Chapter 20 Named Pipes

Named pipes are a simple interprocess communication (IPC) mechanism included in Microsoft Windows NT, and Windows 95, Windows 98, and Windows Me platforms (but not Windows CE). Named pipes provide reliable one-way and two-way data communications among processes on the same computer or among processes on different computers across a network. Developing applications using named pipes is actually quite simple and requires no formal knowledge of underlying network transport protocols (such as TCP/IP or IPX). This is because named pipes use the Microsoft Network Provider (MSNP) redirector to form communication among processes over a network, thus hiding network protocol details from the application. One of the best reasons for using named pipes as a networking communication solution is that they take advantage of security features built into the Windows NT platform.

One possible scenario for using named pipes is developing a data management system that allows only a select group of people to perform transactions. Imagine an office setting in which you have a computer that contains company secrets. You need to have these secrets accessed and maintained by management personnel only. Let's say every employee can see the computer on the network from his or her workstation. However, you do not want regular employees to obtain access to the confidential records. Named pipes work well in this situation because you can develop a server

application that, based on requests from clients, safely performs transactions on the company secrets. The server can easily limit client access to management personnel by using security features of the Windows NT platform.

What's important to remember when using named pipes as a network programming solution is that they feature a simple client/server data communication architecture that reliably transmits data. This chapter explains how to develop named pipe client and server applications. We start by explaining named pipe naming conventions, followed by basic pipe types. We'll then show how to implement a basic server application, followed by advanced server programming details. Next we discuss how to develop a basic client application. By the chapter's end, we uncover the known problems and limitations of named pipes.

20.1 Named Pipe Implementation Details

Named pipes are designed around the Windows file system using the Named Pipe File System (NPFS) interface. As a result, client and server applications use standard Windows file system API functions such as ReadFile and WriteFile to send and receive data. Using these API functions allows applications to take advantage of Windows file system naming conventions and Windows NT file system security. NPFS relies on the MSNP redirector to send and receive named pipe data over a network. This makes the interface protocol-independent: when developing an application that uses named pipes to form communications among processes across a network, so a programmer does not have to worry about the details of underlying network transport protocols, such as TCP and IPX. Named pipes are identified to NPFS using the Universal Naming Convention. Chapter 18 describes the UNC, the Windows redirector, and security in greater detail.

20.1.1 Named Pipe Naming Conventions

Named pipes are identified using the following UNC format:

\\server\Pipe\[path]name

This string is divided into three parts: \\server, \Pipe, and \[path]name. The first string part, \\server, represents the server name in which a named pipe is created and the server that listens for incoming connections. The second part, \Pipe, is a hard-coded mandatory string requirement for identifying that this filename belongs to NPFS. The third part, \[path]name, allows applications to uniquely define and identify a named pipe name, and it can have multiple levels of directories. For example, the following name types are legal for identifying a named pipe:

\\myserver\PIPE\mypipe

\\Testserver\pipe\cooldirectory\funtest\jim
\\.\Pipe\Easynamedpipe

The server string portion can be represented as a dot (.) or a server name.

20.1.2 Byte Mode and Message Mode

Named pipes offer two basic communication modes: byte mode and message mode. In byte mode, messages travel as a continuous stream of bytes between the client and the server. This means that a client application and a server application do not know precisely how many bytes are being read from or written to a pipe at any given moment. Therefore a write on one side will not always result in a same-size read on the other. This allows a client and a server to transfer data without regard to the contents of the data. In message mode, the client and the server send and receive data in discrete units. Every time a message is sent on the pipe, it must be read as a complete message. Figure 20-1 compares the two pipe modes.

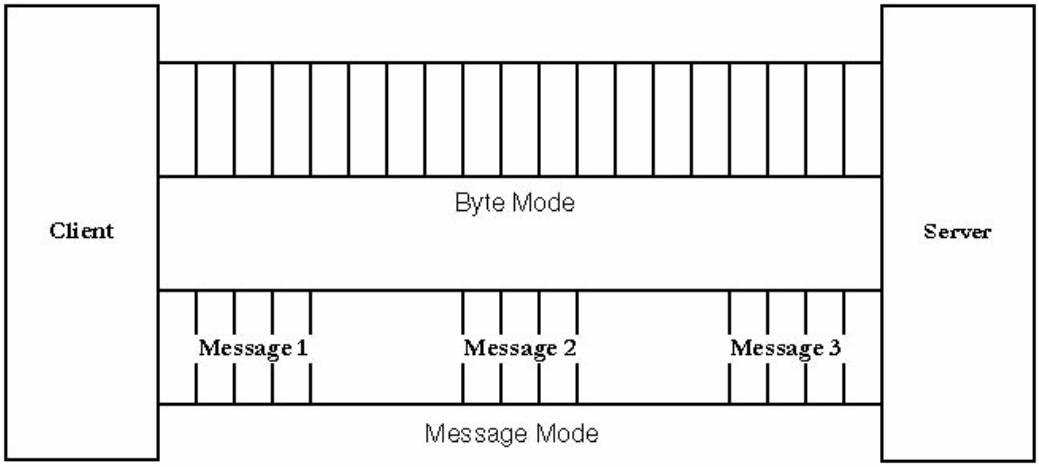


Figure 20-1 *Byte mode and message mode*

20.1.3 Compiling Applications

When you build a named pipe client or server application using Microsoft Visual C++, your application must include the WINBASE.H file in your program files. If your application includes WINDOWS.H—as most do—you can omit WINBASE.H. Your application is also responsible for linking with KERNEL32.LIB, which typically is configured with the Visual C++ linker flags.

20.1.4 Error Codes

All Windows API functions (except CreateFile and CreateNamedPipe) that are used in developing named pipe client and server applications return the value 0 when they fail. CreateFile and CreateNamedPipe return INVALID_HANDLE_VALUE. When either of these functions fails, applications should call the GetLastError function to retrieve specific information about the failure. For a complete list of error codes, consult the header file WINERROR.H.

