Garden Hoses at Work

Ruediger R. Asche
Microsoft Developer Network Technology Group

Created: July 29, 1994
Revised: June 1, 1995 (redesigned class definitions; incorporated information on MFC sockets)

Click to open or copy files in the CommChat sample application for this technical article.

Abstract

This article shows how named pipes can be embedded into a C++ class and discusses named pipes as a network communication mechanism. Use the accompanying sample application, CommChat, and the related article, "Communication with Class" to study possible uses of this C++ class. The **CNamedPipe** class hierarchy is also used to demonstrate security in the article series that begins with "Windows NT Security in Theory and Practice" and its accompanying sample suite.

Road Map

This article is second in a series of technical articles that explore network programming with Visual C++™ and the Microsoft® Foundation Class Library (MFC). The series consists of the following articles:

"Communication with Class" (introduction and description of the CommChat sample)

"Garden Hoses at Work" (named pipes)

"Power Outlets in Action: Windows Sockets" (Windows® Sockets)

"Aristocratic Communication: NetBIOS" (NetBIOS)

"Plugs and Jacks: Network Interfaces Compared" (summary)

The CommChat sample application illustrates the concepts and implements the C++ objects discussed in these articles.

Introduction

Named pipes are a very convenient and powerful mechanism for transferring data over a network. They are part of the Microsoft Win32® application programming interface (API) definition and can, therefore, be employed by any Win32 application without additional dynamic-link libraries (DLLs), import libraries, or header files to link with. The only restriction is that a machine must run as a network server to create server ends of named pipes. (Note that Microsoft Windows for Workgroups is not sufficient.)

In this article, we encapsulate named pipes into a C++ class that you can easily incorporate into your own C++ Microsoft Foundation Class Library (MFC) application. As explained in the article "Communication with Class," the base class for the communication objects we will define, **CCommunication**, is derived from the MFC class **CFile**, thereby treating network communications like files. As we will see, this approach fits very neatly into the design philosophy of named pipes.

What Are Pipes?

A pipe is a simple data transfer mechanism that works, well, pretty much like a pipe: You stuff things in one end and the things come out the other end. You can establish a pipe both locally (using either so-called anonymous pipes or named pipes) and to a remote machine (a named pipe). In order to be able to utilize a named pipe over a network—the task we will focus on in this article—one machine must have created the pipe (the server), and another machine that wants to use the pipe (the client) must know both the name of the server machine and the name of the pipe on the server machine.

Once the client and the server have agreed to communicate using a named pipe, they can stuff data into the pipe and read the data from the other side of the pipe. There are several rules that apply to pipes. Note that these rules talk about using named pipes across different machines, but named pipes may also be used to communicate between processes on the same machine; thus, the same rules hold true when we substitute *tasks* for *machines*.

- A pipe can be created such that communication can take place in only one direction (either inbound from the client to the server or outbound from the server to the client) or in two (bidirectional).
- There are only two ends to a pipe. A pipe can be created to support several instances; that is, several clients can hook up to the same server, but every single instance of a named pipe can only service one communication. That means that it is not possible for tasks on three or more machines to all use the same pipe and cross-communicate. When two machines hook up to the server pipe of a third machine, then that third machine will have two separate conversations going.
- Any machine that can create named pipes (a named pipe server) can create an arbitrary number of pipes, as long as each pipe is uniquely identified and as long as there are system resources available. As we will see later on, this is different from other communication strategies that view a machine address as the "finest granularity" of a communication.

Implementing CNamedPipe

In accordance with our discussion in the article "Communication with Class" in the MSDN Library, there are two classes defined to encapsulate named pipes: **CClientNamedPipe** and **CServerNamedPipe**. Because named pipes are securable objects under Windows NT™, there is also a class derived from **CServerNamedPipe** that supports security functionality: **CSecuredNamedPipe**. You will find the implementation of the named pipe classes in the file NPIPE.CPP in the CommChat sample application. In this section, we will look at the **Open**, **Close**, **Read**, **Write**, **Duplicate**, **AwaitCommunicationAttempt**, and **CancelCommunicationAttempt** member functions. To complicate matters a little bit, the **CNamedPipe** class hierarchy has not one, but three implementations. The Win32® application programming interface (API) defines three variations of named pipes: *synchronous blocking pipes*, *asynchronous non-blocking pipes* (I like to call these "polling" pipes for reasons that I will explain later), and *asynchronous overlapped pipes*. Let us first look at these variations before we discuss their implementations. The three variations are implemented in the PIPEVARS subdirectory in the CommChat sample application.

Synchronous Blocking Pipes

This is the simplest form of a named pipe. All operations on the pipe block the calling thread until the operations are completed. In the synchronous blocking operation mode, named pipes behave a lot like standard files, where I/O operations return only when completed. Synchronous blocking pipes are implemented in SIMPLE.CPP and defined in SIMPLE.H in the PIPEVARS subdirectory.

