`nFlags` can be used to indicate urgency (such as out-of-band data), and the `MSG_PEEK` flag can be used to look at incoming data without reading it.

The most important fact to remember is that `send` and `recv` *are not atomic,* and there is no assurance that all the requested data has been received or sent. "Short sends" are extremely rare but possible, as are "short receives." There is no concept of a message as with named pipes; therefore, you need to test the return value and resend or transmit until all data has been transmitted.

You can also use `ReadFile` and `WriteFile` with sockets by casting the socket to a `HANDLE` in the function call.

Comparing Named Pipes and Sockets

Named pipes, described in Chapter 11, are very similar to sockets, but there are significant usage differences.

- Named pipes can be message-oriented, which can simplify programs.
- Named pipes require `ReadFile` and `WriteFile`, whereas sockets can also use `send` and `recv`.
- Sockets, unlike named pipes, are flexible so that a user can select the protocol to use with a socket, such as TCP or UDP. The user can also select protocols based on quality of service and other factors.
- Sockets are based on an industry standard, allowing interoperability with non-Windows machines.

There are also differences in the server and client programming models.

Comparing Named Pipe and Socket Servers

When using sockets, call `accept` repetitively to connect to multiple clients. Each call will return a different connected socket. Note the following differences relative to named pipes.

- Named pipes require you to create each named pipe instance with `CreateNamedPipe`. `accept` creates the socket instances.
- There is no upper bound on the number of socket clients (`listen` limits only the number of queued clients), but there can be a limit on the number of named pipe instances, depending on the first call to `CreateNamedPipe`.
- There are no socket convenience functions comparable to `TransactNamedPipe`.

- Named pipes do not have explicit port numbers and are distinguished by name.

A named pipe server requires two function calls (`CreateNamedPipe` and `ConnectNamedPipe`) to obtain a usable `HANDLE`, whereas socket servers require four function calls (`socket`, `bind`, `listen`, and `accept`).

Comparing Named Pipes and Socket Clients

Named pipes use `WaitNamedPipe` followed by `CreateFile`. The socket sequence is in the opposite order because the `socket` function can be regarded as the creation function, while `connect` is the blocking function.

An additional distinction is that `connect` is a socket client function, while the similarly named `ConnectNamedPipe` is a server function.

Example: A Socket Message Receive Function

It is frequently convenient to send and receive messages as a single unit. Named pipes can do this, as shown in Chapter 11. Sockets, however, require that you provide a mechanism to specify and determine message boundaries. One common method is to create a message header with a length field, followed by the message itself, and we'll use message headers in the following examples. Later examples use a different technique, end-of-string null characters, to mark message boundaries. Fixed-length messages provide yet another solution.

The following function, `ReceiveMessage`, receives message length headers and message bodies. The `SendMessage` function is similar.

Notice that the message is received in two parts: the header and the contents. The user-defined `MESSAGE` type with a 4-byte message length header is:

```
typedef struct {
   LONG32 msgLen;/* Message length, excluding this field */
   BYTE record [MAX_MSG_LEN];
} MESSAGE;
```

Even the 4-byte header requires repetitive `recv` calls to ensure that it is read in its entirety because `recv` is not atomic.