20.2 Basic Server and Client

Named pipes feature a simple client/server design architecture in which data can flow in both a unidirectional and a bidirectional manner between a client and server. This is useful because it allows you to send and receive data whether your application is a client or a server. The main difference between a named pipe server and a client application is that a named pipe server is the only process capable of creating a named pipe and accepting pipe client connections. A client application is capable only of connecting to an existing named pipe server. Once a connection is formed between a client application and a server application, both processes are capable of reading and writing data on a pipe using standard Windows functions such as ReadFile and WriteFile. Note that a named pipe server application can operate only on the Windows NT platform—Windows 95, Windows 98, and Windows Me systems do not permit applications to create a named pipe. This limitation makes it impossible to form communications directly between two Windows 95, Windows 98, or Windows Me computers. However, Windows 95, Windows 98, and Windows Me clients can form connections to Windows NT–based computers.

20.2.1 Server Details

Implementing a named pipe server requires developing an application to create one or more named pipe instances, which can be accessed by clients. To a server, a pipe instance is nothing more than a handle used to accept a connection from a local or remote client application. The following steps describe how to write a basic server application:

1. Create a named pipe instance handle using the CreateNamedPipe API function.
2. Use the ConnectNamedPipe API function to listen for a client connection on the named pipe instance.
3. Receive data from and send data to the client using the ReadFile and WriteFile API functions.

4. Close down the named pipe connection using the DisconnectNamed Pipe API function.
5. Close the named pipe instance handle using the CloseHandle API function.

First, your server process needs to create a named pipe instance using the CreateNamedPipe API call, which is defined as follows:

```
HANDLE CreateNamedPipe(
    LPCTSTR lpName,
    DWORD dwOpenMode,
    DWORD dwPipeMode,
    DWORD nMaxInstances,
    DWORD nOutBufferSize,
    DWORD nInBufferSize,
    DWORD nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

The first parameter, lpName, specifies the name of a named pipe. The name must have the following UNC form:

```
\\.\Pipe\[path]name
```

Notice that the server name is represented as a dot, which represents the local machine. You cannot create a named pipe on a remote computer. The [path]name part of the parameter must represent a unique name. This might simply be a filename, or it might be a full directory path followed by a filename.

The dwOpenMode parameter describes the directional, I/O control, and security modes of a pipe when it is created. Table 20-1 describes all the available flags that can be used. A pipe can be created using a combination of these flags by ORing them together.

Table 20-1 *Named Pipe Open Mode Flags*

Open Mode	Flags	Description
Directional	PIPE_ACCESS_DUPLEX	The pipe is bidirectional: Both the server and client processes can read from and write data to the pipe.
	PIPE_ACCESS_OUTBOUND	The flow of data in the pipe goes from server to client only.
	PIPE_ACCESS_INBOUND	The flow of data in the pipe goes from client to server only.

I/O Control	FILE_FLAG_WRITE_THROUGH	Works only for byte-mode pipes. Functions writing to a named pipe do not return until the data written is transmitted across the network and is in the pipe's buffer on the remote computer.
I/O control	FILE_FLAG_OVERLAPPED	Allows functions that perform read, write, and connect operations to use overlapped I/O.
Security	WRITE_DAC	Allows your application to have write access to the named pipe's DACL.
Security	ACCESS_SYSTEM_SECURITY	Allows your application to have write access to the named pipe's SACL.
	WRITE_OWNER	Allows your application to have write access to the named pipe's owner and group SID.

The PIPE_ACCESS_ flags determine flow direction on a pipe between a client and a server. A pipe can be opened as bidirectional (two-way) using the PIPE_ACCESS_DUPLEX flag: Data can flow in both directions between the client and the server. In addition, you can also control the direction of data flow by opening the pipe as unidirectional (one-way) using the flag PIPE_ACCESS_INBOUND or PIPE_ACCESS_OUTBOUND: data can flow only one way from the client to the server or vice versa. Figure 20-2 describes the flag combinations further and shows the flow of data between a client and a server.

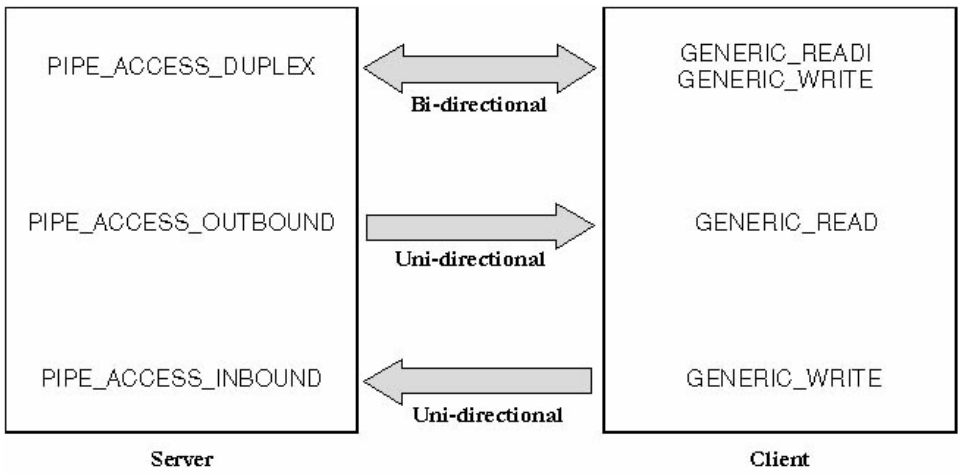


Figure 20-2 *Mode flags and flow direction*

The next set of dwOpenMode flags controls I/O behavior on a named pipe from the server's perspective. The FILE_FLAG_WRITE_THROUGH flag controls the write

operations so that functions writing to a named pipe do not return until the data written is transmitted across the network and is in the pipe's buffer on the remote computer. This flag works only for byte-mode named pipes when the client and the server are on different computers. The FILE_FLAG_OVERLAPPED flag allows functions performing read, write, and connect operations to return immediately, even if those functions take significant time to complete. We discuss the details of overlapped I/O when we develop an advanced server later in this chapter.

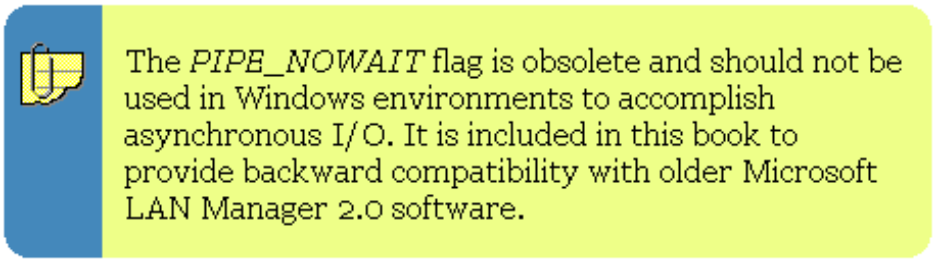
The last set of dwOpenMode flags described in Table 20-1 controls the server's ability to access the security descriptor that is created by a named pipe. If your application needs to modify or update the pipe's security descriptor after the pipe is created, you should set these flags accordingly to permit access. The WRITE_DAC flag allows your application to update the pipe's DACL, whereas ACCESS_SYSTEM_SECURITY allows access to the pipe's SACL. The WRITE_OWNER flag allows you to change the pipe's owner and group SID. For example, if you want to deny access to a particular user who has access rights to your pipe, you can modify the pipe's DACL using security API functions. Chapter 18 discusses DACLs, SACLs, and SIDs in greater detail.

CreateNamedPipe's dwPipeMode parameter specifies the read, write, and wait operating modes of a pipe. Table 20-2 describes all the available mode flags that can be used. The flags can be issued by ORing one flag from each mode category. If a pipe is opened as byte-oriented using the PIPE_READMODE_BYTE | PIPE_TYPE_BYTE mode flags, data can be read and written only as a stream of bytes. This means that when you read and write data to a pipe, you do not have to balance each read and write because your data does not have any message boundaries. For example, if a sender writes 500 bytes to a pipe, a receiver might want to read 100 bytes at a time until it receives all of the data. To establish clear boundaries around messages, place the pipe in message-oriented mode using the flags PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE, meaning each read and write must be balanced. For example, if a sender writes a 500-byte message to a pipe, the receiver must provide the ReadFile function a 500-byte or larger buffer when reading data. If the receiver fails to do so, ReadFile will fail with error ERROR_MORE_DATA. You can also combine PIPE_TYPE_MESSAGE with PIPE_READMODE_BYTE, allowing a sender to write messages to a pipe and the receiver to read an arbitrary amount of bytes at a time. The message delimiters will be ignored in the data stream. You cannot mix the PIPE_TYPE_BYTE flag with the PIPE_READMODE_MESSAGE flag. Doing so will cause the CreateNamedPipe function to fail with the error ERROR_INVALID_PARAMETER because no message delimiters are in the I/O stream when data is written into the pipe as bytes. The PIPE_WAIT or PIPE_NOWAIT flag can also be combined with read and write mode flags. The PIPE_WAIT flag places a pipe in blocking mode and the PIPE_NOWAIT flag places a pipe in nonblocking mode. In blocking mode, I/O operations such as ReadFile block until the I/O request is complete. This is the default behavior if you do not specify any flags. The nonblocking mode flag PIPE_NOWAIT

is designed to allow I/O operations to return immediately. However, it should not be used to achieve asynchronous I/O in Windows applications. It is included to provide backward compatibility with older Microsoft LAN Manager 2.0 applications. The ReadFile and WriteFile functions allow applications to accomplish asynchronous I/O using Windows overlapped I/O, which is demonstrated later in this chapter.

Table 20-2 *Named Pipe Read/Write Mode Flags*

Mode	Flags	Description
Write	PIPE_TYPE_BYTE	Data is written to the pipe as a stream of bytes.
	PIPE_TYPE_MESSAGE	Data is written to the pipe as a stream of messages.
Read	PIPE_READMODE_BYTE	Data is read from the pipe as a stream of bytes.
	PIPE_READMODE_MESSAGE	Data is read from the pipe as a stream of messages.
Wait	PIPE_WAIT	Blocking mode is enabled.
	PIPE_NOWAIT	Nonblocking mode is enabled.



The nMaxInstances parameter specifies how many instances or pipe handles can be created for a named pipe. A pipe instance is a connection from a local or remote client application to a server application that created the named pipe. Acceptable values are in the range 1 through PIPE_UNLIMITED_INSTANCES. For example, if you want to develop a server that can service only five client connections at a time, set this parameter to 5. If you set this parameter to PIPE_UNLIMITED_INSTANCES, the number of pipe instances that can be created is limited only by the availability of system resources.

CreateNamedPipe's nOutBufferSize and nInBufferSize parameters represent the number of bytes to reserve for internal input and output buffer sizes. These sizes are advisory in that every time a named pipe instance is created, the system sets up inbound and/or outbound buffers using the nonpaged pool (the physical memory used by the operating system). The buffer size specified should be reasonable (not too large) so that your system will not run out of nonpaged pool memory, but it should also be large enough to accommodate typical I/O requests. If an application attempts to write data that is larger than the buffer sizes specified, the system tries to automatically

expand the buffers to accommodate the data using nonpaged pool memory. For practical purposes, applications should size these internal buffers to match the size of the application's send and receive buffers used when calling ReadFile and WriteFile.

The nDefaultTimeOut parameter specifies the default timeout value (how long a client will wait to connect to a named pipe) in milliseconds. This affects only client applications that use the WaitNamedPipe function to determine when an instance of a named pipe is available to accept connections. We discuss this concept in greater detail later in this chapter, when we develop a named pipe client application.

The lpSecurityAttributes parameter allows the application to specify a security descriptor for a named pipe and determines whether a child process can inherit the newly created handle. If this parameter is specified as NULL, the named pipe gets a default security descriptor and the handle cannot be inherited. A default security descriptor grants the named pipe the same security limits and access controls as the process that created it following the Windows NT platform security model described in Chapter 18. An application can apply access control restrictions to a pipe by setting access privileges for particular users and groups in a SECURITY_DESCRIPTOR structure using security API functions. If a server wants to open access to any client, you should assign a null DACL to the SECURITY_DESCRIPTOR structure.

After you successfully receive a handle from CreateNamedPipe, which is known as a pipe instance, you have to wait for a connection from a named pipe client. This connection can be made through the ConnectNamedPipe API function, which is defined as follows:

```
BOOL ConnectNamedPipe(
      HANDLE hNamedPipe,
      LPOVERLAPPED lpOverlapped
);
```

The hNamedPipe parameter represents the pipe instance handle returned from CreateNamedPipe. The lpOverlapped parameter allows this API function to operate asynchronously, or in nonblocking mode, if the pipe was created using the FILE_FLAG_OVERLAPPED flag, which is known as Windows overlapped I/O. If this parameter is specified as NULL, ConnectNamedPipe blocks until a client forms a connection to the server. We discuss overlapped I/O in greater detail when you learn to create a more advanced named pipe server later in this chapter.

Once a named pipe client successfully connects to your server, the ConnectNamedPipe API call completes. The server is then free to send data to a client using the WriteFile API function and to receive data from the client using ReadFile. Once the server has finished communicating with a client, it should call DisconnectNamedPipe to close the communication session. The following sample

demonstrates how to write a simple server application that can communicate with one client.