Asynchronous Non-Blocking Pipes (Polling Pipes)

This variation is usually swept swiftly under the carpet because it exists only for compatibility reasons with older versions of Microsoft LAN Manager. An I/O operation on an asynchronous non-blocking pipe returns immediately and must be resubmitted until the operation is completed. In other words, the thread that submits the I/O request must poll until it finds the operation to be finished (hence the nickname "polling pipe"). Unlike asynchronous overlapped pipes, there is no automatic notification system that can be used to determine when an I/O operation has been completed.

The interesting thing about polling pipes is that the state of a pipe can be dynamically changed from synchronous blocking to polling and vice versa. We will use this feature to implement the **AwaitCommunicationAttempt** member in the polling pipe variation. Polling pipes are implemented in NONBLOCK.CPP and defined in NONBLOCK.H in the PIPEVARS subdirectory.

Asynchronous Overlapped Pipes

This is both the most powerful and the least universal variation of named pipes—most powerful because asynchronous overlapped pipes can be used in a multitude of ways, and least universal because they rely on overlapped I/O, which is available only under Windows NT.

I/O operations on asynchronous overlapped pipes return immediately. When the I/O operation (which can take up a significant amount of time on a network) is complete, the calling thread can be notified in one of two ways: by signalling an event, or by calling a callback routine that was specified when the I/O operation was submitted. You can employ asynchronous I/O to ensure throughput and responsiveness in a non-multithreaded environment. You can also use asynchronous I/O to exploit Windows NT's resources more efficiently, because you can use the same pipe to service different communications on the same thread.

For the implementation and definition of asynchronous overlapped pipes, see the files OVERLAP.CPP and OVERLAP.H in the PIPEVARS subdirectory. Note that in the CommChat application, I do not exploit asynchronous I/O. Instead, I delegate I/O into background threads that can process the I/O synchronously. Thus, the implementation I provide for asynchronous overlapped pipes resynchronizes the overlapped I/O and uses overlapping features only in the **AwaitCommunicationAttempt/CancelCommunicationAttempt** member function pair.

Opening a Pipe

When CommChat starts up, it first attempts to create two server-named pipe objects: one for reading and one for writing. Likewise, any client that tries to connect to a server also tries to create two communication objects. Because the server and client communication objects are instances of different C++ classes, the **Open** member functions of the **CClientNamedPipe** and **CServerNamedPipe** class contain the respective client and server code to establish the communication.

This distinction is made by the first parameter that is passed to the **Open** member function: If the name is NULL, a server object is to be created; otherwise, the **Open** function tries to establish a client connection to the machine specified by that name.

Any named pipe is uniquely identified by a pipe name that must be known to both sides. A pipe name must follow the syntax

```
\\.\pipe\<unique name>
```

on the server side. The client opens the pipe using the path

```
\\<servermachinename>\pipe\<unique name>
```

If the server and the client process happen to reside on the same machine, the client process can use the same name as the server.

Thus, if there is a server named GIMMICK and a client GNORPS, GIMMICK would create pipes with these names in CommChat

```
\\.\pipe\chatread
```

and

```
\\.\pipe\chatwrit
```

for the inbound and outbound communications, respectively. When GNORPS tries to establish a communication with GIMMICK, it looks for the pipes named

```
\\gimmick\pipe\chatwrit
```

and

```
\\gimmick\pipe\chatread
```

for inbound and outbound connections, respectively. Note that the client calls **CreateFile** to open the pipe as it would open a file on the network using the Universal Naming Conventions (UNC) file name convention, whereas the server creates the pipe using **CreateNamedPipe**. After a communication has been established, the **ReadFile** and **WriteFile** calls can be used to read from or write to a pipe, respectively, as if the pipe were a file.

This really looks as if the machine GIMMICK had shared a directory under the logical name PIPE that points to some kind of network-aware directory, but this is not what happens here. The special name PIPE is parsed by the operating system when the **CreateFile** call is processed and rerouted to the network. Under Windows NT, you can run the WINOBJ utility that comes with the resource kit and see that there is a symbolic link object, \DosDevices\PIPE, that points to a special object type, \Device\NamedPipe. This logical device is implemented by the network software and redirects the input/output (I/O) requests to the network.

It is obvious that a client can connect to a named pipe on the server only after the server has created the pipe. For CommChat, that means that you can establish a connection with a remote machine only if an instance of CommChat is already running on that machine and the user on that machine has selected named pipes as the communication type.

