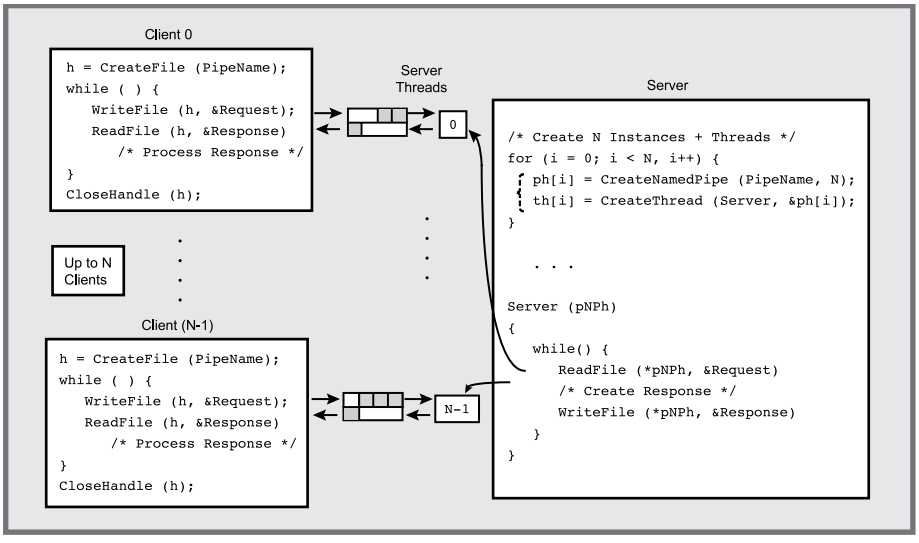

Figure 11–2 Clients and Servers Using Named Pipes

Normally, the creating process is regarded as the *server*. *Client* processes, possibly on other systems, open the pipe with `CreateFile`.

Figure 11–2 shows an illustrative client/server relationship, and the pseudocode shows one scheme for using named pipes. Notice that the server creates multiple instances of the same pipe, each of which can support a client. The server also creates a thread for each named pipe instance, so that each client has a dedicated thread and named pipe instance. Figure 11–2, then, shows how to implement the multithreaded server model of Figure 7–1.

Creating Named Pipes

Here is the specification of the `CreateNamedPipe` function.

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

Parameters

`lpName` indicates the pipe name, which must be of the form:

```
\\.\pipe\pipename
```

The period (`.`) stands for the local machine; thus, you cannot create a pipe on a remote machine. The pipename is case-insensitive, can be up to 256 characters long, and can contain any character other than backslash.

`dwOpenMode` specifies several flags; the important ones for our purposes are:

- One of three mutually exclusive data flow description flags—`PIPE_ACCESS_DUPLEX`, `PIPE_ACCESS_INBOUND`, or `PIPE_ACCESS_OUTBOUND`. The value determines the combination of `GENERIC_READ` and `GENERIC_WRITE` from the server's perspective. Thus, `PIPE_ACCESS_INBOUND` gives the server `GENERIC_READ` access, and the client must use `GENERIC_WRITE` when connecting with `CreateFile`. If the access is `PIPE_ACCESS_DUPLEX`, data flows bidirectionally, and the client can specify `GENERIC_READ`, `GENERIC_WRITE`, or both.
- `FILE_FLAG_OVERLAPPED` enables asynchronous I/O (Chapter 14).

The mode can also specify `FILE_FLAG_WRITE_THROUGH` (not used with message pipes), `FILE_FLAG_FIRST_PIPE_INSTANCE`, and more (see MSDN).

`dwPipeMode` has three mutually exclusive flag pairs. They indicate whether writing is message-oriented or byte-oriented, whether reading is by messages or blocks, and whether read operations block.

- `PIPE_TYPE_BYTE` and `PIPE_TYPE_MESSAGE` indicate whether data is written to the pipe as a stream of bytes or messages. Use the same type value for all pipe instances with the same name.
- `PIPE_READMODE_BYTE` and `PIPE_READMODE_MESSAGE` indicate whether data is read as a stream of bytes or messages. `PIPE_READMODE_MESSAGE` requires `PIPE_TYPE_MESSAGE`.
- `PIPE_WAIT` and `PIPE_NOWAIT` determine whether `ReadFile` will block. Use `PIPE_WAIT` because there are better ways to achieve asynchronous I/O.

`nMaxInstances` determines the maximum number of pipe instances. As Figure 11–2 shows, use this same value for every `CreateNamedPipe` call for a given pipe. Use the value `PIPE_UNLIMITED_INSTANCES` to have Windows base the number on available computer resources.

