names using the job management programs in Chapter 6. Verify that multiple clients simultaneously access this multiprocess server system.

11–4. Run the client and server on different systems to confirm correct network operation. Modify `SrvrBcst` (Program 11–4) so that it includes the server machine name in the named pipe. Also, modify the mailslot name, currently hard-coded in Program 11–4, so that the name is taken from the mailslot response from the application server.

11–5. Modify the server so that you measure the server's utilization. (In other words, what percentage of elapsed time is spent in the server?) Maintain performance information and report this information to the client on request. Consider using the `Request.Command` field to hold the information.

11–6. Enhance the server location programs so that the client will find the server with the lowest utilization rate.

11–7. `serverNP` is designed to run indefinitely as a server, allowing clients to connect, obtain services, and disconnect. When a client disconnects, it is important for the server to free *all* associated resources, such as memory, file handles, and thread handles. Any remaining resource leaks will ultimately exhaust computer resources, causing the server to fail, and before failure there will probably be significant performance degradation. Carefully examine `serverNP` to ensure that there are no resource leaks, and, if you find any, fix them. (Also, please inform the author using the e-mail address in the preface.) *Note:* Resource leaks are a common and serious defect in many production systems. No "industry-strength" quality assurance effort is complete if it has not addressed this issue.

11–8. *Extended exercise:* Synchronization objects can synchronize threads in different processes on the same machine, but they cannot synchronize threads running in processes on different machines. Use named pipes and mailslots to create emulated mutexes, events, and semaphores to overcome this limitation.

This page intentionally left blank

CHAPTER

12 Network Programming with Windows Sockets

Named pipes and mailslots are suitable for interprocess communication between processes on the same computer or processes on Windows computers connected by a local or wide area network. The client/server application system developed in Chapter 11, starting with Program 11–2, demonstrated these capabilities.

Named pipes and mailslots (both simply referred to here as "named pipes" unless the distinction is important) have the distinct drawback, however, of not being an industry standard. Therefore, programs such as those in Chapter 11 will not port easily to non-Windows machines, nor will they interoperate with non-Windows machines. This is the case even though named pipes are protocol-independent and can run over industry-standard protocols such as TCP/IP.

Windows provides interoperability by supporting Windows Sockets, which are nearly the same as, and interoperable with, Berkeley Sockets, a de facto industry standard. This chapter shows how to use the Windows Sockets (or "Winsock") API by modifying Chapter 11's client/server system. The resulting system can operate over TCP/IP-based wide area networks, and the server, for instance, can accept requests from UNIX, Linux, and other non-Windows clients.

Readers who are familiar with Berkeley Sockets may want to proceed directly to the programming examples, which not only use sockets but also show new server features and additional thread-safe library techniques.

Winsock, by enabling standards-based interoperability, allows programmers to exploit higher-level protocols and applications, such as ftp, http, RPCs, and COM, all of which provide different, and higher-level, models for standard, interoperable, networked interprocess communication.

In this chapter, the client/server system is a vehicle for demonstrating Winsock, and, in the course of modifying the server, interesting new features are added. In particular, *DLL entry points* (Chapter 5) and *in-process DLL servers* are used for the first time. (These new features could have been incorporated in the initial Chapter 11 version, but doing so would have distracted from the development and understanding of the basic system architecture.) Finally, additional examples show how to create reentrant thread-safe libraries.

Winsock, because of conformance to industry standards, has naming conventions and programming characteristics somewhat different from the Windows functions described so far. The Winsock API is not strictly a part of the Windows API. Winsock also provides additional functions that are not part of the standard; these functions are used only as absolutely required. Among other advantages, programs will be more portable to other operating systems.

Windows Sockets

The Winsock API was developed as an extension of the Berkeley Sockets API into the Windows environment, and all Windows versions support Winsock. Winsock's benefits include the following.