If you wish to make the server end of a pipe available at any time after a machine has been booted, you need to relocate the code that creates the pipe into a service on Windows NT and have that service launch an instance of your application as soon as a client connects. (A service under Windows NT is roughly a background process that can be made to start automatically once the machine that the server runs on has started.) The SIMPLE server sample in the MSDN Library (Sample Code; Product Samples; System Toolkits, DDKs, and SDKs; Win32 SDK Samples) demonstrates how to accomplish this; we will look at a few network communication strategies later that rely on servers. (In fact, without the network server service, named pipes could not be created under Windows NT.)

Another thing worth mentioning is that there is no way for a client to query whether a certain server pipe exists. A client must establish a connection in order to determine whether another machine has created a specific pipe. We will look at the pitfalls and details of this issue when we implement a network browsing dialog box in a later article.

Note that you can create another instance of a named pipe on a server machine, but if you make a **CreateNamedPipe** using a name that has already been created before, you will obtain a secondary instance as a pipe that has the same properties as the pipe that was created in the first place. You cannot create more instances of a named pipe than specified in the first call to **CreateNamedPipe** when you use the same name.

Thus, if you want to reuse the C++ classes I provide for several processes that run concurrently, you should devise a way to create a unique name for each fresh pipe.

In CommChat, I restrict the server pipe objects to single instances. I suspect that it would be easy to allow several communications at the same time. (Wouldn't it be fun to have two documents open, communicating with your boss in one multiple-document interface (MDI) child window and at the same time bitching about him with your colleague in another window?) But guess what, I had a deadline to keep, so I have postponed this feature.

The **CServerNamedPipe** and **CClientNamedPipe** classes have overloaded constructors to make the pipe names application-configurable: To create an object of the **CServerNamedPipe** or **CClientNamedPipe** class, you can either use no parameters (in which case the pipe names will be assigned hardcoded defaults), or use three strings (which determine the names that will be used to create inbound, outbound, and bidirectional pipes) as parameters. If you decide to reuse the classes for your own communications, you should probably use the parameterized constructor to avoid name conflicts.

Security

Under Windows NT (currently the only platform that can create server ends of named pipes), named pipes are securable objects; that is, by associating the named pipe server end with an access control list (ACL), the server service of a Windows NT machine can control the users allowed to connect to the pipe.

I figured out what this meant when CommChat communicated between my two Windows NT machines like a charm *as long as I had logged onto both with the same password and user name*, but when I logged onto one of the machines as a different user with another password, the client could not open its end of the pipe.

In the articles "Windows NT Security in Theory and Practice" and "The Guts of Security," we will see how the security on named pipes can be controlled. If you do not want to go through the hassle, and you are satisfied with a named pipe that can be opened by everybody, you have two options:

1. Follow the procedure described in the Knowledge Base article Q102798, "Security Attributes on Named Pipes," which provides the complete code necessary to grant users access to a named pipe.

2. In CommChat, replace the references to **CServerNamedPipe** with **CSecuredNamedPipe**, and change the implementation of **CSecuredNamedPipe** to call the constructor of **CSecureableObject** with the parameter set to FALSE instead of TRUE. This will associate the named pipe with a security descriptor that allows everybody to connect to the pipe. Once more, this procedure is described in detail in my article series that begins with "Windows NT Security in Theory and Practice."

Creation flags

Pipes come in a number of varieties and flavors. There are message-sized versus byte-sized pipes, blocking versus nonblocking pipes, write-through versus buffering pipes, overlapped versus nonoverlapped operations, and you can also specify a buffer size or a time-out when creating the pipe.

We defined the three basic versions of named pipes (synchronous non-blocking pipes, polling pipes, and asynchronous overlapped pipes) in the section "Implementing CNamedPipe" earlier in this article.

Using the FILE_FLAG_WRITE_THROUGH operation, an application can bypass the buffering of written data over the network; FILE_FLAG_NO_BUFFERING will go even further in writing the data directly from the sender to the receiver's memory without intervention by a cache. Please refer to Knowledge Base article Q99794 for details on this process.

Finally, there are message-sized versus byte-sized pipes. The difference here is that message-sized pipes transfer data in chunks, whereas byte-sized pipes transfer data byte by byte. What does that mean? Well, imagine you submit a **ReadFile** call to a named pipe that was created in byte-sized transfer mode. When you specify, say, 10 as the number of bytes you want to read, the read operation will not complete until 10 bytes are transferred. This scheme is fairly painful because it implies that the receiver must always know how many bytes the sender sent so that no bytes get stuck in the pipe, or conversely, the read operation waits for bytes that never make it into the pipe.

Message-sized pipes, however, employ an implied protocol. If you submit a **ReadFile** call to a named pipe that was opened in message-size mode, the **ReadFile** call returns as soon as the number of bytes that have been received corresponds to the number that has been written. Thus, if the **ReadFile** call asks for 10 bytes, and the corresponding **WriteFile** call on the sender's side has written 8 bytes, the **ReadFile** call will return after the 8th byte has been received. However, if **ReadFile** only asks for 5 bytes, the call will fail, and **GetLastError** will return ERROR_MORE_DATA, indicating that the buffer passed to **ReadFile** was too small.