`nOutBufferSize` and `nInBufferSize` give the sizes, in bytes, of the input and output buffers used for the named pipes. Specify `0` to get default values.

`nDefaultTimeOut` is a default time-out period (in milliseconds) for the `WaitNamedPipe` function, which is discussed in an upcoming section. This situation, in which the create function specifies a time-out for a related function, is unique.

The error return value is `INVALID_HANDLE_VALUE` because pipe handles are similar to file handles.

`lpSecurityAttributes` operates as in all the other create functions.

The first `CreateNamedPipe` call actually creates the named pipe and an instance. Closing the last handle to an instance will delete the instance (usually, there is only one handle per instance). Closing the last instance of a named pipe will delete the pipe, making the pipe name available for reuse.

Named Pipe Client Connections

Figure 11–2 shows that a client connects to a named pipe using `CreateFile` with the pipe name. In many cases, the client and server are on the same machine, and the name would take this form:

```
\\.\pipe\[path]pipename
```

If the server is on a different machine, the name would take this form:

```
\\servername\pipe\pipename
```

Using the name period (`.`) when the server is local—rather than using the local machine name—delivers significantly better connection-time performance.

Named Pipe Status Functions

There are seven functions to interrogate pipe status information, and an eighth sets state information. They are mentioned briefly, and Program 11–3 demonstrates several of the functions.

- `GetNamedPipeHandleState` returns information, given an open handle, on whether the pipe is in blocking or nonblocking mode, whether it is message-oriented or byte-oriented, the number of pipe instances, and so on.
- `SetNamedPipeHandleState` allows the program to set the same state attributes. The mode and other values are passed by address rather than by value, which is necessary so that a `NULL` value specifies that the mode should not be changed. See the full *Examples* code of Program 11–2 for an example.

- GetNamedPipeInfo determines whether the handle is for a client or server instance, the buffer sizes, and so on.
- Five functions get information about the client name and the client and server session ID and process ID. Representative names are GetNamedPipeClientSessionId and GetNamedPipeServerProcessId.

Named Pipe Connection Functions

The server, after creating a named pipe instance, can wait for a client connection (CreateFile or CallNamedPipe, described in a subsequent function) using ConnectNamedPipe.

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

With lpOverlapped set to NULL, ConnectNamedPipe will return as soon as there is a client connection. Normally, the return value is TRUE. However, it would be FALSE if the client connected between the server's CreateNamedPipe call and the ConnectNamedPipe call. In this case, GetLastError returns ERROR_PIPE_CONNECTED, and the connection is valid despite the FALSE return value.

Following the return from ConnectNamedPipe, the server can read requests using ReadFile and write responses using WriteFile. Finally, the server should call DisconnectNamedPipe to free the handle (pipe instance) for connection with another client.

WaitNamedPipe, the final function, is for use by the client to synchronize connections to the server. The call will return successfully as soon as the server has a pending ConnectNamedPipe call. By using WaitNamedPipe, the client can be certain that the server is ready for a connection and the client can then call CreateFile. Nonetheless, the client's CreateFile call could fail if some other client opens the named pipe using CreateFile or if the server closes the instance handle; that is, there is a race involving the server and the clients. The server's ConnectNamedPipe call will not fail. Notice that there is a time-out period for WaitNamedPipe that, if specified, will override the time-out period specified with the server's CreateNamedPipe call.

Client and Server Named Pipe Connection

The proper connection sequences for the client and server are as follows. First is the server sequence, in which the server makes a client connection, communicates with the client until the client disconnects (causing `ReadFile` to return `FALSE`), disconnects the server-side connection, and then connects to another client.

```
/* Named pipe server connection sequence. */
hNp = CreateNamedPipe ("\\\\.\\pipe\\my_pipe", ...);
while (... /* Continue until server shuts down. */) {
      ConnectNamedPipe (hNp, NULL);
      while (ReadFile (hNp, Request, ...) {
         ...
         WriteFile (hNp, Response, ...);
      }
      DisconnectNamedPipe (hNp);
}
CloseHandle (hNp);
```

The client connection sequence is as follows, where the client terminates after it finishes, allowing another client to connect on the same named pipe instance. As shown, the client can connect to a networked server if it knows the server name.

```
/* Named pipe client connection sequence. */
WaitNamedPipe ("\\\\ServerName\\pipe\\my_pipe",
         NMPWAIT_WAIT_FOREVER);
hNp =
         CreateFile ("\\\\ServerName\\pipe\\my_pipe", ...);
while (... /* Run until there are no more requests. */ {
      WriteFile (hNp, Request, ...);
      ...
      ReadFile (hNp, Response);
}
CloseHandle (hNp); /* Disconnect from the server. */
```