Note: The message length variables are fixed-precision `LONG32` type to remind readers that the length, which is included in messages that may be transferred to and from programs written in other languages (such as Java) or running on other machines, where long integers may be 64 bits, will have a well-defined, unambiguous length.

```
DWORD ReceiveMessage (MESSAGE *pMsg, SOCKET sd)
{
   /* A message has a 4-byte length field, followed
      by the message contents. */
   DWORD disconnect = 0;
   LONG32 nRemainRecv, nXfer;
   LPBYTE pBuffer;
   /* Read message. */
   /* First the length header, then contents. */
   nRemainRecv = 4; /* Header field length. */
   pBuffer = (LPBYTE) pMsg; /* recv may not */
   /* Receive the header. */
   while (nRemainRecv > 0 && !disconnect) {
      nXfer = recv (sd, pBuffer, nRemainRecv, 0);
      disconnect = (nXfer == 0);
      nRemainRecv -=nXfer; pBuffer += nXfer;
   }
   /* Read the message contents. */
   nRemainRecv = pMsg->msgLen;
   /* Exclude buffer overflow */
   nRemainRecv = min(nRemainRecv, MAX_RQRS_LEN);
   while (nRemainRecv > 0 && !disconnect) {
      nXfer = recv (sd, pBuffer, nRemainRecv, 0);
      disconnect = (nXfer == 0);
      nRemainRecv -=nXfer; pBuffer += nXfer;
   }
   return disconnect;
}
```

Example: A Socket-Based Client

Program 12–1 reimplements the client program, which in named pipe form is Program 11–2, `clientNP`. The conversion is straightforward, with several small differences.

- Rather than locating a server using mailslots, the user enters the IP address on the command line. If the IP address is not specified, the default address is `127.0.0.1`, which indicates the current machine.
- Functions for sending and receiving messages, such as `ReceiveMessage`, are used but are not shown here.

- The port number, SERVER_PORT, is defined in the header file, ClientServer.h.

While the code is written for Windows, there are no Windows dependencies other than the WSA calls.

Comment: The programs in this chapter do not use generic characters. This is a simplification driven by the fact that inet_addr does not accept Unicode strings.

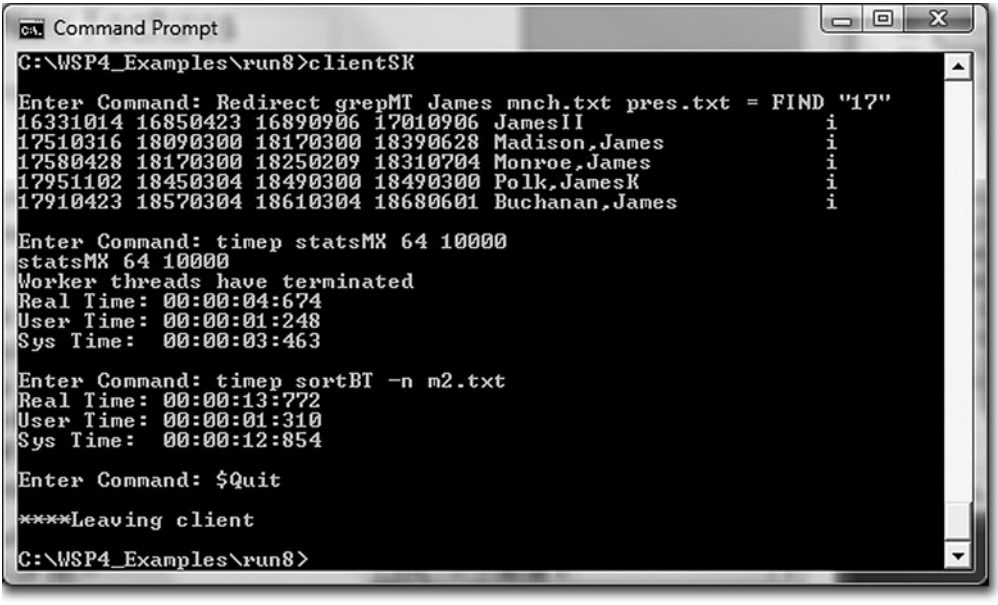
Program 12–1 clientSK: Socket-Based Client

```
/* Chapter 12. clientSK.c */
/* Single-threaded command line client. */
/* WINDOWS SOCKETS VERSION. */
/* Reads a sequence of commands to send to a server process */
/* over a socket connection. Wait for and display response. */

#include "Everything.h"
#include "ClientServer.h"/* Defines request and response records. */

static BOOL SendRequestMessage (REQUEST *, SOCKET);
static BOOL ReceiveResponseMessage (RESPONSE *, SOCKET);

struct sockaddr_in clientSAddr;/* Client Socket address structure */

int main (int argc, LPSTR argv[])
{
    SOCKET clientSock = INVALID_SOCKET;
    REQUEST request;/* See clntcrvr.h */
    RESPONSE response;/* See clntcrvr.h */
    WSADATA WSStartData;    /* Socket library data structure   */
    BOOL quit = FALSE;
    DWORD conVal;

    /* Initialize the WS library. Ver 2.0 */
    WSAStartup (MAKEWORD (2, 0), &WSStartData);
    /* Connect to the server */
    /* Follow the standard client socket/connect sequence */
    clientSock = socket(AF_INET, SOCK_STREAM, 0);
    memset (&clientSAddr, 0, sizeof(clientSAddr));
    clientSAddr.sin_family = AF_INET;
    if (argc >= 2)
        clientSAddr.sin_addr.s_addr = inet_addr (argv[1]);
    else
        clientSAddr.sin_addr.s_addr = inet_addr ("127.0.0.1");

    clientSAddr.sin_port = htons(SERVER_PORT);
    conVal = connect (clientSock, (struct sockaddr *)&clientSAddr,
                      sizeof(clientSAddr));
```

```
    /*  Main loop to prompt user, send request, receive response */
    while (!quit) {
        _tprintf (_T("%s"), _T("\nEnter Command: "));
        fgets (request.record, MAX_RQRS_LEN, stdin);
        /* Get rid of the new line at the end */
        request.record[strlen(request.record)-1] = '\0';
        if (strcmp (request.record, "$Quit") == 0) quit = TRUE;
        SendRequestMessage (&request, clientSock);
        if (!quit) ReceiveResponseMessage (&response, clientSock);
    }

    shutdown (clientSock, SD_BOTH); /* Disallow sends and receives */
    closesocket (clientSock);
    WSACleanup();
    _tprintf (_T("\n****Leaving client\n"));
    return 0;
}
```

