

```

DWORD nXfer, conThStatus, clientProcessId;
STARTUPINFO startInfoCh;
SECURITY_ATTRIBUTES tempSA =
    {sizeof (SECURITY_ATTRIBUTES), NULL, TRUE};
PROCESS_INFORMATION procInfo;
FILE *fp;
REQUEST request;
RESPONSE response;

GetStartupInfo (&startInfoCh);
hNamedPipe = pThArg->hNamedPipe;

/* Open temporary results file for connections to this instance. */
hTmpFile = CreateFile (pThArg->tempFileName,
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, &tempSA,
    CREATE_ALWAYS, FILE_ATTRIBUTE_TEMPORARY, NULL);

while (!shutDownFlag) { /* Connection loop */
    /* Create a connection thread, and wait for it to terminate */
    /* Use timeout on wait so that shutdown flag can be tested */
    hConTh = (HANDLE)_beginthreadex (NULL, 0, Connect,
        pThArg, 0, NULL);

    /* Wait for a client connection. */
    while (!shutDownFlag &&
        WaitForSingleObject (hConTh, CS_TIMEOUT) == WAIT_TIMEOUT)
        { /* Empty loop body */};
    if (shutDownFlag) _tprintf(_T("Thread %d received shutdown\n"),
        pThArg->threadNumber);
    if (shutDownFlag) continue; /* Could be set by other threads */

    CloseHandle (hConTh); hConTh = NULL;
    /* A connection now exists */
    GetNamedPipeClientProcessId(pThArg->hNamedPipe,
        &clientProcessId);
    _tprintf(_T("Connect to client process id: %d\n"),
        clientProcessId);
    while (!shutDownFlag && ReadFile (hNamedPipe, &request,
        RQ_SIZE, &nXfer, NULL)) {
        _tprintf(_T("Command from client thread: %d. %s\n"),
            clientProcessId, request.record);
        /* Receive new commands until the client disconnects */
        shutDownFlag = shutDownFlag ||
            (_tcscmp (request.record, shutRequest) == 0);
        if (shutDownFlag) continue;

        /* Main command loop */
        /* Create a process to carry out the command. */
        startInfoCh.hStdOutput = hTmpFile;

```

```

startInfoCh.hStdError = hTmpFile;
startInfoCh.hStdInput = GetStdHandle (STD_INPUT_HANDLE);
startInfoCh.dwFlags = STARTF_USESTDHANDLES;

CreateProcess (NULL, request.record, NULL,
               NULL, TRUE, /* Inherit handles. */
               0, NULL, NULL, &startInfoCh, &procInfo);

CloseHandle (procInfo.hThread);
WaitForSingleObject (procInfo.hProcess, INFINITE);
CloseHandle (procInfo.hProcess);

/* Respond a line at a time. It is convenient to use
   C library line-oriented routines at this point. */
fp = _tfopen (pThArg->tempFileName, _T ("r"));
while (_fgetts(response.record, MAX_RQRS_LEN, fp) != NULL) {
    response.rsLen = strlen(response.record) + 1;
    WriteFile (hNamedPipe, &response,
               response.rsLen+sizeof(response.rsLen), &nXfer, NULL);
}
/* Write a terminating record. */
_tcscpy (response.record, _T(""));
response.rsLen = 0;
WriteFile (hNamedPipe, &response, sizeof(response.rsLen),
           &nXfer, NULL);
FlushFileBuffers (hNamedPipe);
fclose (fp);

/* Erase temp file contents */
SetFilePointer (hTmpFile, 0, NULL, FILE_BEGIN);
SetEndOfFile (hTmpFile);
} /* End of main command loop. Get next command */

/* Client has disconnected or there was a shutdown request */
/* Terminate this client connection and then wait for another */
FlushFileBuffers (hNamedPipe);
DisconnectNamedPipe (hNamedPipe);
}

/* Force connection thread to shut down if it is still active */
if (hConTh != NULL) {
    GetExitCodeThread (hConTh, &conThStatus);
    if (conThStatus == STILL_ACTIVE) {
        hClient = CreateFile (SERVER_PIPE,
                              GENERIC_READ | GENERIC_WRITE, 0, NULL,
                              OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hClient != INVALID_HANDLE_VALUE) CloseHandle (hClient);
        WaitForSingleObject (hConTh, INFINITE);
    }
}
}