Notice the race conditions between the client and the server. First, the client's `WaitNamedPipe` call will fail if the server has not yet created the named pipe; the failure test is omitted for brevity but is included in the sample programs in the *Examples* file. Next, the client may, in rare circumstances, complete its `CreateFile` call before the server calls `ConnectNamedPipe`. In that case, `ConnectNamedPipe` will return `FALSE` to the server, but the named pipe communication will still function properly.

The named pipe instance is a global resource, so once the client disconnects, another client can connect with the server.

Named Pipe Transaction Functions

Figure 11–2 shows a typical client configuration in which the client does the following:

- Opens an instance of the pipe, creating a long-lived connection to the server and consuming a pipe instance
- Repetitively sends requests and waits for responses
- Closes the connection

The common `WriteFile`, `ReadFile` sequence could be regarded as a single client transaction, and Windows provides such a function for message pipes.

```
BOOL TransactNamedPipe (
   HANDLE hNamedPipe,
   LPVOID lpWriteBuf,
   DWORD cbWriteBuf,
   LPVOID lpReadBuf,
   DWORD cbReadBuf,
   LPDWORD lpcbRead,
   LPOVERLAPPED lpOverlapped)
```

The parameter usage is clear because this function combines `WriteFile` and `ReadFile` on the named pipe handle. Both the output and input buffers are specified, and `*lpcbRead` receives the message length. Overlapped operations (Chapter 14) are possible. More typically, the function waits for the response.

`TransactNamedPipe` is convenient, but, as in Figure 11–2, it requires a permanent connection, which limits the number of clients.[3]

`CallNamedPipe` is the second client convenience function:

[3] Note that `TransactNamedPipe` is more than a mere convenience compared with `WriteFile` and `ReadFile` and can provide some performance advantages. One experiment shows throughput enhancements ranging from 57% (small messages) to 24% (large messages).

```
BOOL CallNamedPipe (
   LPCTSTR lpPipeName,
   LPVOID lpWriteBuf,
   DWORD cbWriteBuf,
   LPVOID lpReadBuf,
   DWORD cbReadBuf,
   LPDWORD lpcbRead,
   DWORD dwTimeOut)
```

`CallNamedPipe` does not require a permanent connection; instead, it makes a temporary connection by combining the following complete sequence:

```
CreateFile
WriteFile
ReadFile
CloseHandle
```

into a single function. The benefit is that clients do not have long-lived connections, and the server can service more clients at the cost of per-request connection overhead.

The parameter usage is similar to that of `TransactNamedPipe` except that a pipe name, rather than a handle, specifies the pipe. `CallNamedPipe` is synchronous (there is no overlapped structure). It specifies a time-out period, in milliseconds, for the connection but not for the transaction. There are three special values for `dwTimeOut`:

- `NMPWAIT_NOWAIT`
- `NMPWAIT_WAIT_FOREVER`
- `NMPWAIT_USE_DEFAULT_WAIT`, which uses the default time-out period specified by `CreateNamedPipe`

Peeking at Named Pipe Messages

In addition to reading a named pipe using `ReadFile`, you can also determine whether there is actually a message to read using `PeekNamedPipe`. This is useful to poll the named pipe (an inefficient operation), determine the message length so

as to allocate a buffer before reading, or look at the incoming data so as to prioritize its processing.

```
BOOL PeekNamedPipe (
   HANDLE hPipe,
   LPVOID lpBuffer,
   DWORD cbBuffer,
   LPDWORD lpcbRead,
   LPDWORD lpcbAvail,
   LPDWORD lpcbMessage)
```

`PeekNamedPipe` nondestructively reads any bytes or messages in the pipe, but it does not block; it returns immediately.

Test `*lpcbAvail` to determine whether there is data in the pipe; if there is, `*lpcbAvail` will be greater than `0`. `lpBuffer` and `lpcbRead` can be `NULL`, but if you need to look at the data, call `PeekNamedPipe` a second time with a buffer and count large enough to receive the data (based on the `*lpcbAvail` value). If a buffer is specified with `lpBuffer` and `cbBuffer`, then `*lpcbMessage` will tell whether there are leftover message bytes that could not fit into the buffer, allowing you to allocate a large buffer before reading from the named pipe. This value is `0` for a byte mode pipe.

