```
            nRemainRecv = 0;
            memcpy (psh->staticBuff, &tempBuff[k+1], nXfer - k - 1);
            psh->staticBuffLen = nXfer -k - 1;
        }
    }
    return !disconnect;
}

__declspec(dllexport)
BOOL SendCSMessage (MESSAGE *pMsg, PSOCKET_HANDLE psh)
{
    /* Send the the request to the server on socket sd */
    BOOL disconnect = FALSE;
    LONG32 nRemainSend, nXfer;
    LPSTR pBuffer;
    SOCKET sd;

    if (psh == NULL || pMsg == NULL) return FALSE;
    sd = psh->sk;

    pBuffer = pMsg->record;
    /* Ignore Win64 conversion warning. strlen is size_t */
    nRemainSend = min(strlen (pBuffer) + 1, MAX_MESSAGE_LEN);

    while (nRemainSend > 0 && !disconnect)  {
        /* send does not guarantee that the entire message is sent */
        nXfer = send (sd, pBuffer, nRemainSend, 0);
        if (nXfer <= 0) {
            disconnect = TRUE;
        }
        nRemainSend -=nXfer; pBuffer += nXfer;
    }

    return !disconnect;
}
```

Datagrams

Datagrams are similar to mailslots and are used in similar circumstances. There is no connection between the sender and receiver, and there can be multiple receivers. Delivery to the receiver is not ensured with either mailslots or datagrams, and successive messages will not necessarily be received in the order they were sent.

The first step in using datagrams is to specify `SOCK_DGRAM` in the `type` field when creating the socket with the `socket` function.

Next, use `sendto` and `recvfrom`, which take the same arguments as `send` and `recv`, but add two arguments to designate the partner station. Thus, the `sendto` function is as follows.

```
int sendto (
   SOCKET s,
   LPSTR lpBuffer,
   int nBufferLen,
   int nFlags,
   LPSOCKADDR lpAddr,
   int nAddrLen);
```

`lpAddr` points to an `sockaddr` address structure where you can specify the name of a specific machine and port, or you can specify that the datagram is to be broadcast to multiple computers; see the next section.

When using `recvfrom`, you specify the computers (perhaps all) from which you are willing to accept datagrams; also see the next section.

As with mailslots, datagram messages should be short; MSDN recommends 512 as the length limit for the data portion, as that limit avoids having the message sent in fragments.

Datagram Broadcasting

Several steps are necessary to broadcast `sendto` messages to multiple computers. Here are the basic steps; see MSDN for complete details:

- Set the `SOCK_DGRAM` socket options by calling `setsockopt`, specifying the `SO_BROADCAST` option. Also, set this option for sockets that are to receive broadcast messages.
- Set the client's `lpAddr sin_addr_in.s_addr` value to `INADDR_BROADCAST`.
- Set the port number as in the preceding examples.
- The broadcasts will be sent to and received by all computer interfaces (that is, all computers with a datagram socket with the `SO_BROADCAST` option) to that port.

Using Datagrams for Remote Procedure Calls

A common datagram usage is to implement RPCs. Essentially, in the most common situation, a client sends a request to a server using a datagram. Because delivery is not ensured, the client will retransmit the request if a response, also using a datagram, is not received from the server after a wait period. The server must be prepared to receive the same request several times.

The important point is that the RPC client and server do not require the overhead of a stream socket connection; instead, they communicate with simple requests and responses. As an option, the RPC implementation ensures reliability through time-outs and retransmissions, simplifying the application program. Alternatively, the client and server are frequently implemented so as to use *stateless protocol* (they do not maintain any state information about previous messages), so each request is independent of other requests. Again, application design and implementation logic are greatly simplified.

Berkeley Sockets versus Windows Sockets

Programs that use standard Berkeley Sockets calls will port to Windows Sockets, with the following important exceptions.

- You must call `WSAStartup` to initialize the Winsock DLL.
- You must use `closesocket` (which is not portable), rather than `close`, to close a socket.
- You must call `WSACleanup` to shut down the DLL.

Optionally, you can use the Windows data types such as `SOCKET` and `LONG` in place of `int`, as was done in this chapter. Programs 12–1 and 12–2 were ported from UNIX, and the effort was minimal. It was necessary, however, to modify the DLL and process management sections. Exercise 12–12 suggests that you port these two programs back to UNIX.