Running the Socket Client

The socket server is complex with long program listings. Therefore, Run 12–1 shows the client in operation, assuming that there is a running server. The commands are familiar, and operation is very similar to Chapter 11's named pipe client.



Run 12–1 `clientSK`: Socket Client Operation

Example: A Socket-Based Server with New Features

`serverSK`, Program 12–2, is similar to `serverNP`, Program 11–3, but there are several changes and improvements.

- Rather than creating a fixed-size thread pool, we now create *server threads on demand*. Every time the server accepts a client connection, it creates a server worker thread, and the thread terminates when the client quits.
- The server creates a separate *accept thread* so that the main thread can poll the global shutdown flag while the `accept` call is blocked. While it is possible to specify nonblocking sockets, threads provide a convenient and uniform solution. It's worth noting that a lot of the extended Winsock functionality is designed to support asynchronous operation, and Windows threads allow you to use the much simpler and more standard synchronous socket functionality.
- The thread management is improved, at the cost of some complexity, so that the state of each thread is maintained.
- This server also supports *in-process servers* by loading a DLL during initialization. The DLL name is a command line option, and the server thread first tries to locate an entry point in the DLL. If successful, the server thread calls the DLL entry point; otherwise, the server creates a process, as in `serverNP`. A sample DLL is shown in Program 12–4. The DLL needs to be trusted, however, because any unhandled exception could crash or corrupt the server, as would changes to the environment.

In-process servers could have been included in `serverNP` if desired. The biggest advantage of in-process servers is that no context switch to a different process is required, potentially improving performance. The disadvantage is that the DLL runs in the server process and could corrupt the server, as described in the last bullet. Therefore, use only trusted DLLs.

The server code is Windows-specific, unlike the client, due to thread management and other Windows dependencies.

The Main Program

Program 12–2 shows the main program and the thread to accept client connections. It also includes some global declarations and definitions, including an enumerated type, `SERVER_THREAD_STATE`, used by each individual server thread.