- Porting code already written for Berkeley Sockets is straightforward.
- Windows machines easily integrate into TCP/IP networks, both IPv4 and IPv6. IPv6, among other features, allows for longer IP addresses, overcoming the 4-byte address limit of IPv4.
- Sockets can be used with Windows overlapped I/O (Chapter 14), which, among other things, allows servers to scale when there is a large number of active clients.
- Sockets can be treated as file `HANDLE`s for use with `ReadFile`, `WriteFile`, and, with some limitations, other Windows functions, just as UNIX allows sockets to be used as file descriptors. This capability is convenient whenever there is a need to use asynchronous I/O and I/O completion ports (Chapter 14).
- Windows provides nonportable extensions.
- Sockets can support protocols other than TCP/IP, but this chapter assumes TCP/IP. See MSDN if you use some other protocol, particularly Asynchronous Transfer Mode (ATM).

Winsock Initialization

The Winsock API is supported by a DLL (`WS2_32.DLL`) that can be accessed by linking `WS2_32.LIB` with your program (these names do not change on 64-bit machines). The DLL needs to be initialized with a nonstandard, Winsock-specific function, `WSAStartup`, which must be the first Winsock function a program calls. `WSACleanup` should be called when the program no longer needs to use Winsock functionality. *Note:* The prefix `WSA` denotes "Windows Sockets asynchronous. . . ." The asynchronous capabilities will not be used here because we'll use threads for asynchronous operation.

`WSAStartup` and `WSACleanup`, while always required, may be the only nonstandard functions you will use. A common practice is to use `#ifdef` statements to test the `WIN32` macro (normally defined from Visual Studio) so that the `WSA` functions are called only if you are building on Windows. This approach assumes, of course, that the rest of your code is platform-independent.

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

Parameters

`wVersionRequired` indicates the highest version of the Winsock DLL that you need and can use. Nonetheless, Version 2.x is available on all current Windows versions, and the examples use 2.0.

The return value is nonzero if the DLL cannot support the version you want.

The low byte of `wVersionRequired` specifies the major version, and the high byte specifies the minor version, which is the opposite of what you might expect. The `MAKEWORD` macro is usually used; thus, `MAKEWORD (2, 0)` represents Version 2.0.

`lpWSAData` points to a `WSADATA` structure that returns information on the configuration of the DLL, including the highest version available. The Visual Studio on-line help shows how to interpret the results.

`WSAGetLastError` can be used to get the error; `GetLastError` also works but is not entirely reliable. The *Examples* file socket programs use `WSAReport-Error`, a `ReportError` variation that uses `WSAGetLastError`.

When a program has completed or no longer needs to use sockets, it should call `WSACleanup` so that `WS2_32.DLL`, the sockets DLL, can free resources allocated for this process.

Creating a Socket

Once the Winsock DLL is initialized, you can use the standard (i.e., Berkeley Sockets) functions to create sockets and connect for client/server or peer-to-peer communication.

A Winsock `SOCKET` data type is analogous to the Windows `HANDLE` and can even be used with `ReadFile` and other Windows functions requiring a `HANDLE`. Call the `socket` function in order to create (or open) a `SOCKET` and return its value.

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

Parameters

The type `SOCKET` is actually defined as an `int`, so UNIX code will port without the necessity of using the Windows type definitions.

`af` denotes the address family, or protocol; use `PF_INET` (or `AF_INET`, which has the same value but is more properly used with the `bind` call) to designate IP (the Internet protocol component of TCP/IP).

`type` specifies connection-oriented (`SOCK_STREAM`) or datagram communications (`SOCK_DGRAM`), slightly analogous to named pipes and mailslots, respectively.

`protocol` is unnecessary when `af` is `AF_INET`; use `0`.

`socket` returns `INVALID_SOCKET` on failure.

You can use Winsock with protocols other than TCP/IP by specifying different protocol values; we will use only TCP/IP.

`socket`, like all the other standard functions, does not use uppercase letters in the function name. This is a departure from the Windows convention and is due to the need to conform to industry standards.

Socket Server Functions

In this discussion, a *server* is a process that accepts connections on a specified port. While sockets, like named pipes, can be used for peer-to-peer communication, this distinction is convenient and reflects the manner in which two machines connect to one another.

Unless specifically mentioned, the socket type will always be `SOCK_STREAM` in the examples. `SOCK_DGRAM` is described later in this chapter.