Overlapped I/O with Windows Sockets

Chapter 14 describes asynchronous I/O, which allows a thread to continue running while an I/O operation is in process. Sockets with Windows asynchronous I/O are discussed in that chapter.

Most asynchronous programming can be achieved uniformly and easily using threads. For example, `serverSK` uses an accept thread rather than a nonblocking

socket. Nonetheless, I/O completion ports, which are associated with asynchronous I/O, are important for scalability when there is a large number of clients. This topic is also described in Chapter 14, and Chapter 9 discussed the same situation in the context of NT6 thread pools.

Windows Sockets Additional Features

Windows Sockets 2 adds several areas of functionality, including those listed here.

- Standardized support for overlapped I/O (see Chapter 14). This is considered to be the most important enhancement.
- Scatter/gather I/O (sending and receiving from noncontiguous buffers in memory).
- The ability to request quality of service (speed and reliability of transmission).
- The ability to organize sockets into groups. The quality of service of a socket group can be configured, so it does not have to be done on a socket-by-socket basis. Also, the sockets belonging to a group can be prioritized.
- Piggybacking of data onto connection requests.
- Multipoint connections (comparable to conference calls).

Summary

Windows Sockets allows the use of an industry-standard API, so that your programs can be interoperable and nearly portable in source code form. Winsock is capable of supporting nearly any network protocol, but TCP/IP is the most common.

Winsock is comparable to named pipes (and mailslots) in both functionality and performance, but portability and interoperability are important reasons for considering sockets. Keep in mind that socket I/O is not atomic, so it is necessary to ensure that a complete message is transmitted.

This chapter covered the Winsock essentials, which are enough to build a workable client/server application system. There is, however, much more, including asynchronous usage; see the Additional Reading references for more information.

This chapter also provided examples of using DLLs for in-process servers and for creating thread-safe libraries.

Looking Ahead

Chapters 11 and 12 have shown how to develop servers that respond to client requests. Servers, in various forms, are common Windows applications. Chapter 13 describes Windows Services, which provide a standard way to create and manage servers, in the form of services, permitting automated service start-up, shutdown, and monitoring. Chapter 13 shows how to turn a server into a manageable service.

Additional Reading

Windows Sockets

Network Programming for Microsoft Windows by Jim Ohlund is a good Winsock reference.

Berkeley Sockets and TCP/IP

W. R. Stevens's *TCP/IP Illustrated, Volume 3,* covers sockets and much more, while the first two volumes in the series describe the protocols and their implementation. The same author's *UNIX Network Programming* provides comprehensive coverage that is valuable even for non-UNIX machines. Another reference is Michael Donahoo and Kenneth Calvert, *TCP/IP Sockets in C: Practical Guide for Programmers*.

Exercises

12–1. Use `WSAStartup` to determine the highest and lowest Winsock version numbers supported on the machines accessible to you.

12–2. Use the `JobShell` program from Chapter 6 to start the server and several clients, where each client is created using the "detached" option. Eventually, shut down the server by sending a console control event through the `kill` command. Can you suggest any improvements in the `serverSK` shutdown logic?

12–3. Modify the client and server programs (Programs 12–1 and 12–2) so that they use datagrams to locate a server. The mailslot solution in Chapter 11 could be used as a starting point.

12–4. Modify the named pipe server in Chapter 11 (Program 11–3) so that it creates threads on demand instead of a server thread pool. Rather than predefining a fixed maximum for the number of named pipe instances, allow the application to determine the maximum.

12–5. Perform experiments to determine whether in-process servers are faster than out-of-process servers. For example, you can use the word count example (Program 12–4); there is an executable `wc` program as well as the DLL function shown in Program 12–4.

12–6. The number of clients that `serverSK` can support is bounded by the array of server thread arguments. Modify the program so that there is no such bound. You will need to create a data structure that allows you to add and delete thread arguments, and you also need to be able to scan the structure for terminated server threads. An alternative, and arguably simpler, approach would be to have each server thread manage its own state without the boss thread being involved other than to ask the server threads to shut down and wait for them to complete.