```

```

    _tprintf (_T("Thread %d shutting down.\n"), pThArg->threadNumber);
    /* End of command processing loop. Free thread resources; exit. */
    CloseHandle (hTmpFile); hTmpFile = INVALID_HANDLE_VALUE;
    DeleteFile (pThArg->tempFileName);
    _tprintf (_T ("Exiting server thread number %d\n"),
        pThArg->threadNumber);
    return 0;
}

static DWORD WINAPI Connect (LPTHREAD_ARG pThArg)
{
    /* Connection thread allowing server to poll ShutDown flag. */
    ConnectNamedPipe (pThArg->hNamedPipe, NULL);
    return 0;
}

BOOL WINAPI Handler (DWORD CtrlEvent)
{
    /* Shut down the system. */
    shutDownFlag = TRUE;
    return TRUE;
}

```

Comments on the Client/Server Command Line Processor

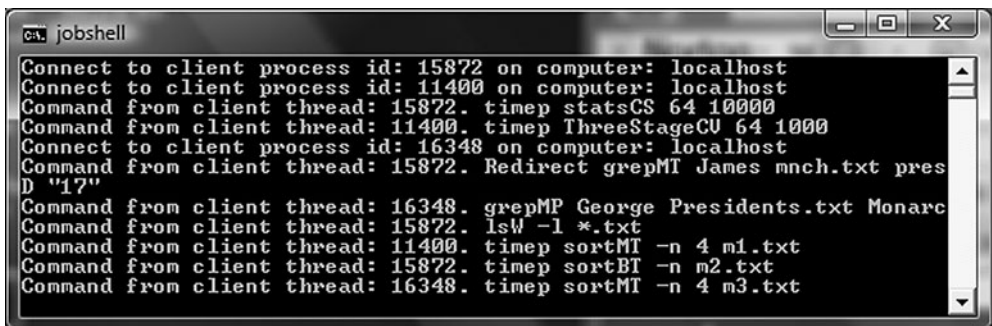
This solution includes a number of features as well as limitations that will be addressed in later chapters.

- Multiple client processes can connect with the server and perform concurrent requests; each client has a dedicated server (or worker) thread allocated from the thread pool.
- The server and clients can run from separate command prompts or can run under control of JobShell (Program 6–3).
- If all the named pipe instances are in use when a client attempts to connect, the new client will wait until a different client disconnects on receiving a \$Quit command, making a pipe instance available for another client. Several new clients may be attempting to connect concurrently and will race to open the available instance; threads that lose this race will need to wait again.
- Each server thread performs synchronous I/O, but some server threads can be processing requests while others are waiting for connections or client requests.

- Extension to networked clients is straightforward, subject to the limitations of named pipes discussed earlier in this chapter. Simply change the pipe names in the header file or add a client command line parameter for the server name.
- Each server worker thread creates a simple connection thread, which calls `ConnectNamedPipe` and terminates as soon as a client connects. This allows a worker thread to wait, with a time-out, on the connection thread handle and test the global shutdown flag periodically. If the worker threads blocked on `ConnectNamedPipe`, they could not test the flag and the server could not shut down. For this reason, the server thread performs a `CreateFile` on the named pipe in order to force the connection thread to resume and shut down. Asynchronous I/O (Chapter 14) is an alternative, so that an event could be associated with the `ConnectNamedPipe` call. The comments in the *Examples* file source provide additional alternatives and information. Without this solution, connection threads might never terminate by themselves, resulting in resource leaks. Chapter 12 discusses this subject.
- There are a number of opportunities to enhance the system. For example, there could be an option to execute an in-process server by using a DLL that implements some of the commands. This enhancement is added in Chapter 12.
- The number of server threads is limited by the `WaitForMultipleObjects` call in the main thread. While this limitation is easily overcome, the system here is not truly scalable; too many threads will impair performance, as we saw in Chapter 10. Chapter 14 uses asynchronous I/O ports to address this issue.