```
// Server.cpp

#include <windows.h>
#include <stdio.h>

void main(void)
{
	HANDLE PipeHandle;
	DWORD BytesRead;
	CHAR buffer[256];
	if ((PipeHandle = CreateNamedPipe("\\\\.\\Pipe\\Jim",
		PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE |
PIPE_READMODE_BYTE, 1,e
		0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
	{
		printf("CreateNamedPipe failed with error %d\n",
			GetLastError());
		return;
	}

	printf("Server is now running\n");

	if (ConnectNamedPipe(PipeHandle, NULL) == 0)
	{
		printf("ConnectNamedPipe failed with error %d\n",
			GetLastError());
		CloseHandle(PipeHandle);
		return;
	}

	if (ReadFile(PipeHandle, buffer, sizeof(buffer),
		&BytesRead,    NULL) <= 0)
	{
		printf("ReadFile failed with error %d\n", GetLastError());
		CloseHandle(PipeHandle);
		return;
	}

	printf("%.*s\n", BytesRead, buffer);

	if (DisconnectNamedPipe(PipeHandle) == 0)
```

```
        {
            printf("DisconnectNamedPipe failed with error %d\n",
                GetLastError());
            return;
        }

        CloseHandle(PipeHandle);
}
```

20.2.2 Building Null DACLs

When applications create securable objects such as files and named pipes on the Windows NT platform using Windows API functions, the operating system grants the applications the ability to set up access control rights by specifying a SECURITY_ATTRIBUTES structure, defined as follows:

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD      nLength;
    LPVOID     lpSecurityDescriptor;
    BOOL       bInheritHandle
} SECURITY_ATTRIBUTES;
```

The lpSecurityDescriptor field defines the access rights for an object in a SECURITY_DESCRIPTOR structure. A SECURITY_DESCRIPTOR structure contains a DACL field that defines which users and groups can access the object. If you set this field to NULL, any user or group can access your resource.

Applications cannot directly access a SECURITY_DESCRIPTOR structure and must use Windows security API functions to do so. If you want to assign a null DACL to a SECURITY_DESCRIPTOR structure, you must do the following:

1. Create and initialize a SECURITY_DESCRIPTOR structure by calling the InitializeSecurityDescriptor API function.
2. Assign a null DACL to the SECURITY_DESCRIPTOR structure by calling the SetSecurityDescriptorDacl API function.

After you successfully build a new SECURITY_DESCRIPTOR structure, you must assign it to the SECURITY_ATTRIBUTES structure. Now you are ready to begin calling Windows functions such as CreateNamedPipe with your new SECURITY_ATTRIBUTES structure, which contains a null DACL. The following code fragment demonstrates how to call the security API functions needed to accomplish this:

```
// Create new SECURITY_ATTRIBUTES and SECURITY_DESCRIPTOR
```

```
// structure objects
SECURITY_ATTRIBUTES sa;
SECURITY_DESCRIPTOR sd;

// Initialize the new SECURITY_DESCRIPTOR object to empty values
if (InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION)
      == 0)
{
      printf("InitializeSecurityDescriptor failed with error %d\n",
            GetLastError());
      return;
}

// Set the DACL field in the SECURITY_DESCRIPTOR object to NULL
if (SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE) == 0)
{
      printf("SetSecurityDescriptorDacl failed with error %d\n",
            GetLastError());
      return;
}

// Assign the new SECURITY_DESCRIPTOR object to the
// SECURITY_ATTRIBUTES object
sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.lpSecurityDescriptor = &sd;
sa.bInheritHandle = TRUE;
```

20.2.3 Advanced Server

The previous sample demonstrates how to develop a named pipe server application that handles only a single pipe instance. All of the API calls operate in a synchronous mode in which each call waits until an I/O request is complete. A named pipe server is also capable of having multiple pipe instances so that clients can form two or more connections to the server; the number of pipe instances is limited by the number specified in the nMaxInstances parameter of the CreateNamedPipe API call. To handle more than one pipe instance, a server must consider using multiple threads or asynchronous Windows I/O mechanisms—such as overlapped I/O and completion ports—to service each pipe instance. Asynchronous I/O mechanisms allow a server to service all pipe instances simultaneously from a single application thread. Our discussion demonstrates how to develop advanced servers using threads and overlapped I/O. See Chapter 5 for more information on completion ports as they apply to Windows sockets.