Message-sized pipes are very convenient because they take some work away from your protocols. If you are dealing with byte-sized pipes only, the sender needs to explicitly inform the receiver of the size of any transmission before doing the transmission itself, so that the receiver knows exactly after how many bytes to stop. Message-sized pipes do that for you: As long as the buffer passed to **ReadFile** is larger than the size of the message, **ReadFile** will return after the message is read entirely. If the buffer is smaller than the message, well, we discussed this in the preceding paragraph.

To close the pipe, you can destroy the pipe object using **CloseHandle**. However, a server that wishes to terminate a connection can also call **DisconnectNamedPipe**, which will not destroy the pipe object, but will only shut off the current connection. This is useful because it means that another client can use the pipe after the current communication is finished.

AwaitCommunicationAttempt

To establish a communication, the server submits the **ConnectNamedPipe** call. The behavior of **ConnectNamedPipe** will differ, depending on the version of the named pipe you are using. You might want to check the three versions of the named pipe classes in the PIPEVARS subdirectory as we go along.

First for synchronous blocking pipes, as implemented in SIMPLE.CPP: In this pipe implementation, **ConnectNamedPipe** will not return until a client has successfully connected to the server, using its **CreateFile** call. On both sides, the named pipe handle will now behave exactly like a file handle. That means that the **DuplicateHandle**, **CloseHandle**, and **WaitForSingleObject** system functions on those handles work exactly as they do on other system objects, and **ReadFile** and **WriteFile** are used to communicate via the pipe.

So far, so good. But what about **CancelCommunicationAttempt**, that is, what if the server application decides to abandon a pending connection? (Although we do not use this feature in CommChat, there are good uses for it; for example, see the CLIAPP/SRVAPP application suite that accompanies my security article series.) If one thread has a pending

ConnectNamedPipe call, the only way to recover from the pipe's listening state is to have a client connect to the pipe. Even a **CloseHandle** call on the pipe handle from another thread does not help us here. (The **CloseHandle** call will be blocked until **ConnectNamedPipe** returns—classical deadlock!) I found one trick for implementing **CancelCommunicationAttempt** on synchronous blocking named pipes: Connect to the pipe from the server application itself, then clean up the communication and let the **AwaitCommunicationAttempt** member (that has now returned from its **ConnectNamedPipe** call) do the rest. The following code is from the SIMPLE.CPP file:

```
BOOL CServerNamedPipe::CancelCommunicationAttempt()
{
m_bNotCancelled = FALSE;
BOOL bReturn = FALSE;
CClientCommunication *csTerminator = new CClientNamedPipe(m_szPipeEndNameIn,m_szPipeEndName
UINT iFlags = (m_iFlagsOpened == modeRead)? modeWrite :
             ((m_iFlagsOpened == modeWrite) ? modeRead : modeReadWrite);
bReturn = csTerminator->Open(".",iFlags);
csTerminator->Close();
delete csTerminator;
return bReturn;
};
```

This is fine so far, but there is one problem with that solution, and that problem is security. It is possible to secure a named pipe so strictly that not even the owner of the named pipe can connect—that is, the attempt to open the client end from the server will fail with the error ACCESS_DENIED. To ensure that **CancelCommunicationAttempt** works in this implementation, the owner of the pipe must have the appropriate permissions.

To circumvent the security problem, you can implement **CancelCommunicationAttempt** using polling pipes. Here is the implementation from NONBLOCK.CPP:

```
BOOL CServerNamedPipe::AwaitCommunicationAttempt(void)
{
 m_iStatusPending = STATUS_CONNECTION_PENDING;
 m_bInterrupted = FALSE;
 OVERLAPPED ol;
                                          // Block until a client connects.
 DWORD dwInstances,dwStateNew;
 if (!GetNamedPipeHandleState((HANDLE)m_hFile,&m_dwState,&dwInstances,NULL,NULL,NULL,0))
 {
  m_iErrorCode = GetLastError();
  return FALSE;
 };
 dwStateNew = (m_dwState & !PIPE_WAIT) | PIPE_NOWAIT;
 if (!SetNamedPipeHandleState((HANDLE)m_hFile,&dwStateNew,NULL,NULL))
 {
  m_iErrorCode = GetLastError();
  return FALSE;
 };                // I hope we'll never get here as that would be pretty serious re-cleanu
 ol.hEvent = m_hEvent;
 while (!m_bInterrupted)
 {
  ConnectNamedPipe((HANDLE)m_hFile,&ol);
  if (GetLastError()==ERROR_PIPE_CONNECTED) break;
 };
 if (!SetNamedPipeHandleState((HANDLE)m_hFile,&m_dwState,NULL,NULL))
 {
  m_iErrorCode = GetLastError();
  return FALSE;
 };                // I hope we'll never get here as that would be pretty serious re cleanu
 if (m_bInterrupted)
 {
  m_iErrorCode = ERROR_OPERATION_ABORTED;
  m_iStatusPending = STATUS_NOT_CONNECTED;
  m_bInterrupted = FALSE;
  return FALSE;
 };
 m_iStatusPending = STATUS_CONNECTED;
 return TRUE;
};
```