Program 12–3 shows the server thread function; there is one instance for each connected client. The server state can change in both the main program and the server thread.

The server state logic involves both the boss and server threads. Exercise 12–6 suggests an alternative approach where the boss is not involved.

Program 12–2 `serverSK`: Socket-Based Server with In-Process Servers

```
/* Chapter 12. Client/server. SERVER PROGRAM. SOCKET VERSION. */
/* Execute the command in the request and return a response. */
/* Commands will be executed in process if a shared library */
/* entry point can be located, and out of process otherwise. */
/* ADDITIONAL FEATURE: argv [1] can be name of a DLL supporting */
/* in-process servers. */

#include "Everything.h"
#include "ClientServer.h"/* Defines the request and response records.
*/

struct sockaddr_in srvSAddr;/* Server's Socket address structure */
struct sockaddr_in connectSAddr;/* Connected socket */
WSADATA WSStartData;       /* Socket library data structure   */

enum SERVER_THREAD_STATE {SERVER_SLOT_FREE, SERVER_THREAD_STOPPED,
                        SERVER_THREAD_RUNNING, SERVER_SLOT_INVALID};

typedef struct SERVER_ARG_TAG { /* Server thread arguments */
   CRITICAL_SECTION threadCs;
   DWORDnumber;
   SOCKET sock;
   enum SERVER_THREAD_STATE thState;
   HANDLE hSrvThread;
   HINSTANCE hDll; /* Shared library handle */
} SERVER_ARG;

static BOOL ReceiveRequestMessage (REQUEST *pRequest, SOCKET);
static BOOL SendResponseMessage (RESPONSE *pResponse, SOCKET);
static DWORD WINAPI Server (PVOID);
static DWORD WINAPI AcceptThread (PVOID);
static BOOL  WINAPI Handler (DWORD);

volatile static int shutFlag = 0;
static SOCKET SrvSock = INVALID_SOCKET, connectSock = INVALID_SOCKET;

int main (int argc, LPCTSTR argv [])
{
   /* Server listening and connected sockets. */
   DWORD iThread, tStatus;
   SERVER_ARG sArgs[MAX_CLIENTS];
   HANDLE hAcceptThread = NULL;
   HINSTANCE hDll = NULL;
```

```
/* Console control handler to permit server shutdown */
SetConsoleCtrlHandler (Handler, TRUE);

/*  Initialize the WS library. Ver 2.0 */
WSAStartup (MAKEWORD (2, 0), &WSStartData);

/* Open command library DLL if it is specified on command line */
if (argc > 1) hDll = LoadLibrary (argv[1]);

/* Initialize thread arg array */
for (iThread = 0; iThread < MAX_CLIENTS; iThread++) {
   InitializeCriticalSection (&sArgs[iThread].threadCs);
   sArgs[iThread].number = iThread;
   sArgs[iThread].thState = SERVER_SLOT_FREE;
   sArgs[iThread].sock = 0;
   sArgs[iThread].hDll = hDll;
   sArgs[iThread].hSrvThread = NULL;
}
/*  Follow standard server socket/bind/listen/accept sequence */
SrvSock = socket(PF_INET, SOCK_STREAM, 0);

/*  Prepare the socket address structure for binding the
    server socket to port number "reserved" for this service.
    Accept requests from any client machine.  */

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, MAX_CLIENTS);

/* Main thread becomes listening/connecting/monitoring thread */
/* Find an empty slot in the server thread arg array */
while (!shutFlag) {
   iThread = 0;
   while (!shutFlag) {
      /* Continuously poll thread State of all server slots */
      EnterCriticalSection(&sArgs[iThread].threadCs);
      __try {
         if (sArgs[iThread].thState ==
            SERVER_THREAD_STOPPED) {
            /* stopped, either normally or a shutdown request */
            /* Wait for it to stop, and free the slot */
            WaitForSingleObject(sArgs[iThread].hSrvThread,
                                INFINITE);
            CloseHandle (sArgs[iThread].hSrvThread);
            sArgs[iThread].hSrvThread = NULL;
            sArgs[iThread].thState = SERVER_SLOT_FREE;
         }
         /* Free slot or shut down. Use slot for new connection */