Running the Client and Server

The details of how clients locate servers are explained in the next section (“Mailslots”). However, we can now show the programs in operation. Run 11–3 shows the server, Program 11–3, which was started using `JobShell` from



```

C:\> jobshell
Connect to client process id: 15872 on computer: localhost
Connect to client process id: 11400 on computer: localhost
Command from client thread: 15872. timep statsCS 64 10000
Command from client thread: 11400. timep ThreeStageCV 64 1000
Connect to client process id: 16348 on computer: localhost
Command from client thread: 15872. Redirect grepMT James mnch.txt pres
D "17"
Command from client thread: 16348. grepMP George Presidents.txt Monarc
Command from client thread: 15872. lsW -l *.txt
Command from client thread: 11400. timep sortMT -n 4 m1.txt
Command from client thread: 15872. timep sortBT -n m2.txt
Command from client thread: 16348. timep sortMT -n 4 m3.txt
  
```

Run 11–3 serverNP: Servicing Several Clients

Chapter 6. The server accepts connections from three client processes, reporting the connections and the commands.

Run 11-4 shows one of the clients in operation; this is the client represented by process ID 15872 in Run 11-3. The commands are familiar from previous chapters.

```

C:\> Command Prompt - clientNP
c:\WSP4_Examples\run8>clientNP
Looking for a server.
Server has been located. Pipe Name: \\.\PIPE\SERVERER.

Enter Command: timep statsCS 64 10000
statsCS 64 10000
Worker threads have terminated
Real Time: 00:00:00:231
User Time: 00:00:00:624
Sys Time: 00:00:00:093

Enter Command: Redirect grepMT James mnch.txt pres.txt = FIND "17"
16331014 16850423 16890906 17010906 JamesII i
17510316 18090300 18170300 18390628 Madison,James i
17580428 18170300 18250209 18310704 Monroe,James i
17951102 18450304 18490300 18490300 Polk,JamesK i
17910423 18570304 18610304 18680601 Buchanan,James i

Enter Command: lsW -l *.txt
64000000 10/12/2009 00:52:26 m1.txt
64000000 10/12/2009 00:52:43 m2.txt
64000000 10/12/2009 00:52:58 m3.txt
64000000 10/12/2009 00:53:13 m4.txt
64000000 10/12/2009 00:53:30 m5.txt
64000000 10/12/2009 00:53:45 m6.txt
64000000 10/12/2009 00:54:01 m7.txt
64000000 10/12/2009 00:54:16 m8.txt
3968 09/20/2009 14:49:29 mnch.txt
3968 09/20/2009 14:49:29 Monarchs.TXT
2816 09/20/2009 14:56:27 pres.txt
2816 09/20/2009 14:56:27 Presidents.TXT
2048 09/29/2009 21:35:16 small.txt
384 09/27/2009 19:11:05 small2.txt

Enter Command: timep sortBT -n m2.txt
Real Time: 00:02:36:529
User Time: 00:00:13:962
Sys Time: 00:02:19:262

Enter Command:

```

Run 11-4 clientNP: Client Commands and Results

Mailslots

A Windows mailslot, like a named pipe, has a name that unrelated processes can use for communication. Mailslots are a broadcast mechanism, similar to datagrams (see Chapter 12), and behave differently from named pipes, making them useful in some important but limited situations. Here are the significant mailslot characteristics:

- A mailslot is one-directional.
- A mailslot can have multiple writers and multiple readers, but frequently it will be one-to-many of one form or the other.
- A writer (client) does not know for certain that all, some, or any readers (servers) actually received the message.
- Mailslots can be located over a network domain.
- Message lengths are limited.