Again, `PeekNamedPipe` reads nondestructively, so a subsequent `ReadFile` is required to remove messages or bytes from the pipe.

The UNIX FIFO is similar to a named pipe, thus allowing communication between unrelated processes. There are limitations compared with Windows named pipes.

- FIFOs are half-duplex.
- FIFOs are limited to a single machine.
- FIFOs are still byte-oriented, so it is easiest to use fixed-size records in client/server applications. Nonetheless, individual read and write operations are atomic.

A server using FIFOs must use a separate FIFO for each client's response, although all clients can send requests to a single, well-known FIFO. A common practice is for the client to include a FIFO name in a connect request.

The UNIX function `mkfifo` is a limited version of `CreateNamedPipe`.

If the clients and server are to be networked, use sockets or a similar transport mechanism. Sockets are full-duplex, but there must still be one separate connection per client.

Example: A Client/Server Command Line Processor

Everything required to build a request/response client/server system is now available. This example is a command line server that executes a command on behalf of the client. Features of the system include:

- Multiple clients can interact with the server.
- The clients can be on different systems on the network, although the clients can also be on the server machine.
- The server is multithreaded, with a thread dedicated to each named pipe instance. That is, there is a *thread pool* of worker threads[4] ready for use by connecting clients. Worker threads are allocated to a client on the basis of the named pipe instance that the system allocates to the client.
- The individual server threads process a single request at a time, simplifying concurrency control. Each thread handles its own requests independently. Nonetheless, exercise the normal precautions if different server threads are accessing the same file or other resource.

Program 11–2 shows the single-threaded client, and its server is Program 11–3. The server corresponds to the model in Figures 7–1 and 11–2. The client request is simply the command line. The server response is the resulting output, which is sent in several messages. The programs also use the include file `ClientServer.h`, which is included in the *Examples* file, and defines the request and response data structures as well as the client and server pipe names.

The client in Program 11–2 also calls a function, `LocateServer`, which finds a server pipe by name. `LocateServer` uses a mailslot, described in a later section and shown in Program 11–5.

The defined records have `DWORD32` length fields; this is done to emphasize the field size.

Program 11–2 `clientNP`: Named Pipe Connection-Oriented Client

```
/* Chapter 11. Client/server system. CLIENT .
   clientNP -- connection-oriented client. */
/* Execute a command line (on the server); display the response. */
/* The client creates a long-lived connection with the server
   (consuming a pipe instance) and prompts user for a command. */

#include "Everything.h"
```

[4] This application-managed thread pool is different from the NT6 thread pool (see Chapter 10).

```
#include "ClientServer.h" /* Defines the request, records. */

int _tmain (int argc, LPTSTR argv[])
{
   HANDLE hNamedPipe = INVALID_HANDLE_VALUE;
   TCHAR quitMsg[] = _T ("$Quit");
   TCHAR serverPipeName[MAX_PATH];
   REQUEST request;/* See ClientServer.h */
   RESPONSE response;/* See ClientServer.h */
   DWORD nRead, nWrite, npMode = PIPE_READMODE_MESSAGE | PIPE_WAIT;

   LocateServer (serverPipeName, MAX_PATH);

   /* Obtain a handle to a NP instance */
   while (INVALID_HANDLE_VALUE == hNamedPipe) {
      WaitNamedPipe (serverPipeName, NMPWAIT_WAIT_FOREVER);
      /* An instance has become available. Attempt to open it
       * before another thread, or the server closes the instance */
      hNamedPipe = CreateFile (serverPipeName,
         GENERIC_READ | GENERIC_WRITE, 0, NULL,
         OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
   }

   /*  Read NP handle in waiting, message mode. Note the 2nd argument
    *  is an address. Client and server may be on the same computer, so
    *  do not set the collection mode and timeout (last 2 args) */
   SetNamedPipeHandleState (hNamedPipe, &npMode, NULL, NULL);
   /* Prompt the user for commands. Terminate on "$Quit". */
   request.command = 0;
   request.rqLen = RQ_SIZE;
   while (ConsolePrompt (promptMsg, request.record,
                MAX_RQRS_LEN, TRUE)
         && (_tcscmp (request.record, quitMsg) != 0)) {
      WriteFile (hNamedPipe, &request, RQ_SIZE, &nWrite, NULL);

      /* Read each response and send it to std out */
      while (ReadFile (hNamedPipe, &response, RS_SIZE, &nRead, NULL))
      {
         if (response.rsLen <= 1) { break; /* Server response end */
         _tprintf (_T ("%s"), response.record);
      }
   }

   _tprintf (_T("Quit command received. Disconnect."));
   CloseHandle (hNamedPipe);
   return 0;
}
```

Program 11–3 is the server program, including the server thread function, that processes the requests from Program 11–2. The server also creates a "server broadcast" thread (see Program 11–4) to broadcast its pipe name on a mailslot to clients that want to connect. Program 11–2 calls the `LocateServer` function, shown in Program 11–5, which reads the information sent by this process. Mailslots are described later in this chapter.