```

```
            if (sArgs[iThread].thState == SERVER_SLOT_FREE
                  ||shutFlag) break;
         }
         __finally {
            LeaveCriticalSection(&sArgs[iThread].threadCs);
         }

         iThread = (iThread++) % MAX_CLIENTS;
         if (iThread == 0) Sleep(50); /* Break the polling loop */
         /* An alternative: use an event to signal a free slot */
      }
      if (shutFlag) break;
      /* sArgs[iThread] == SERVER_SLOT_FREE */
      /* Wait for a connection on this socket */
      /* Use a separate accept thread to poll the shutFlag flag */
      hAcceptThread = (HANDLE)_beginthreadex (NULL, 0, AcceptThread,
                        &sArgs[iThread], 0, NULL);
      while (!shutFlag) {
         tStatus = WaitForSingleObject (hAcceptThread, CS_TIMEOUT);
         if (tStatus == WAIT_OBJECT_0) {
            /* sArgs[iThread] == SERVER_THREAD_RUNNING */
            if (!shutFlag) {
               CloseHandle (hAcceptThread);
               hAcceptThread = NULL;
            }
            break;
         }
      }
   }  /* OUTER while (!shutFlag) */

   /* shutFlag == TRUE */
   _tprintf(_T("Shutdown in process. Wait for server threads\n"));
   /* Wait for any active server threads to terminate */
   /* Try continuously as some threads may be long running. */

   while (TRUE) {
      int nRunningThreads = 0;
      for (iThread = 0; iThread < MAX_CLIENTS; iThread++) {
         EnterCriticalSection(&sArgs[iThread].threadCs);
         __try {
            if (
               sArgs[iThread].thState == SERVER_THREAD_RUNNING ||
               sArgs[iThread].thState == SERVER_THREAD_STOPPED) {
                  if (WaitForSingleObject
                     (sArgs[iThread].hSrvThread, 10000) ==
                              WAIT_OBJECT_0) {
                  CloseHandle (sArgs[iThread].hSrvThread);
                  sArgs[iThread].hSrvThread = NULL;
                  sArgs[iThread].thState = SERVER_SLOT_INVALID;
               } else
```

```
                    if (WaitForSingleObject
                        (sArgs[iThread].hSrvThread, 10000) ==
                                WAIT_TIMEOUT) {
                        nRunningThreads++;
                    } else {
                        _tprintf(_T("Error waiting: slot %d\n"), iThread);
                    }

                }
            }
            __finally { LeaveCriticalSection(&sArgs[iThread].threadCs);}
        }
        if (nRunningThreads == 0) break;
    }

    if (hDll != NULL) FreeLibrary (hDll);

    /* Redundant shutdown */
    shutdown (SrvSock, SD_BOTH);
    closesocket (SrvSock);
    WSACleanup();
    if (hAcceptThread != NULL)
        WaitForSingleObject(hAcceptThread, INFINITE));
    return 0;
}

static DWORD WINAPI AcceptThread (PVOID pArg)
{
    LONG addrLen;
    SERVER_ARG * pThArg = (SERVER_ARG *)pArg;

    addrLen = sizeof(connectSAddr);
    pThArg->sock =
         accept (SrvSock, (struct sockaddr *)&connectSAddr, &addrLen);

    /* A new connection. Create a server thread */
    EnterCriticalSection(&(pThArg->threadCs));
    __try {
        pThArg->hSrvThread = (HANDLE)_beginthreadex (NULL, 0, Server,
                  pThArg, 0, NULL);
        pThArg->thState = SERVER_THREAD_RUNNING;
    }
    __finally { LeaveCriticalSection(&(pThArg->threadCs)); }
    return 0;
}

BOOL WINAPI Handler (DWORD CtrlEvent)
{
    /* Shutdown the program */
    _tprintf (_T("In console control handler\n"));
```

```
    InterlockedIncrement (&shutFlag);
    return TRUE;
}
```

The Server Thread

Program 12–3 shows the socket server thread function. There are many similarities to the named pipe server function, and some code is elided for simplicity. Also, the code uses some of the global declarations and definitions from Program 12–2.