Binding a Socket

A server should first create a socket (using socket) and then "bind" the socket to its address and *endpoint* (the communication path from the application to a service). The socket call, followed by the bind, is analogous to creating a named pipe. There is, however, no name to distinguish sockets on a given machine. A *port number* is used instead as the service endpoint. A given server can have multiple endpoints. The bind function is shown here.

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

Parameters

s is an unbound SOCKET returned by socket.

saddr, filled in before the call, specifies the protocol and protocol-specific information, as described next. The port number is part of this structure.

namelen is sizeof(sockaddr).

The return value is normally 0 or SOCKET_ERROR in case of error. The sockaddr structure is defined as follows.

```
struct sockaddr {
   u_short sa_family;
   char sa_data [14];
   };
typedef struct sockaddr SOCKADDR, *PSOCKADDR;
```

The first member, sa_family, is the protocol. The second member, sa_data, is protocol-specific. The Internet version of sockaddr is sockaddr_in, and we use sockaddr_in in the examples.

```
struct sockaddr_in {
   short sin_family; /* AF_INET */
   u_short sin_port;
   struct in_addr sin_addr; /* 4-byte IP addr */
   char sin_zero [8];
   };
typedef struct sockaddr_in SOCKADDR_IN,
   *PSOCKADDR_IN;
```

Note the use of a short integer for the port number. The port number and other information must also be in the proper byte order, big-endian, so as to allow interoperability. The `sin_addr` member has a submember, `s_addr`, which is filled in with the familiar 4-byte IP address, such as `127.0.0.1`, to indicate the machine from which connections will be accepted. Normally, applications accept connections from any machine, so the value `INADDR_ANY` is common, although this symbolic value must be converted to the correct form, as in the next code fragment.

Use the `inet_addr` function to convert a known IP address text string (use `char` characters, not Unicode) into the required form so that you can initialize the `sin_addr.s_addr` member of a `sockaddr_in` variable, as follows:

```
sa.sin_addr.s_addr = inet_addr ("192.13.12.1");
```

A bound socket, with a protocol, port number, and IP address, is sometimes said to be a *named socket*.

Putting a Bound Socket into the Listening State

`listen` makes a server socket available for client connection. There is no analogous named pipe function.

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

`nQueueSize` indicates the number of connection requests you are willing to have queued at the socket. There is no upper bound in Winsock Version 2.0.

Accepting a Client Connection

Finally, a server can wait for a client to connect, using the accept function, which returns a new connected socket for use in the I/O operations. Notice that the original socket, now in the listening state, is used solely as an accept parameter and is not used directly for I/O.

accept blocks until a client connection request arrives, and then it returns the new I/O socket. It is possible to make a socket be nonblocking (see MSDN), but the server (Program 12–2) uses a separate accepting thread so that the server does not block. Call accept after the socket is placed in the listening state with calls to bind and listen.

```
SOCKET accept (
   SOCKET s,
   LPSOCKADDR lpAddr,
   LPINT lpAddrLen);
```

Parameters

s, the first argument, is the listening socket.

lpAddr points to a sockaddr_in structure that gives the address of the client machine.

lpAddrLen points to a variable that will receive the length of the returned sockaddr_in structure. Initialize this variable to sizeof(struct sockaddr_in) before the accept call.

Disconnecting and Closing Sockets

Disconnect a socket using

```
shutdown (s, how)
```

where s is the value returned by accept. The how value indicates if you want to disconnect send operations (SD_SEND), receive operations (SD_RECEIVE), or both (SD_BOTH). The effects of shutting down sending and/or receiving are:

- SD_SEND or SD_BOTH — Subsequent send calls will fail and the sender will send a FIN (no more data from the sender). You cannot re-enable sending on the socket.

- `SD_RECEIVE` or `SD_BOTH` — Subsequent `recv` calls will fail and the sender will send a FIN (no more data from the sender). Any queued data or data that arrives later is lost. There is no way to re-enable receiving on the socket.

`shutdown` does not close the socket or free its resources, but it does assure that all data is sent or received before the socket is closed. Nonetheless, an application should not reuse a socket after calling `shutdown`.

If you shut down a socket for sending only, you can still receive, so if there is possibility of more data, call `recv` until it returns 0 bytes. Once there is no more data, shut down receiving. If there is a socket error, then a clean disconnect is impossible.