12–7. Develop additional in-process servers. For example, convert the `grep` program (see Chapter 6).

12–8. Enhance the server (Program 12–2) so that you can specify multiple DLLs on the command line.

12–9. Investigate the `setsockopt` function and the `SO_LINGER` option. Apply the option to one of the server examples.

12–10. Ensure that `serverSK` is free of resource leaks. Do the same with `serverSKST`, which was modified to use the DLL in Program 12–5.

12–11. Extend the exception handler in Program 12–4 so that it reports the exception and exception type at the end of the temporary file used for the server results.

12–12. *Extended exercise (requires extra equipment):* If you have access to a UNIX machine that is networked to your Windows machine, port `clientSK` to the UNIX machine and have it access `serverSK` to run Windows programs. You will, of course, need to convert data types such as `DWORD` and `SOCKET` to other types (`unsigned int` and `int`, respectively, in these two cases). Also, you will need to ensure that the message length is transmitted in big-endian format. Use functions such as `htonl` to convert the message lengths. Finally, port `serverSK` to UNIX so that Windows machines can execute UNIX commands on a remote system. Convert the DLL calls to shared library calls.

12–13. `serverSK` shuts down the accept thread by closing the connection socket (see Run 12–3). Is there a better way to terminate the accept thread? Potential approaches to investigate include queuing an APC as in Chapter 10. Or, can you use the Windows extended function, `AcceptEx` (Chapter 14 may help)?

12–14. A comment after Program 12–5 (`SendReceiveSKST`) mentions that you cannot assure that `DLL_THREAD_DETACH` will be invoked for every thread, and, therefore, there could be resource leaks (memory, temporary files, open file handles, etc.). Implement a solution in `SendReceiveSKST` that uses `DLL_PROCESS_DETACH` to free all allocated memory. Because you cannot find the allocated memory with `TlsGetValue`, maintain a list of all allocated memory.

12–15. Read about the SSL in MSDN and the Additional Reading references. Enhance the programs to use SSL for secure client/server communication.

This page intentionally left blank

CHAPTER

13 Windows Services

The server programs in Chapters 11 and 12 are console applications. In principle, the servers could run indefinitely, serving numerous clients as they connect, send requests, receive responses, and disconnect. That is, these servers could provide continuous services, but to be fully effective, the services should be manageable.

Windows Services,[1] previously known as NT Services, provide the management capabilities required to convert the servers into services that can be initiated on command or at boot time, before any user logs in, and can also be paused, resumed, terminated, and monitored. The registry maintains information about services.

Ultimately, any server, such as those developed in Chapters 11 and 12, should be converted to a service, especially if it is to be widely used by customers or within an organization.

Windows provides a number of services; examples include the DNS Client, several SQL Server services, and Terminal Services. The computer management snap-in, accessible from the Control Panel, displays the full set of services.

Chapter 6's `JobShell` (Program 6–3) provides rudimentary server management by allowing you to bring up a server under job control and send a termination signal. Windows Services, however, are much more comprehensive and robust, and the main example is a conversion of `JobShell` so that it can control Windows Services.

This chapter also shows how to convert an existing console application into a Windows service and how to install, monitor, and control the service. Logging, which allows a service to log its actions to a file, is also described.

[1] This terminology can be confusing because Windows provides numerous services that are not the Windows Services described here. However, the context should make the meaning clear, just as using the term "Windows" throughout the book when talking specifically about the API has not been a problem.

Writing Windows Services—Overview

Windows Services run under the control of a Service Control Manager (SCM). You can interact with the SCM to control services in three ways:

1. Use the management snap-in labeled Services under Systems and Maintenance, Administrative Tools in the Control Panel.
2. Control services with the `sc.exe` command line tool.
3. Control the SCM programmatically, as Program 13-3 demonstrates.

Converting a console application, such as `serverNP` or `serverSK`, to a Windows Service requires three major steps to place the program under the SCM.

1. Create a new `main()` entry point that registers the service with the SCM, supplying the logical service entry points and names.
2. Convert the old `main()` entry point function to *`ServiceMain()`*, which registers a service control handler and informs the SCM of its status. The remaining code is essentially that of the existing program, although you can add event logging commands. The name *`ServiceMain()`* is a placeholder for the name of a logical service, and there can be one or more logical services in a single process.
3. Write the service control handler function to respond to commands from the SCM.