Program 12–3 `serverSK`: Server Thread Code

```
static DWORD WINAPI Server (PVOID pArg)

/* Server thread function. One thread for every potential client. */
{
    /* Each thread keeps its own request, response,
       and bookkeeping data structures on the stack. */
    BOOL done = FALSE;
    STARTUPINFO startInfoCh;
    SECURITY_ATTRIBUTES tempSA = {. . .}; /* Inheritable handles */
    PROCESS_INFORMATION procInfo;
    SOCKET connectSock;
    int commandLen;
    REQUEST request;/* Defined in ClientServer.h */
    RESPONSE response;/* Defined in ClientServer.h.*/
    char sysCommand[MAX_RQRS_LEN], tempFile[100];
    HANDLE hTmpFile;
    FILE *fp = NULL;
    int (__cdecl *dl_addr)(char *, char *);
    SERVER_ARG * pThArg = (SERVER_ARG *)pArg;
    enum SERVER_THREAD_STATE threadState;

    GetStartupInfo (&startInfoCh);

    connectSock = pThArg->sock;
    /* Create a temp file name */
    _stprintf (tempFile, _T("ServerTemp%d.tmp"), pThArg->number);

    while (!done && !shutFlag) { /* Main Server Command Loop. */
        done = ReceiveRequestMessage (&request, connectSock);

        request.record[sizeof(request.record)-1] = '\0';
        commandLen = strcspn (request.record, "\n\t");
        memcpy (sysCommand, request.record, commandLen);
        sysCommand[commandLen] = '\0';
        _tprintf (_T("Command received on server slot %d: %s\n"),
```

```
            pThArg->number, sysCommand);

        /* Retest shutFlag; could be set in console control handler. */
        done = done || (strcmp (request.record, "$Quit") == 0)
                  || shutFlag;
        if (done) continue;

        /* Open the temporary results file. */
        hTmpFile = CreateFile (tempFile, GENERIC_READ | GENERIC_WRITE,
               FILE_SHARE_READ | FILE_SHARE_WRITE, &tempSA,
               CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

        /* Check for shared library command. For simplicity, shared */
        /* library commands take precedence over process commands */
        dl_addr = NULL; /* will be set if GetProcAddress succeeds */
        if (pThArg->hDll != NULL) { /* Try Server "In process" */
           char commandName[256] = "";
           int commandNameLength = strcspn (sysCommand, " ");
           strncpy (commandName, sysCommand,
              min(commandNameLength, sizeof(commandName)));
           dl_addr = (int (*)(char *, char *))GetProcAddress
              (pThArg->hDll, commandName);
           /* Trust this DLL not to corrupt the server */
           if (dl_addr != NULL) { /* Call the DLL */
              (*dl_addr)(request.record, tempFile);
           }
        }

        if (dl_addr == NULL) { /* No inprocess support */
           /* Create a process to carry out the command. */
           /* Same as in serverNP*/
           . . .
        }

        /* Respond a line at a time. It is convenient to use
           C library line-oriented routines at this point. */

        /* Send temp file, one line at a time, to the client. */
        /* Same as in serverNP */
           . . .
    }   /* End of main command loop. Get next command */

    /* done || shutFlag */
    /* End of command processing loop. Free resources; exit thread. */
    _tprintf (_T("Shuting down server thread # %d\n"), pThArg->number);
    closesocket (connectSock);
```