Using a mailslot requires the following operations.

- Each server creates a mailslot handle with `CreateMailslot`.
- The server then waits to receive a mailslot message with a `ReadFile` call.
- A write-only client should open the mailslot with `CreateFile` and write messages with `WriteFile`. The open will fail (name not found) if there are no waiting readers.

A client's message can be read by *all* servers; all of them receive the same message.

There is one further possibility. The client, in performing the `CreateFile`, can specify a name of this form:

```
\\*\mailslot\mailslotname
```

In this way, the `*` acts as a wildcard, and the client can locate every server in the *domain*, a networked group of systems assigned a common name by the network administrator. The client can then connect to one of the servers, assuming that they all provide the same basic functionality, although the server responses could contain information (current load, performance, etc.) that would influence the client's choice.

Using Mailslots

The preceding client/server command processor suggests several ways that mailslots might be useful. Here is one scenario that will solve the server location problem in the preceding client/server system (Programs 11–2 and 11–3).

The *application server*, acting as a *mailslot client*, periodically broadcasts its name and a named pipe name. Any *application client* that wants to find a server can receive this name by being a *mailslot server*. In a similar manner, the command line server can periodically broadcast its status, including information such as utilization, to the clients. This situation could be described as a single

writer (the mailslot client) and multiple readers (the mailslot servers). If there were multiple mailslot clients (that is, multiple application servers), there would be a many-to-many situation.

Alternatively, a single reader could receive messages from numerous writers, perhaps giving their status—that is, there would be multiple writers and a single reader. This usage, for example, in a bulletin board application, justifies the term *mailslot*. These first two uses—name and status broadcast—can be combined so that a client can select the most appropriate server.

The inversion of the terms *client* and *server* is confusing in this context, but notice that both named pipe and mailslot servers perform the `CreateNamedPipe` (or `CreateMailSlot`) calls, while the client (named pipe or mailslot) connects using `CreateFile`. Also, in both cases, the client performs the first `WriteFile` and the server performs the first `ReadFile`.

```
HANDLE CreateMailslot (LPCTSTR lpName,
    DWORD cbMaxMsg,
    DWORD dwReadTimeout,
    LPSECURITY_ATTRIBUTES lpsa)
```

Figure 11–3 shows the use of mailslots for the first approach.

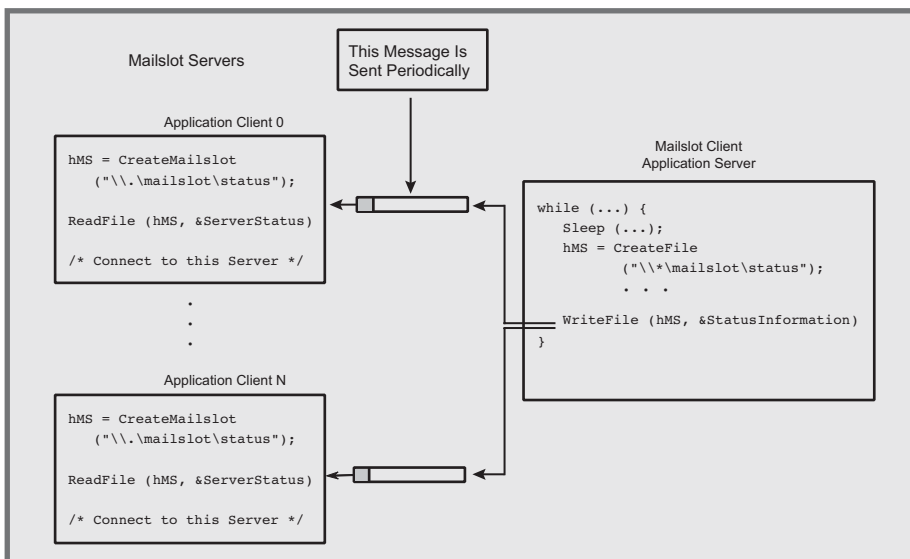


Figure 11–3 Clients Using a Mailslot to Locate a Server

Creating and Opening a Mailslot

The mailslot servers (readers) use `CreateMailslot` to create a mailslot and to get a handle for use with `ReadFile`. There can be only one mailslot of a given name on a specific machine, but several systems in a network can use the same name to take advantage of mailslots in a multireader situation.