20.2.4 Threads

Developing an advanced server that can support more than one pipe instance using threads is simple. All you need to do is create one thread for each pipe instance and service each instance using the techniques we described earlier for the simple server. The following sample demonstrates a server that is capable of serving five pipe instances. The application is an echo server that reads data from a client and echoes the data back.

```
// Threads.cpp

#include <windows.h>
#include <stdio.h>
#include <conio.h>

#define NUM_PIPES 5

DWORD WINAPI PipeInstanceProc(LPVOID lpParameter);

void main(void)
{
	HANDLE ThreadHandle;
	INT i;
	DWORD ThreadId;

	for(i = 0; i < NUM_PIPES; i++)
	{
		// Create a thread to serve each pipe instance
		if ((ThreadHandle = CreateThread(NULL, 0, PipeInstanceProc,
			NULL, 0, &ThreadId)) == NULL)
		{
			printf("CreateThread failed with error %\n",
				GetLastError());
			return;
		}
		CloseHandle(ThreadHandle);
	}

	printf("Press a key to stop the server\n");
	_getch();
}

//
// Function: PipeInstanceProc
```

```
//
// Description:
//       This function handles the communication details of a single
//       named pipe instance
//
DWORD WINAPI PipeInstanceProc(LPVOID lpParameter)
{
      HANDLE PipeHandle;
      DWORD BytesRead;
      DWORD BytesWritten;
      CHAR Buffer[256];

      if ((PipeHandle = CreateNamedPipe("\\\\.\\PIPE\\jim",
            PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE |
PIPE_READMODE_BYTE,
            NUM_PIPES, 0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
      {
            printf("CreateNamedPipe failed with error %d\n",
                  GetLastError());
            return 0;
      }

      // Serve client connections forever
      while(1)
      {
            if (ConnectNamedPipe(PipeHandle, NULL) == 0)
            {
                  printf("ConnectNamedPipe failed with error %d\n",
                        GetLastError());
                  break;
            }

            // Read data from and echo data to the client until
            // the client is ready to stop
            while(ReadFile(PipeHandle, Buffer, sizeof(Buffer),
                  &BytesRead,    NULL) > 0)
            {
                  printf("Echo %d bytes to client\n", BytesRead);

                  if (WriteFile(PipeHandle, Buffer, BytesRead,
                        &BytesWritten, NULL) == 0)
                  {
                        printf("WriteFile failed with error %d\n",
                              GetLastError());
```

```
                    break;
                }
            }

            if (DisconnectNamedPipe(PipeHandle) == 0)
            {
                printf("DisconnectNamedPipe failed with error %d\n",
                    GetLastError());
                break;
            }
        }

        CloseHandle(PipeHandle);
        return 0;
}
```

To develop your server to handle five pipe instances, start by calling the CreateThread API function. CreateThread starts five execution threads, all of which execute the PipeInstanceProc function simultaneously. The PipeInstanceProc function operates exactly like the basic server application (the previous sample) except that it reuses a named pipe handle by calling the DisconnectNamedPipe API function, which closes a client's session to the server. Once an application calls DisconnectNamedPipe, it is free to service another client by calling the ConnectNamedPipe function with the same pipe instance handle.

20.2.5 Overlapped I/O

Overlapped I/O is a mechanism that allows Windows API functions such as ReadFile and WriteFile to operate asynchronously when I/O requests are made. This is accomplished by passing an OVERLAPPED structure to these API functions and later retrieving the results of an I/O request through the original OVERLAPPED structure using the GetOverlappedResult API function. When a Windows API function is invoked with an overlapped structure, the call returns immediately.

To develop an advanced named pipe server that can manage more than one named pipe instance using overlapped I/O, you need to call CreateNamedPipe with the nMaxInstances parameter set to a value greater than 1. You also must set the dwOpenMode flag to FILE_FLAG_OVERLAPPED. The next sample demonstrates how to develop this advanced named pipe server. The application is an echo server that reads data from a client and writes the data back.

```
// Overlap.cpp

#include <windows.h>
```

```
#include <stdio.h>

#define NUM_PIPES 5
#define BUFFER_SIZE 256

void main(void)
{
	HANDLE PipeHandles[NUM_PIPES];
	DWORD BytesTransferred;
	CHAR Buffer[NUM_PIPES][BUFFER_SIZE];
	INT i;
	OVERLAPPED Ovlap[NUM_PIPES];
	HANDLE Event[NUM_PIPES];

	// For each pipe handle instance, the code must maintain the
	// pipes' current state, which determines if a ReadFile or
	// WriteFile is posted on the named pipe. This is done using
	// the DataRead variable array. By knowing each pipe's
	// current state, the code can determine what the next I/O
	// operation should be.
	BOOL DataRead[NUM_PIPES];

	DWORD Ret;
	DWORD Pipe;

	for(i = 0; i < NUM_PIPES; i++)
	{
		// Create a named pipe instance
		if ((PipeHandles[i] = CreateNamedPipe("\\\\.\\PIPE\\jim",
			PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
			PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, NUM_PIPES,
			0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
		{
			printf("CreateNamedPipe for pipe %d failed "
				"with error %d\n", i, GetLastError());
			return;
		}

		// Create an event handle for each pipe instance. This
		// will be used to monitor overlapped I/O activity on
		// each pipe.
		if ((Event[i] = CreateEvent(NULL, TRUE, FALSE, NULL))
			== NULL)
		{
```

```
            printf("CreateEvent for pipe %d failed with error %d\n",
                  i, GetLastError());
            continue;
      }

      // Maintain a state flag for each pipe to determine when data
      // is to be read from or written to the pipe
      DataRead[i] = FALSE;

      ZeroMemory(&Ovlap[i], sizeof(OVERLAPPED));
      Ovlap[i].hEvent = Event[i];

      // Listen for client connections using ConnectNamedPipe()
      if (ConnectNamedPipe(PipeHandles[i], &Ovlap[i]) == 0)
      {
            if (GetLastError() != ERROR_IO_PENDING)
            {
                  printf("ConnectNamedPipe for pipe %d failed with"
                        " error %d\n", i, GetLastError());
                  CloseHandle(PipeHandles[i]);
                  return;
            }
      }
}

printf("Server is now running\n");


// Read and echo data back to Named Pipe clients forever
while(1)
{
      if ((Ret = WaitForMultipleObjects(NUM_PIPES, Event,
            FALSE, INFINITE)) == WAIT_FAILED)
      {
            printf("WaitForMultipleObjects failed with error %d\n",
                  GetLastError());
            return;
      }

      Pipe = Ret - WAIT_OBJECT_0;

      ResetEvent(Event[Pipe]);

      // Check overlapped results, and if they fail, reestablish
```

```
// communication for a new client; otherwise, process read
// and write operations with the client

if (GetOverlappedResult(PipeHandles[Pipe], &Ovlap[Pipe],
      &BytesTransferred, TRUE) == 0)
{
      printf("GetOverlapped result failed %d start over\n",
            GetLastError());

      if (DisconnectNamedPipe(PipeHandles[Pipe]) == 0)
      {
            printf("DisconnectNamedPipe failed with error %d\n",
                  GetLastError());
            return;
      }

      if (ConnectNamedPipe(PipeHandles[Pipe],
            &Ovlap[Pipe]) == 0)
      {
            if (GetLastError() != ERROR_IO_PENDING)
            {
                  // Severe error on pipe. Close this
                  // handle forever.
                  printf("ConnectNamedPipe for pipe %d failed with"
                        " error %d\n", i, GetLastError());
                  CloseHandle(PipeHandles[Pipe]);
            }
      }

      DataRead[Pipe] = FALSE;
}
else
{
      // Check the state of the pipe. If DataRead equals
      // FALSE, post a read on the pipe for incoming data.
      // If DataRead equals TRUE, then prepare to echo data
      // back to the client.

      if (DataRead[Pipe] == FALSE)
      {
            // Prepare to read data from a client by posting a
            // ReadFile operation

            ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
```

```
                    Ovlap[Pipe].hEvent = Event[Pipe];

                    if (ReadFile(PipeHandles[Pipe], Buffer[Pipe],
                        BUFFER_SIZE, NULL, &Ovlap[Pipe]) == 0)
                    {
                        if (GetLastError() != ERROR_IO_PENDING)
                        {
                            printf("ReadFile failed with error %d\n",
                            GetLastError());
                        }
                    }

                    DataRead[Pipe] = TRUE;
                }
                else
                {
                    // Write received data back to the client by
                    // posting a WriteFile operation
                    printf("Received %d bytes, echo bytes back\n",
                        BytesTransferred);

                    ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
                    Ovlap[Pipe].hEvent = Event[Pipe];

                    if (WriteFile(PipeHandles[Pipe], Buffer[Pipe],
                        BytesTransferred, NULL, &Ovlap[Pipe]) == 0)
                     {
                        if (GetLastError() != ERROR_IO_PENDING)
                        {
                            printf("WriteFile failed with error %d\n",
                            GetLastError());
                        }
                    }

                    DataRead[Pipe] = FALSE;
                }
            }
        }
}
```