Phew! That is a lot of code. But, in fact, it is not too bad. You will notice that the **ConnectNamedPipe** call is still there, but now it is wrapped into a loop (the "polling" loop that we discussed earlier). The code will exit the loop as soon as the pipe is connected or the **m_bInterrupted** member variable (which is set from the **CancelCommunicationAttempt** member function) has a TRUE value. To ensure that the **ConnectNamedPipe** call does not block until a client has connected, the **SetNamedPipeHandleState** function is called to convert the pipe from a synchronous blocking pipe to a polling pipe before **ConnectNamedPipe** is called. The polling pipe is converted back to a synchronous blocking type after the polling loop has terminated.

There is one more catch to this solution, again related to security. For the **SetNamedPipeHandle** state to succeed, the pipe must be created with the PIPE_ACCESS_WRITE or PIPE_ACCESS_DUPLEX flag passed to **CreateNamedPipe**, even when the pipe is created as a read-only pipe. Likewise, **GetNamedPipeHandleState** requires the pipe to be created with the PIPE_ACCESS_READ or PIPE_ACCESS_DUPLEX flag, even for a write-only pipe. Thus, the implementation of **CServerNamedPipe::Open** for this solution always creates server ends of pipes with the PIPE_ACCESS_DUPLEX flags.

Although this solution works, it is somewhat awkward because it requires polling, which is not a good strategy in an operating system. You should probably ensure that the thread that executes **AwaitCommunicationAttempt** runs on a low priority so it will not impact performance too much.

The last implementation of **CServerNamedPipe::AwaitCommunicationAttempt** uses overlapped I/O. As in the previous two implementations, the **ConnectNamedPipe** call is used to wait for a client to connect. However, in this implementation, the last parameter to **ConnectNamedPipe** is the address of an **OVERLAPPED** structure. The **hEvent** member of that structure is specified as the handle in the **WaitForSingleObject** call that immediately follows **ConnectNamedPipe**. The catch here is that the handle can be set to the signalled state in two ways: either through successful completion of the **ConnectNamedPipe** call, or through the **CancelCommunicationAttempt** member executing in another thread. The following code is from OVERLAP.CPP:

```
BOOL CServerNamedPipe::AwaitCommunicationAttempt(void)
{
 m_iStatusPending = STATUS_CONNECTION_PENDING;
 m_bInterrupted = FALSE;
 OVERLAPPED ol;
 ol.hEvent = m_hEvent;
 ConnectNamedPipe((HANDLE)m_hFile,&ol);
 WaitForSingleObject(m_hEvent,INFINITE);

 if (m_bInterrupted)
 {
  m_iErrorCode = ERROR_OPERATION_ABORTED;
  m_iStatusPending = STATUS_NOT_CONNECTED;
  m_bInterrupted = FALSE;
  return FALSE;
 };
 m_iStatusPending = STATUS_CONNECTED;
 return TRUE;
};
```

This implementation is certainly the best one in terms of CPU utilization and reliability.

Reading from and Writing to Pipes

From the point of view of the Win32 API, a named pipe is pretty much like a file. Thus, the functions provided by the operating system to access the pipe are the same ones you use to access files: **ReadFile** and **WriteFile** (or, in the case of overlapped I/O, optionally **ReadFileEx** and **WriteFileEx**).

Let us look at the code for **CClientNamedPipe::Read** and **CClientNamedPipe::Write**, first for the synchronous blocking and polling pipe types:

```
void CClientNamedPipe::Write(const void FAR* pBuf, UINT iCount)
{
 unsigned long bytesSent;
 if (!WriteFile ((HANDLE)m_hFile, pBuf, iCount,&bytesSent, NULL))
 {
  DWORD dwErrorCode = GetLastError();
  RaiseException(dwErrorCode,0,0,NULL);
 };
};
UINT CClientNamedPipe::Read(void FAR* pBuf, UINT iCount){
 unsigned long uReturn; BOOL bRetCode; bRetCode = ReadFile((HANDLE)m_hFile,pBuf,iCount,&uRe
  DWORD dwErrorCode = GetLastError();    RaiseException(dwErrorCode,0,0,NULL);  return 0; }
```

As I discussed in the article "Communication with Class," exception handling allows errors to be handled on different levels, depending on what the condition is. Note that there is a degree of sloppiness in this implementation: The error I pass to **RaiseException** is the same value that is propagated from the kernel and, therefore, depends on the communication type. When the same application is run on another type of communication—say, sockets—the error return values will be different. Thus, ideally I would define custom error codes for each error condition I want to handle and translate the operating-system-provided error codes into those custom error codes.