Parameters

`lpName` points to a mailslot name of this form:

```
\\.\mailslot\[path]name
```

The name must be unique. The period (.) indicates that the mailslot is created on the current machine. The path, if any, represents a pseudo directory, and path components are separated by backslash characters.

`cbMaxMsg` is the maximum size (in bytes) for messages that a client can write. A value of 0 means no limit.

`dwReadTimeout` is the number of milliseconds that a read operation will wait. A value of 0 causes an immediate return, and `MAILSLOT_WAIT_FOREVER` is an infinite wait (no time-out).

The client (writer), when opening a mailslot with `CreateFile`, can use the following name forms.

`\\.\mailslot\[path]name` specifies a local mailslot.

`\\computername\mailslot\[path]name` specifies a mailslot on a specified machine.

`\\domainname\mailslot\[path]name` specifies all mailslots on machines in the domain. In this case, the maximum message length is 424 bytes.

`*\mailslot\[path]name` specifies all mailslots on machines in the domain. In this case, the maximum message length is also 424 bytes.

Finally, the client must specify the `FILE_SHARE_READ` flag.

The functions `GetMailslotInfo` and `SetMailslotInfo` are similar to their named pipe counterparts.

UNIX does not have a facility comparable to mailslots. A broadcast or multicast TCP/IP datagram, however, could be used for this purpose.

Pipe and Mailslot Creation, Connection, and Naming

Table 11–1 summarizes the valid pipe names that can be used by application clients and servers. It also summarizes the functions to create and connect with named pipes.

Table 11–2 gives similar information for mailslots. Recall that the mailslot client (or server) may not be the same process or even on the same computer as the application client (or server).

Table 11–1 Named Pipes: Creating, Connecting, and Naming

	Application Client	Application Server
Named Pipe Handle or Connection	CreateFile CallNamedPipe, TransactNamedPipe	CreateNamedPipe
Pipe Name	\\.\pipe <i>pipename</i> (pipe is local) \\sys_name\pipe <i>pipename</i> (pipe is local or remote)	\\.\pipe <i>pipename</i> (pipe is created locally)

Table 11–2 Mailslots: Creating, Connecting, and Naming

	Mailslot Client	Mailslot Server
Mailslot Handle	CreateFile	CreateMailslot
Mailslot Name	\\.\msname (mailslot is local) \\sys_name\mailslot\msname (mailslot is on a specific remote system) \\domain_name\mailslot\msname (all domain mailslots with this name) *\mailslot\msname (all mailslots with this name)	\\.\msname (mailslot is created locally)

Example: A Server That Clients Can Locate

Program 11–4 shows the thread function that the command line server (Program 11–3), *acting as a mailslot client*, uses to broadcast its pipe name to waiting clients. There can be multiple servers with different characteristics and pipe names, and the clients obtain their names from the well-known mailslot name. Program 11–3 starts this function as a thread.

Note: In practice, many client/server systems invert the location logic used here. The alternative is to have the application client also act as the mailslot client and broadcast a message requesting a server to respond on a specified named pipe; the client determines the pipe name and includes that name in the message. The application server, acting as a mailslot server, then reads the request and creates a connection on the specified named pipe.

Program 11–4’s inverted logic solution has advantages, although it consumes a mailslot name:

- The latency time to discover a server decreases because there is no need to wait for a server to broadcast its name.
- Network bandwidth and CPU cycles are used only as required when a client needs to discover a server.