As we describe these three steps, there are several references to creating, starting, and controlling services. Later sections describe the specifics, and Figure 13–1, later in the chapter, illustrates the component interactions.

The `main()` Function

The new `main()` function, which the SCM calls, has the task of registering the service with the SCM and starting the service control dispatcher. This requires a call to the `StartServiceCtrlDispatcher` function with the name(s) and entry point(s) of one or more logical services.

```
BOOL StartServiceCtrlDispatcher (
   SERVICE_TABLE_ENTRY *lpServiceStartTable)
```

The single parameter, `lpServiceStartTable`, is the address of an array of `SERVICE_TABLE_ENTRY` items, where each item is a logical service name and entry point. The end of the array is indicated by a pair of `NULL` entries.

The return is `TRUE` if the registration was successful.

The main thread of the service process that calls `StartServiceCtrlDispatcher` connects the thread to the SCM. The SCM registers the service(s) with the calling thread as the service control dispatcher thread. The SCM does not return to the calling thread until all services have terminated. Notice, however, that the logical services do not actually start at this time; starting the service requires the `StartService` function, which we describe later.

Program 13–1 shows a typical service main program with a single logical service.

Program 13–1 `main`: The Main Service Entry Point

```
#include "Everything.h"

void WINAPI ServiceMain (DWORD argc, LPTSTR argv[]);
static LPTSTR serviceName = _T("SocketCommandLineService");

/* Main routine that starts the service control dispatcher. */
VOID _tmain (int argc, LPTSTR argv[])
{
   SERVICE_TABLE_ENTRY dispatchTable[] =
   {
      { serviceName, ServiceMain },
      { NULL, NULL }
   };

   if (!StartServiceCtrlDispatcher (dispatchTable))
      ReportError (_T("Failed to start srvc ctrl dis."), 1, TRUE);
   /* ServiceMain () will not run until started by the SCM. */
   /* Return here only when all services have terminated. */
   return;
}
```

ServiceMain() Functions

The dispatch table specifies the functions, as shown in Program 13–1, and each function represents a logical service. The functions are enhanced versions of the base program that is being converted to a service, and the SCM invokes each logical service on its own thread. A logical service may, in turn, start up additional threads, such as the server worker threads that `serverSK` and `serverNP` create. Frequently, there is just one logical service within a Windows Service. In Program 13–2, the logical service is

adapted from the main server (Program 12–2). It would be possible, however, to run both socket and named pipe logical services under the same Windows service, in which case you would supply two service main functions.

While the *ServiceMain()* function is an adaptation of a `main()` function with argument count and argument string parameters, there is one small change. The function should be declared `void WINAPI` rather than having an `int` return of a normal `main()` function.

Registering the Service Control Handler

A service control handler, called by the SCM, must be able to control the associated logical service. The console control handler in `serverSK`, which sets a global shutdown flag, illustrates, in limited form, what is expected of a handler. First, however, each logical service must immediately register a handler using `RegisterServiceCtrlHandlerEx`. The function call should be at the beginning of *ServiceMain()* and not called again. The SCM, after receiving a control request for the service, calls the handler.

```
SERVICE_STATUS_HANDLE
   RegisterServiceCtrlHandlerEx (
   LPCTSTR lpServiceName,
   LPHANDLER_FUNCTION_EX lpHandlerProc,
   LPVOID lpContext)
```

Parameters

`lpServiceName` is the user-supplied service name provided in the service table entry for this logical service; it should match a `ServiceMain` function name registered with `StartServiceCtrlDispatcher`.

`lpHandlerProc` is the address of the extended handler function, described in a later section.

`lpContext` is user-defined data passed to the control handler. This allows a single control handler to distinguish between multiple services using the same handler.

The return value, which is a `SERVICE_STATUS_HANDLE` object, is `0` if there is an error, and the usual methods can be used to analyze errors.

Setting the Service Status

Now that the handler is registered, the next immediate task is to set the service status to `SERVICE_START_PENDING` using `SetServiceStatus`. `SetService-`