For the server application to service five pipe instances at a time, it must call CreateNamedPipe five times to retrieve an instance handle for each pipe. After the server retrieves all the instance handles, it begins to listen for clients by calling ConnectNamedPipe asynchronously five times using an overlapped I/O structure for

each pipe. As clients form connections to the server, all I/O is processed asynchronously. When clients disconnect, the server reuses each pipe instance handle by calling DisconnectNamedPipe and reissuing a ConnectNamedPipe call.

20.2.6 Security Impersonation

One of the best reasons for using named pipes as a network programming solution is that they rely on Windows NT platform security features to control access when clients attempt to form communication to a server. Windows NT security offers security impersonation, which allows a named pipe server application to execute in the security context of a client. When a named pipe server executes, it normally operates at the security context permission level of the process that starts the application. For example, if a person with administrator privileges starts up a named pipe server, the server has the ability to access almost every resource on a Windows NT system. Such security access for a named pipe server is bad if the SECURITY_DESCRIPTOR structure specified in CreateNamedPipe allows all users to access your named pipe.

When a server accepts a client connection using the ConnectNamedPipe function, it can make its execution thread operate in the security context of the client by calling the ImpersonateNamedPipeClient API function, which is defined as follows:

```
BOOL ImpersonateNamedPipeClient(
      HANDLE hNamedPipe
);
```

The hNamedPipe parameter represents the pipe instance handle that is returned from CreateNamedPipe. When this function is called, the operating system changes the thread security context of the server to the security context of the client. This is quite handy: If your server is designed to access resources such as files, it will do so using the client's access rights, thereby allowing your server to preserve access control to resources regardless of who started the process.

When a server thread executes in a client's security context, it does so through a security impersonation level. There are four basic impersonation levels: anonymous, identification, impersonation, and delegation. Security impersonation levels govern the degree to which a server can act on behalf of a client. We discuss these impersonation levels in greater detail when we develop a client application later in this chapter. After the server finishes processing a client's session, it should call RevertToSelf to return to its original thread execution security context. The RevertToSelfAPI function is defined as follows:

```
BOOL RevertToSelf(VOID);
```

This function does not have any parameters.

20.2.7 Client Details

Implementing a named pipe client requires developing an application that forms a connection to a named pipe server. Clients cannot create named pipe instances. However, clients do open handles to preexisting instances from a server. The following steps describe how to write a basic client application:

1. Wait for a named pipe instance to become available using the WaitNamedPipe API function.
2. Connect to the named pipe using the CreateFile API function.
3. Send data to and receive data from the server using the WriteFile and ReadFile API functions.
4. Close the named pipe session using the CloseHandle API function.

Before forming a connection, clients need to check for the existence of a named pipe instance using the WaitNamedPipe function, which is defined as follows:

```
BOOL WaitNamedPipe(
    LPCTSTR lpNamedPipeName,
    DWORD nTimeOut
);
```

The lpNamedPipeName parameter represents the named pipe you are trying to connect to. The nTimeOut parameter represents how long a client is willing to wait for a pipe's server process to have a pending ConnectNamedPipe operation on the pipe.