Note that I do not provide an implementation of **Read** and **Write** for polling pipes; in the "polling" implementation of the **CNamedPipe** class, I use nonblocking pipes only to implement **AwaitCommunicationAttempt** and **CancelCommunicationAttempt**; for the most part, the pipe object is synchronous and blocking.

As I mentioned earlier, the implementation of **Read** and **Write** for asynchronous pipes is basically cheating, because I synchronize the asynchronous I/O operations immediately using **GetOverlappedResult**. This function blocks until the operation is completed (notice how the last parameter for **GetOverlappedResult** is set to TRUE to enable blocking):

```
void CClientNamedPipe::Write(const void FAR* pBuf, UINT iCount)
{
 unsigned long bytesSent;
 OVERLAPPED ol;
 BOOL bRetCode;
 ol.hEvent = m_hEvent;
 ol.Offset =0;
 ol.OffsetHigh=0;
 WriteFile ((HANDLE)m_hFile, pBuf, iCount,&bytesSent, &ol);
 bRetCode = GetOverlappedResult((HANDLE)m_hFile,&ol,&bytesSent,TRUE);
 if (!bRetCode)
  RaiseException(GetLastError(),0,0,NULL);
 };

UINT CClientNamedPipe::Read(void FAR* pBuf, UINT iCount)
{
 unsigned long uReturn;
 BOOL bRetCode;
OVERLAPPED ol;
 ol.hEvent = m_hEvent;
 ol.Offset =0;
 ol.OffsetHigh=0;
 ReadFile((HANDLE)m_hFile,pBuf,iCount,&uReturn,&ol);
 bRetCode = GetOverlappedResult((HANDLE)m_hFile,&ol,&uReturn,TRUE);
 if (!bRetCode)
 {
  DWORD dwErrorCode = GetLastError();
   RaiseException(dwErrorCode,0,0,NULL);
  return 0;   // to shut the compiler up
 }
 else return uReturn;
};
```

It seems somewhat phony to demonstrate an asynchronous solution just to synchronize it, although the immediate advantage of doing so is that by signalling the event, a pending I/O request can always be preempted for overlapped I/O, whereas there is no way to terminate a pending I/O operation on a synchronous pipe. Because I designed CommChat to be a multithreaded application from the beginning, I have not provided a "pure" asynchronous solution. The choice between multithreaded synchronous I/O and single-threaded asynchronous I/O has many dimensions and can involve endless hours of discussion. Watch for a future article on this subject.

The Exception Is the Rule

So far we have talked about the theory of named pipes, that is, how named pipes are supposed to work when they work. The deceiving calls **ReadFile** and **WriteFile** make us think that the communications should be fairly stable—after all, we do not expect those operations to fail on disks, right?

Unfortunately, networks are not that stable at all. When I wrote CommChat, I ran into all sorts of weird troubles, from interrupted network communications to buffer overflows to network time-outs to lost data—the whole spectrum of network malignancies showed up at my door.

To give you a first-hand impression of what kind of trouble you can encounter when programming networks (and I'm not even 1/100 on my way to having seen it all), here is my log of the last week. (This article was finished on a Saturday afternoon one week before its due date to Editorial.)

Monday morning. I finished writing "Communication with Class" and handed in the article for technical review. I have tested CommChat with a small communication between my two computers (about two lines—-works like a charm) and a transfer of a small file, also between my two machines. Cowabunga! Looks good! So up on a server it goes.

Monday afternoon. The first field reports from my technical reviewers come in. Nobody can establish a communication. I trace this down to a security feature (see the section "Security" earlier in this article) and fix it with the generous help of the Knowledge Base.

At the same time, I decide to give the file-transfer option a little challenge, and try to transfer a 2 MB file from one of my machines to the other one. After I discover that the file transfer works only when one of the machines runs under the control of a debugger with a breakpoint set (yes, it is this kind of problem!) and hangs otherwise, Nancy Cluts tries to chat with me using CommChat. The first three lines of sending data back and forth work fine, but all of a sudden the applications on both sides freeze and must be terminated. (Thank heavens Windows NT has secure, shielded applications; Windows for Workgroups would have needed rebooting at this point.) I make a note that I will look at the problem after having solved the file transfer problem.

It is 11 p.m. I decide to stop working for the day because my brain is mush. All I have been able to figure out is that one of the **ReadFile** operations returned error 59, "unexpected network error." Great.