While the code is omitted in Program 11–4, the server (in the *Examples* file) optionally secures its named pipe to prevent access by unauthorized clients. Chapter 15 describes object security and how to use this option. Also, see the example for the server process shutdown logic.

Program 11–3 `serverNP`**:** Multithreaded Named Pipe Server Program

```
/* Chapter 11. ServerNP.
 * Multithreaded command line server. Named pipe version. */

#include "Everything.h"
#include "ClientServer.h" /* Request & response message definitions. */

typedef struct {    /* Argument to a server thread. */
   HANDLE hNamedPipe;  /* Named pipe instance. */
   DWORD threadNumber;
   TCHAR tempFileName[MAX_PATH]; /* Temporary file name. */
} THREAD_ARG;
typedef THREAD_ARG *LPTHREAD_ARG;

volatile static int shutDown = 0;
static DWORD WINAPI Server (LPTHREAD_ARG);
static DWORD WINAPI Connect (LPTHREAD_ARG);
static DWORD WINAPI ServerBroadcast (LPLONG);
static BOOL  WINAPI Handler (DWORD);
static THREAD_ARG threadArgs[MAX_CLIENTS];

_tmain (int argc, LPTSTR argv[])
{
   /* MAX_CLIENTS is defined in ClientServer.h. */
   /* Limited to MAXIMUM_WAIT_OBJECTS WaitForMultipleObjects */
   /* is used by the main thread to wait for the server threads */

   HANDLE hNp, hMonitor, hSrvrThread[MAX_CLIENTS];
   DWORD iNp, monitorId, threadId;
   LPSECURITY_ATTRIBUTES pNPSA = NULL;

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

   /* Create a thread broadcast pipe name periodically. */
```

```
    hMonitor = (HANDLE)_beginthreadex (NULL, 0, ServerBroadcast,
                                       NULL, 0, &monitorId);

    /* Create a pipe instance for every server thread.
     * Create a temp file name for each thread.
     * Create a thread to service that pipe. */

    for (iNp = 0; iNp < MAX_CLIENTS; iNp++) {
        hNp = CreateNamedPipe ( SERVER_PIPE, PIPE_ACCESS_DUPLEX,
              PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE | PIPE_WAIT,
              MAX_CLIENTS, 0, 0, INFINITE, pNPSA);

        threadArgs[iNp].hNamedPipe = hNp;
        threadArgs[iNp].threadNumber = iNp;
        GetTempFileName (_T ("."), _T ("CLP"), 0,
                         threadArgs[iNp].tempFileName);
        hSrvrThread[iNp] = (HANDLE)_beginthreadex (NULL, 0, Server,
                         &threadArgs[iNp], 0, &threadId);
    }

    /* Wait for all the threads to terminate. */
    WaitForMultipleObjects (MAX_CLIENTS, hSrvrThread, TRUE, INFINITE);
    _tprintf (_T ("All Server worker threads have shut down.\n"));

    WaitForSingleObject (hMonitor, INFINITE);
    _tprintf (_T ("Monitor thread has shut down.\n"));

    CloseHandle (hMonitor);
    for (iNp = 0; iNp < MAX_CLIENTS; iNp++) {
        /* Close pipe handles and delete temp files */
        /* Closing temp files is redundant; the worker threads do it */
        CloseHandle (hSrvrThread[iNp]);
        DeleteFile (threadArgs[iNp].tempFileName);
    }

    _tprintf (_T ("Server process will exit.\n"));
    return 0;
}

static DWORD WINAPI Server (LPTHREAD_ARG pThArg)
/* Server thread function. One thread for every potential client. */
{
    /* Each thread keeps its own request, response,
     * and bookkeeping data structures on the stack.
     * Also, each thread creates an additional "connect thread"
     * so that the main worker thread can test the shutdown flag
     * periodically while waiting for a client connection. */

    HANDLE hNamedPipe, hTmpFile = INVALID_HANDLE_VALUE,
           hConTh = NULL, hClient;
```