Likewise, if you shut down a socket for receiving only, you can still send remaining data. For example, a server might stop receiving requests while it still has response or other data to transmit before shutting down sending.

Once you are finished with a socket, you can close it with

```
closesocket (SOCKET s)
```

The server first closes the socket created by `accept`, not the listening `socket`. The server should not close the listening socket until the server shuts down or will no longer accept client connections. Even if you are treating a socket as a `HANDLE` and using `ReadFile` and `WriteFile`, `CloseHandle` alone will not destroy the socket; use `closesocket`.

Example: Preparing for and Accepting a Client Connection

The following code fragment shows how to create a socket and then accept client connections. This example uses two standard functions, `htons` ("host to network short") and `htonl` ("host to network long"), that convert integers to big-endian form, as IP requires.[1]

The server port can be any unassigned short integer. Well-known ports (0–1023) and registered ports (1024–49151, with some exceptions) should not be used for your server. Select a port from the an unassigned range such as 48557–48618, 48620–49150, or 49152 and above. However, check www.iana.org/assignments/port-numbers to be certain that your port number is currently unassigned. You may also find that the port you select is in use by some other process on your computer, so you'll need to make another selection. The examples use `SERVER_PORT`, defined in `ClientServer.h` as `50000`.

[1] Windows supports little-endian, and `htons` and `htonl` perform the required conversion. The functions are implemented on non-Windows machines to behave as required by the machine architecture.

```
struct sockaddr_in srvSAddr; /* Server address struct. */
struct sockaddr_in connectAddr;
SOCKET srvSock, sockio;
DWORD addrLen;
...
srvSock = socket (AF_INET, SOCK_STREAM, 0);
srvSAddr.sin_family = AF_INET;
srvSAddr.sin_addr.s_addr = htonl (INADDR_ANY);
srvSAddr.sin_port = htons (SERVER_PORT);
bind (srvSock, (struct sockaddr *) &srvSAddr,
      sizeof srvSAddr);
listen (srvSock, 5);
addrLen = sizeof (connectAddr);
sockio = accept (srvSock,
      (struct sockaddr *) &connectAddr, &addrLen);
... Receive requests and send responses ...
/* Requests and Responses complete. Shutdown socket */
shutdown (sockio, SD_BOTH);
closesocket (sockio);
```

Socket Client Functions

A client station wishing to connect to a server must also create a socket with the `socket` function. The next step is to connect with a server, specifying a port, host address, and other information. There is just one additional function, `connect`.

Connecting to a Server

If there is a server with a listening socket, the client can connect with the `connect` function.

```
int connect (
   SOCKET s,
   LPSOCKADDR lpName,
   int nNameLen);
```

Parameters

`s` is a socket created with the `socket` function.

lpName points to a sockaddr_in structure that has been initialized with the port number and IP address of a machine with a socket, bound to the specified port, that is in listening mode.

Initialize nNameLen with sizeof (struct sockaddr_in).

A return value of 0 indicates a successful connection, whereas SOCKET_ERROR indicates failure, possibly because there is no listening socket at the specified address.

Example: Client Connecting to a Server

The following code sequence allows a client to connect to a server. Just two function calls are required, but be certain to initialize the address structure before the connect call. Error testing is omitted here but should be included in actual programs. In the example, assume that the IP address (a text string such as 192.76.33.4) is given in argv [1] on the command line.

```
SOCKET clientSock;
...
clientSock = socket (AF_INET, SOCK_STREAM, 0);
memset (&clientSAddr, 0, sizeof (clientSAddr));
clientSAddr.sin_family = AF_INET;
clientSAddr.sin_addr.s_addr = inet_addr (argv [1]);
clientSAddr.sin_port = htons (SERVER_PORT);
conVal = connect (clientSock,
      (struct sockaddr *) &clientSAddr,
      sizeof (clientSAddr));
```

Sending and Receiving Data

Socket programs exchange data using send and recv, which have nearly identical argument forms (the send buffer has the const modifier). Only send is shown here.

```
int send (
   SOCKET s,
   const char * lpBuffer,
   int nBufferLen,
   int nFlags);
```

The return value is the actual number of bytes transmitted. An error is indicated by the return value SOCKET_ERROR.