After WaitNamedPipe successfully completes, the client needs to open a handle to the server's named pipe instance using the CreateFile API function. CreateFile is defined as follows:

```
HANDLE CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

The lpFileName parameter is the name of the pipe you are trying to open; the name must conform to the named pipe naming conventions mentioned earlier in this chapter.

The dwDesiredAccess parameter defines the access mode and should be set to GENERIC_READ for reading data off the pipe and GENERIC_WRITE for writing data to the pipe. These flags can also be specified together by ORing both flags. The access mode must be compatible with how the pipe was created in the server. Match the mode specified in the dwOpenMode parameter of CreateNamedPipe, as described earlier. For example, if the server creates a pipe with PIPE_ACCESS_INBOUND, the client should specify GENERIC_WRITE.

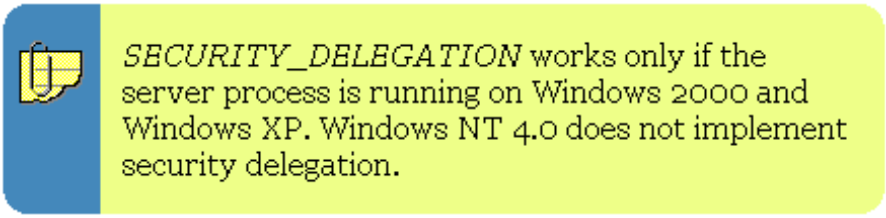
The dwShareMode parameter should be set to 0 because only one client is capable of accessing a pipe instance at a time. The lpSecurityAttributes parameter should be set to NULL unless you need a child process to inherit the client's handle. This parameter is incapable of specifying security controls because CreateFile is not capable of creating named pipe instances. The dwCreationDisposition parameter should be set to OPEN_EXISTING, which means that the CreateFile function will fail if the named pipe does not exist.

The dwFlagsAndAttributes parameter should always be set to FILE_ATTRIBUTE_NORMAL. Optionally, you can specify the FILE_FLAG_WRITE_THROUGH, FILE_FLAG_OVERLAPPED, and SECURITY_SQOS_PRESENT flags by ORing them with the FILE_ATTRIBUTE_NORMAL flag. The FILE_FLAG_WRITE_THROUGH and FILE_FLAG_OVERLAPPED flags behave like the server's mode flags described earlier in this chapter. The SECURITY_SQOS_PRESENT flag controls client impersonation security levels in a named pipe server. Security impersonation levels govern the degree to which a server process can act on behalf of a client process. A client can specify this information when it connects to a server. When the client specifies the SECURITY_SQOS_PRESENT flag, it must use one or more of the following security flags:

- SECURITY_ANONYMOUS. Specifies to impersonate the client at the anonymous impersonation security level. The server process cannot obtain identification information about the client, and it cannot execute in the security context of the client.
- SECURITY_IDENTIFICATION. Specifies to impersonate the client at the identification impersonation security level. The server process can obtain information about the client, such as security identifiers and privileges, but it cannot execute in the security context of the client. This is useful for named pipe clients that want to allow the server to identify the client but not to act as the client.
- SECURITY_IMPERSONATION. Specifies to impersonate the client at the impersonation security level. The client wants to allow the server process to

obtain information about the client and execute in the client's security context on the local system. Using this flag, the client allows the server to access any local resource on the server as the client. The server, however, cannot impersonate the client on remote systems.

- SECURITY_DELEGATION. Specifies to impersonate the client at the delegation impersonation security level. The server process can obtain information about the client and execute in the client's security context on its local system and on remote systems.



- SECURITY_CONTEXT_TRACKING. Specifies that the security-tracking mode is dynamic. If this flag is not specified, security-tracking mode is static.
- SECURTIY_EFFECTIVE_ONLY. Specifies that only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available.

Named pipe security impersonation is described earlier in this chapter in the section entitled “Server Details.”

The final parameter of CreateFile, hTemplateFile, does not apply to named pipes and should be specified as NULL. If CreateFile completes without an error, the client application can begin to send and receive data on the named pipe using the ReadFile and WriteFile functions. Once the application is finished processing data, it can close down the connection using the CloseHandle function.

The next program listing is a simple named pipe client that demonstrates the API calls needed to successfully develop a basic named pipe client application. When this application successfully connects to a named pipe, it writes the message “This is a test” to the server.

```
// Client.cpp

#include <windows.h>
#include <stdio.h>

#define PIPE_NAME "\\\\.\\Pipe\\jim"

void main(void)
{
```

```
    HANDLE PipeHandle;
    DWORD BytesWritten;

    if (WaitNamedPipe(PIPE_NAME, NMPWAIT_WAIT_FOREVER) == 0)
    {
        printf("WaitNamedPipe failed with error %d\n",
            GetLastError());
        return;
    }

    // Create the named pipe file handle
    if ((PipeHandle = CreateFile(PIPE_NAME,
        GENERIC_READ | GENERIC_WRITE, 0,
        (LPSECURITY_ATTRIBUTES) NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        (HANDLE) NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }

    if (WriteFile(PipeHandle, "This is a test", 14, &BytesWritten,
        NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        CloseHandle(PipeHandle);
        return;
    }

    printf("Wrote %d bytes", BytesWritten);

    CloseHandle(PipeHandle);
}
```

20.3 Other API Calls

There are several additional named pipe functions that we haven't touched on yet. The first set of these API functions—CallNamedPipe and TransactNamedPipe—is designed to reduce coding complexity in an application. Both functions perform a write and read operation in one call. The CallNamedPipe function allows a client application to connect to a message-type pipe (and waits if an instance of the pipe is not available), writes to and reads from the pipe, and then closes the pipe. This is practically an entire client application written in one call. CallNamedPipe is defined as follows:

```
BOOL CallNamedPipe(
   LPCTSTR lpNamedPipeName,
   LPVOID lpInBuffer,
   DWORD nInBufferSize,
   LPVOID lpOutBuffer,
   DWORD nOutBufferSize,
   LPDWORD lpBytesRead,
   DWORD nTimeOut
);
```

The lpNamedPipeName parameter is a string that represents the named pipe in UNC form. The lpInBuffer and nInBufferSize parameters represent the address and the size of the buffer that the application uses to write data to the server. The lpOutBuffer and nOutBufferSize parameters represent the address and the size of the buffer that the application uses to retrieve data from the server. The lpBytesRead parameter receives the number of bytes read from the pipe. The nTimeOut parameter specifies how many milliseconds to wait for the named pipe to be available.

The TransactNamedPipe function can be used in both a client and a server application. It is designed to combine read and write operations in one API call, thus optimizing network I/O by reducing send and receive transactions in the MSNP redirector. TransactNamedPipe is defined as follows:

```
BOOL TransactNamedPipe(
   HANDLE hNamedPipe,
   LPVOID lpInBuffer,
   DWORD nInBufferSize,
   LPVOID lpOutBuffer,
   DWORD nOutBufferSize,
   LPDWORD lpBytesRead,
   LPOVERLAPPED lpOverlapped
);
```

The hNamedPipe parameter identifies the named pipe returned by the CreateNamedPipe or CreateFile API functions. The lpInBuffer and nInBufferSize parameters represent the address and the size of the buffer that the application uses to write data to the pipe. The lpOutBuffer and nOutBufferSize parameters represent the address and the size of the buffer that the application uses to retrieve data from the pipe. The lpBytesRead parameter receives the number of bytes read from the pipe. The lpOverlapped parameter allows this TransactNamedPipe to operate asynchronously using overlapped I/O.

The next set of functions—GetNamedPipeHandleState, SetNamedPipeHandleState, and GetNamedPipeInfo—are designed to make named pipe client and server communication more flexible at run time. For example, you can use these functions to

change the operating mode of a pipe at run time from message mode to byte mode and vice versa. GetNamedPipeHandleState retrieves information such as the operating mode (message mode and byte mode), pipe instance count, and buffer caching information about a specified named pipe. The information that GetNamedPipeHandleState returns can vary during the lifetime of an instance of the named pipe. GetNamedPipeHandleState is defined as follows:

