff/ms-windows/winsock*). On CompuServe, check the Microsoft Software Library forum (*go msl*). A variety of different formats are readily available: **.doc** (Microsoft Word), **.wri** (Microsoft Windows Write), **.ps** (PostScript), **.txt** (ASCII text), or **.rtf** (Rich Text Format). In addition to the specification, you will also find Joel Goldberger's extremely useful Windows Sockets online help file (winsock.hlp), sample applications and other nifty stuff.

Note For developers familiar with the 1.0 specification, the Microsoft Word and PostScript versions of the 1.1 specification offer "change-bars" in the left-hand margin to denote changes made to the 1.0 specification.

The Windows Sockets Electronic Discussion List

If you're interested in Windows Sockets, your comments, experience and feedback is welcome. Currently, there is an electronic mailing list of several hundred Windows Sockets developers, users and wizards which offer technical support and discussion of the Windows Sockets API. To subscribe, e-mail *listserv@sunsite.unc.edu*, including as both subject and body the line "SUBSCRIBE WINSOCK <your Internet address>". If you're a Usenet user, the mailing list is cross-posted to the *alt.winsock* newsgroup. In addition, you also might want to peruse the *comp.programmer.win-32* and *comp.protocols.tcp-ip.ibmpc* groups which frequently see Windows Sockets-related traffic.