Tuesday morning. I try the file transfer test again. I cannot reproduce the problem, but I decide to split the reads recursively into half because I suspect that there is a memory transfer problem. (I occasionally get the OUT_OF_MEMORY error.) Now I seem to be fine here, but I make another note that this problem needs to be addressed. To distract my mind from this thing, I decide to look into the chat problem. To my surprise, I discover that the problem seems to be an asymmetry between the client and the server: Transfers from the client to the server are nice and fast, but the very same operations (the very same code, in fact) from the server to the client is pitifully slow and eventually hoses the application, which eventually fails on the read. I send off a piece of e-mail to the Microsoft network gods asking for an explanation. In the meantime, I suspect that the problem might be my working with byte-sized pipes, so I look into message-sized pipes.

Tuesday afternoon. I get a call from a network god explaining to me that the asymmetry can be explained by the way the LAN Manager software is set up: Because it is assumed that most of the data transfer goes from the client to the server, the read buffer on the server side is set up to be very small. A registry parameter can be changed to increase the buffer size of the server.

Meanwhile, I have modified the class library to work on message-sized pipes instead of byte-sized pipes. This makes the protocol much easier and seems to relieve most of the file transfer problems. It is 9:30 P.M. Only 12 hours at work—I start to feel guilty, but I go home anyway.

Wednesday. I have finished modifying the class libraries, so the next thing I do is test the file-transfer option. Since I am no sissy, I try to send the 2 MB file again. I make a note that if the transfer is successful, I will contact somebody on a 386SX in the Australian subsidiary to see if I can receive a 12 MB file from him or her.

Unfortunately, I never get to that point. Although the reader side had been modified earlier to split the reads recursively if an out-of-memory error was encountered (as the avid reader will, of course, remember), message-sized pipes behave differently on the writer side than byte-sized pipes. Back in the days of byte-sized pipes (Monday and Tuesday), a write of 2 MB would bravely wait until the read had completed, but today the **WriteFile** call on the writer side fails with ERROR_MORE_DATA as soon as the reader on the other end fails the first time on ERROR_NOT_ENOUGH_MEMORY. According to the documentation, neither scenario should occur. So I send off another question via e-mail. After fiddling with this problem again (and starting to write this article), I call it a day just in time for the sunrise.

Thursday. I log onto my mail server and find a message in response to yesterday's question saying that the ERROR_MORE_DATA message is mismapped and should in reality be an ERROR_BUFFER_OVERFLOW message. Great. What does that mean for me? Now it seems as if the reader, even though I took care of the problem of a reader overflow, escalates its problem back to the writer, which fails again. I am probably looking at a rewrite of the protocol again.

There are several possible solutions for this problem, but I vaguely remember that my communication has not even been tested on more than five 486 machines that span maybe five offices in our building (probably not even one network router apart). Shouldn't I therefore provide some kind of time-out or a guaranteed error return that would retry the operation periodically and fail after *n* times? This is what happened frequently under Windows 3.1, normally ending in the "beeping death." I start to wonder, how do those huge database transfers—more than half the world away, 24 hours a day, over telephone lines—work in an even remotely stable manner if my communication does not survive a 2 MB transfer between two like machines in the same office?

When I sent e-mail to internal people about those problems and how to solve them, I basically received two kinds of reactions. The initial reaction was silence. I posted a few questions on the appropriate aliases, and when my Inbox began to attract spider webs, I started to wonder if my message had even been sent in the first place.

Then I received a message from somebody who works in internal computing. He is in charge of maintaining those huge data transfers all around the world in our company's heterogeneous network, so if anybody knows, he would.

His message was pretty disillusioning. He said that according to his experience, a named pipe should not be relied on to stably transfer major amounts of data over a network. He has resolved to use the pipes only for transferring messages back and forth, and the real work is done by shared files on a server. Typically, an application programmatically logs on to a remote server, writes the data to a file on that server, and uses a named pipe to send a message to that server, informing it that the data is ready for pick-up. I make a note to research why access to a remote disk should be any more reliable than writing to a remote pipe. Miraculously, it is Friday morning by now. On Friday afternoon, I have a fairly stable application, which, however, has still to prove reliable in a major field test. The good news is that, due to the C++-based architecture, more reliable communication techniques can easily be plugged in if necessary.

What do we conclude from this captain's log? That named pipes are bad or unreliable?

Nope. We have to keep in mind that named pipes are not network protocols, they are simply interfaces. Protocol suites accomplish data transfers over a network, and these are almost totally unrelated to the interface. Although named pipes do a little work in breaking up a data communication, the data will be wrapped in a network protocol, and, eventually, will end up on the physical network cable. Some protocols provide very reliable and fast communications, whereas others do not go out of their way to verify that all of the data has been transmitted or received correctly.