Program 11–4 SrvrBcst: Mailslot Client Thread Function

```
int _tmain (int argc, LPTSTR argv[])
{
    BOOL exit;
    MS_MESSAGE mailslotNotify;
    DWORD nXfer;
    HANDLE hMailslot;

    /* Open the mailslot for the MS "client" writer. */
    while (TRUE) {
        exit = FALSE;
        while (!exit) { /* Wait for a client to create a MS. */
            hMailslot = CreateFile (MS_CLTNAME,
                GENERIC_WRITE | GENERIC_READ,
                FILE_SHARE_READ | FILE_SHARE_WRITE,
                NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
            if (hMailslot == INVALID_HANDLE_VALUE) {
                Sleep (5000);
            }
            else exit = TRUE;
        }
    }
}
```

```

/* Send out the message to the mailslot. */
/* Avoid an "information discovery" security exposure. */
ZeroMemory(&mailSlotNotify, sizeof(mailSlotNotify));
mailSlotNotify.msStatus = 0;
mailSlotNotify.msUtilization = 0;

_tcscpy (mailSlotNotify.msName, SERVER_PIPE);
WriteFile (hMailSlot, &mailSlotNotify, MSM_SIZE, &nXfer, NULL);
CloseHandle (hMailSlot);

/* Wait for another client to open a mailslot. */
}

/* Not reachable. */
return(0);
}

```

Program 11-5 shows the `LocSrver` function called by the client (see Program 11-2) so that it can locate the server.

Program 11-5 LocSrver: Mailslot Server

```

/* Chapter 11. LocSrver.c */
/* Find a server by reading the mailslot
   used to broadcast server names. */

#include "Everything.h"
#include "ClientServer.h" /* Defines mailslot name. */

BOOL LocateServer (LPTSTR pPipeName, DWORD size)
{
    HANDLE hMailSlot;
    MS_MESSAGE serverMsg;
    BOOL found = FALSE;
    DWORD cbRead;

    hMailSlot = CreateMailslot (MS_SRVNAME, 0, CS_TIMEOUT, NULL);
    if (hMailSlot == INVALID_HANDLE_VALUE)
        ReportError (_T ("MS create error."), 11, TRUE);

    /* Communicate with the server to be certain that it is running.
       * Server must have time to find the mailslot & send pipe name. */

    while (!found) {
        _tprintf (_T ("Looking for a server.\n"));
        found = ReadFile (hMailSlot, &serverMsg,
                        sizeof(serverMsg), &cbRead, NULL);
    }
}

```

```

    _tprintf (_T ("Server has been located.\n"));

    /* Close the mailslot. */
    CloseHandle (hMailSlot);
    if (found) _tcsncpy (pPipeName, serverMsg.msName, size-1);
    return found;
}

```

Summary

Windows pipes and mailslots, which are accessed with file I/O operations, provide stream-oriented interprocess and networked communication. The examples show how to pipe data from one process to another and a simple, multithreaded client/server system. Pipes also provide another thread synchronization method because a reading thread blocks until another thread writes to the pipe.

Looking Ahead

Chapter 12 shows how to use industry-standard, rather than Windows proprietary, interprocess and networking communication. The same client/server system, with some server enhancements, will be rewritten to use the standard methods.

Exercises

- 11-1. Carry out experiments to determine the accuracy of the performance advantages cited for `TransactNamedPipe`. You will need to make some changes to the server code as given. Also compare the results with the current implementation.
- 11-2. Use the `JobShell` program from Chapter 6 to start the server and several clients, where each client is created using the “detached” option. Eventually, shut down the server by sending a console control event through the `kill` command. Can you suggest any improvements to the `serverNP` shutdown logic so that a connected server thread can test the shutdown flag while blocked waiting for a client request? *Hint:* Create a read thread similar to the connection thread.
- 11-3. Enhance the server so that the name of its named pipe is an argument on the command line. Bring up multiple server processes with different pipe