```
BOOL GetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpState,
    LPDWORD lpCurInstances,
    LPDWORD lpMaxCollectionCount,
    LPDWORD lpCollectDataTimeout,
    LPTSTR lpUserName,
    DWORD nMaxUserNameSize
);
```

The hNamedPipe parameter identifies the named pipe returned by the CreateNamedPipe or CreateFile function. The lpState parameter is a pointer to a variable that receives the current operating mode of the pipe handle. The lpState parameter can return the value PIPE_NOWAIT or the value PIPE_READMODE_MESSAGE. The lpCurInstances parameter is a pointer to a variable that receives the number of current pipe instances. The lpMaxCollectionCount parameter receives the maximum number of bytes to be collected on the client's computer before transmission to the server. The lpCollectDataTimeout parameter receives the maximum time in milliseconds that can pass before a remote named pipe transfers information over a network. The lpUserName and nMaxUserNameSize parameters represent a buffer that receives a null-terminated string containing the user name string of the client application.

The SetNamedPipeHandleState function allows you to change the pipe characteristics retrieved with GetNamedPipeHandleState. SetNamedPipeHandleState is defined as follows:

```
BOOL SetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpMode,
    LPDWORD lpMaxCollectionCount,
    LPDWORD lpCollectDataTimeout
);
```

The hNamedPipe parameter identifies the named pipe returned by CreateNamedPipe or CreateFile. The lpMode parameter sets the operating mode of a pipe. The lpMaxCollectionCount parameter specifies the maximum number of bytes collected on the client computer before data is transmitted to the server. The

lpCollectDataTimeout parameter specifies the maximum time in milliseconds that can pass before a remote named pipe client transfers information over the network.

The GetNamedPipeInfo API function is used to retrieve buffer size and maximum pipe instance information. GetNamedPipeInfo is defined as follows:

```
BOOL GetNamedPipeInfo(
   HANDLE hNamedPipe,
   LPDWORD lpFlags,
   LPDWORD lpOutBufferSize,
   LPDWORD lpInBufferSize,
   LPDWORD lpMaxInstances
);
```

The hNamedPipe parameter identifies the named pipe returned by Create-NamedPipe or CreateFile. The lpFlags parameter retrieves the type of the named pipe and determines whether it is a server or a client and whether the pipe is in byte mode or message mode. The lpOutBufferSize parameter determines the size in bytes of the internal buffer for outgoing data. The lpInBufferSize parameter receives the size of the internal buffer for incoming data. The lpMaxInstance parameter receives the maximum number of pipe instances that can be created.

The final API function, PeekNamedPipe, allows an application to look at the data in a named pipe without removing it from the pipe's internal buffer. This function is useful if an application wants to poll for incoming data to avoid blocking on the ReadFile API call. The function can also be useful for applications that need to examine data before they actually receive it. For example, an application might want to adjust its application buffers based on the size of incoming messages. PeekNamedPipe is defined as follows:

```
BOOL PeekNamedPipe(
   HANDLE hNamedPipe,
   LPVOID lpBuffer,
   DWORD nBufferSize,
   LPDWORD lpBytesRead,
   LPDWORD lpTotalBytesAvail,
   LPDWORD lpBytesLeftThisMessage
);
```

The hNamedPipe parameter identifies the named pipe returned by Create-NamedPipe or CreateFile. The lpBuffer and nBufferSize parameters represent the receiving buffer along with the receiving buffer size to retrieve data from the pipe. The lpBytesRead parameter receives the number of bytes read from the pipe into the lpBuffer parameter. The lpTotalBytesAvail parameter receives the total number of bytes that are available to be read from the pipe. The lpBytesLeftThisMessage parameter receives the number

of bytes remaining in a message if a pipe is opened in message mode. If a message cannot fit in the lpBuffer parameter, the remaining bytes in a message are returned. This parameter always returns 0 for byte-mode named pipes.

20.4 Platform and Performance Considerations

The Microsoft Knowledge Base documents the following problems and limitations. You can access the Knowledge Base at http://support.microsoft.com/support. The following are brief descriptions of each issue.

- **Q100291 Restriction on Named-Pipe Names**

  If a pipe named \\.\Pipe\Mypipes is created, it is not possible to subsequently create a pipe named \\.\Pipe\Mypipes\Pipe1, because \\.\Pipe\Mypipes is already a pipe name and cannot be used as a subdirectory.

- **Q119218 Named Pipe Write Limited to 64K**

  The WriteFile API function returns FALSE and GetLastError returns ERROR_MORE_DATA when WriteFile writes to a message-mode named pipe using a buffer greater than 64 KB.

- **Q110148 ERROR_INVALID_PARAMETER from WriteFile or ReadFile**

  The WriteFile or ReadFile function call can fail with the error ERROR_INVALID_PARAMETER if you are operating on a named pipe and using overlapped I/O. A possible cause for the failure is that the Offset and OffsetHigh members of the OVERLAPPED structure are not set to 0.

- **Q180222 WaitNamedPipe and Error 253 in Windows 95**

  In Windows 95, when WaitNamedPipe fails because of an invalid pipe name passed as the first parameter, GetLastError returns Error 253, which is not listed as a possible error code for this function. When you run the same code on Windows NT 4, the error code 161 (ERROR_BAD_PATHNAME) appears. To work around the problem, resolve Error 253 the same way as Error 161, ERROR_BAD_PATHNAME.

- **Q141709 Limit of 49 Named Pipe Connections from a Single Workstation**

  If a named pipe server creates more than 49 distinctly named pipes, a single client on a remote computer cannot connect more than 49 pipes on the named pipe server.

- **Q126645 Access Denied When Opening a Named Pipe from a Service**

  If a service running in the Local System account attempts to open a named pipe on a computer running Windows NT, the operation can fail with an Access Denied error (Error 5).

20.5 Conclusion

This chapter introduced the named pipe networking technology, which provides a simple client/server data-communication architecture that reliably transmits data. The interface relies on the Windows redirector to transmit data over a network. A major benefit of named pipes is that it takes advantage of Windows NT platform security features—an advantage offered by no other networking technology described in this book.