How are interfaces associated with protocols? It depends on the interface. There is no way to programmatically associate a named pipe with a certain protocol; that is, if multiple protocols are available for network communications on a given machine, an application cannot ask a named pipe to communicate over a specific protocol. The protocol that a specific communication uses is determined by two factors: the binding order on the two machines (accessible from the Windows NT Control Panel, Network applet, Network Bindings dialog box) and the common bindings on both machines. The two computers "negotiate" a protocol by finding a common binding, moving down the binding list in ascending order of priority.

Exception Handlers

As I mentioned earlier, structured exception handlers are used to handle error conditions on several levels. The main usage of structured exception handlers in CommChat is to deal with buffer overflows. As I mentioned before, a **WriteFile** operation on a message-sized pipe will fail with ERROR_MORE_DATA (actually a mismapped STATUS_BUFFER_OVERFLOW error from the kernel) when the size of the buffer passed to **WriteFile** exceeds 64K. The file transfer protocol catches this condition by disassembling a larger file transfer into smaller chunks.

If a connection is interrupted, **ReadFile** and **WriteFile** return with an error, and **GetLastError** returns ERROR_PIPE_NOT_CONNECTED. Currently, I do not process this condition in a structured exception handler but instead have the **Read** member return 0, which will be caught by the protocol as a termination-connection message.

Each call in an application or protocol that will eventually call into the **Read** or **Write** member of a communication object should be wrapped into a **_try** clause so that either a protocol or an application can gracefully recover from an error condition.

What Other Cool Things Can You Do with Pipes?

One of the favorite uses of pipes is to redirect input from and output to console applications programmatically, even between remote machines. This technique is demonstrated in the remote shell sample RSHELL, which can be found in the MSDN Library (Sample Code; Product Samples; System Toolkits, DDKs, and SDKs; Win32 SDK Samples). Capturing the output of a child application via a named pipe is shown in the INHERIT sample, mentioned in a Knowledge Base article (Q84082) and found in the Library (Sample Code, Product Samples, Languages, Visual C++ 1.0 (32-bit) Samples).

Most samples that employ named pipes also use multiple threads to service several named pipes simultaneously. There are several samples from the Win32 SDK in the MSDN Library that show these techniques: NPCLIENT, NPSERVER, and SIMPLE (Sample Code; Product Samples; System Toolkits, DDKs, and SDKs; Win32 SDK Samples). More information about the SIMPLE sample application can be found in Knowledge Base article Q99460.

Good articles to read for more background information are Knowledge Base Q95900, "Interprocess Communication Under Windows NT," which gives you an overview of network APIs; Knowledge Base Q65125, "Developing Network Applications for Windows"; and Scot Gellock's Tech*Ed paper, "Developing Distributed Applications with Windows NT" (Conferences and Seminars; Tech*Ed, March 1994; Windows NT).

It is also possible to use named pipes in a one-shot operation, that is, to send stuff through a named pipe without opening and closing the pipe explicitly. The calls to do so are **CallNamedPipe** and **TransactNamedPipe**.

Because named pipes are native Windows NT objects, and can therefore be used as arguments to **WaitForSingleObject** and **WaitForSingleObjectsEx**, remote named pipes could even be used to provide some very rudimentary remote thread synchronization objects. Maybe I will work that idea into a future article.

Summary

Named pipes provide a fairly convenient mechanism by which to implement data transfer over a network. Through the Win32 API, which treats named pipes pretty much like files, the implementation of the **CNamedPipe** object as a derivative of **CFile** is fairly straightforward.

Depending on whether a pipe is created as a message-mode or byte-mode pipe, a communications protocol can be either trivial (in the current implementation of **CNamedPipe** using message-mode pipes, all calls from the application to **CProtocol::Read** and **CProtocol::Write** are basically routed right through to the pipe) or can require a little work.

Both a blessing and a curse of named pipes is that they operate on a fairly high level of abstraction in the networking hierarchy. This is good because it relieves the programmer from a lot of details about the underlying network, but it is also a potential problem because many of the possible causes of failures inside the network are not easily traced—a pipe may simply return an error code from an I/O operation, after which it is hard to figure out what the cause might be, and it might not always be straightforward to recover from the failure.

It turns out that using lower-level network communication strategies allows for a greater control of the details of the communication, such as packet sizes, time-out parameters, communication retries, or even routing paths. We will look at those issues in future articles.

Named pipes can also be used to transfer data within heterogeneous networks, that is, all networks that support named pipe servers on their nodes. You can even utilize named pipes in Windows for Workgroups; however, that platform only supports client ends of named pipes (that is, you cannot use **CreateNamedPipe** on those machines, only connect to named pipes created on machines that run server software).

After having discussed some of the typical problems occurring in network programming, I will use future articles in this series to discuss how other network communication mechanisms solve those problems.
