


*Interprocess Communication Technical Articles***Plugs and Jacks: Network Interfaces Compared**

Ruediger R. Asche
Microsoft Developer Network Technology Group

Created: October 21, 1994

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

Sample Files:

-  3188.exe

[Load Sample Solution](#)

[Copy All Files](#)

[Help](#)

Abstract

This article wraps up the series of articles on network programming interfaces (named pipes, Windows® Sockets, and NetBIOS) by comparing the functionalities of these three interfaces. The article also discusses the relationship between networking interfaces and protocols.

Acknowledgments

Since this article is the last in a series of five articles in which, for the first time in my career as a full-time technical writer, I started from ground zero to work on an issue, I feel that I should mention the names of the individuals at Microsoft who set aside some of their valuable time to review the articles and answer my rookie questions.

I am very much in debt to the expertise and patience of Carlos Alayo, Craig Link, Todd Lucas, Larry Osterman, David Treadwell, and Colin Watson, who didn't get anything but a T-shirt each for their reviews. I am always grateful to the other personalities of Dr. GUI—currently Dale Rogerson, Nigel Thompson, Nancy Cluts, Dennis Crain, Kenneth Lassesen, and Renan Jeffereis—who do NOT get T-shirts because they get paid for reading my articles (among other things).

Also helpful in putting together this series were Dan Perry, Robert Nelson, and Douglas Adams.

Road Map

This article is last 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

We now have yet another C++ class hierarchy—**CCommunication** and its derivatives, **CNamedPipe**, **CSocket**, and **CNetBIOS**—which allows your application to establish and maintain network communications fairly painlessly, regardless of the network interface(s) provided on your computer.

If the interfaces are pretty much interchangeable, why use any one in particular? Couldn't your C++ application simply use the **CCommunication** class, check to see which interface is supported, and use any derived class that is available?

Sure enough, but remember that the classes I provided implement only a "minimal set" of common features—namely, the file primitives **Open**, **Read**, **Write**, and **Close**, and a few

network-specific extensions, such as **AwaitCommunicationAttempt**. Depending on what your application needs to accomplish, some features provided by only one or two of the interfaces can greatly simplify your application design and make you choose one interface over the other, possibly sacrificing the option to switch to other interfaces.

In this article, we will deal roughly with two areas: First, what do the interfaces have in common, and second, what are the unique features of each interface (in other words, how do the interfaces differ from one another)? For those of you on a tight schedule, I've summarized the contents of the entire article in a chart at the end. Before we dive into these questions, I will quickly discuss a fourth interface for which I did *not* provide a C++ class: NetDDE.

To NetDDE or Not to NetDDE?

If you compared this version of the CommChat sample application to the version provided in the October edition of the MSDN Library, you'd find that I promised a NetDDE class in the Select Communication menu. This menu option is silently swept under the carpet in this release. Why did I wimp out on NetDDE?

Well, for beginners, there are already two versions of a NetDDE-based chat application available: One (CHAT.EXE) ships with both Windows for Workgroups and Windows NT®, and the other (NDDECHAT.EXE), written by Craig Link, is included in the Platform SDK with complete source code.

Nevertheless, after I finished "[Aristocratic Communication: NetBIOS](#)," I decided to give NetDDE a shot—after all, with Craig's free gift, it would only be a matter of cut-and-paste to create a **CNetDDE** class library from the networking heart of NDDECHAT, right?

Wrong. After a lot of coding, I eventually decided not to pursue a **CNetDDE** class further, because NetDDE strictly refuses to have itself abstracted into a communication class that follows the open/read/write/close scheme for the following two reasons: First, NetDDE is designed to interact very closely with the other modules of a NetDDE-aware application (thus, the logical separation between the user interface, the data processing logic, and the communication module that CommChat is designed upon does not easily incorporate the NetDDE model); and second, the way the dynamic data exchange management library (DDEML) is set up does not work too well with instance-based communications. That means that under the DDEML, all communications begin with a function call on one side and end in a "message-loop" type function on the other side, and it is non-trivial to sort out the incoming communications on the receiving side and assign them to the appropriate communication objects. Even if every communication object initialized a new instance of a DDEML message loop, it would not be easy to dynamically associate a message loop with a communication object.

Another reason why NDDECHAT is better designed than CommChat to work with NetDDE is that NetDDE, unlike the other interfaces we have talked about so far, is server-centered; that is, DDE has built-in provisions to let the server automatically inform the client of changes in data on the server. In CommChat, a secondary thread constantly listens on an incoming communications line because it does not make any assumptions as to when incoming data is ready (that is, it does not know when the user on the other side types something in his or her instance of CommChat). Under NetDDE, this is not necessary, because the DDEML can be advised to call the application back as soon as the other side has new data ready—NDDECHAT very nicely shows how this can be done.

Note that the way the DDEML provides for asynchronous communications is different from the notion of asynchronous I/O that we have encountered so far—in named pipes, sockets, and NetBIOS, an asynchronous input call must first be explicitly submitted to put the application in a state in which it can accept the input at a later time. As soon as the input is processed, the application must submit another asynchronous input call in anticipation of the next input.

NetDDE is different. As soon as an application has successfully registered with the DDEML, the "message loop" function constantly waits for notifications, typically posting a message to an application window whenever input from a connected application is ready. This setup makes it

very easy to monitor infrequent input from another application, because it is the DDEML that monitors input from the other side, not the application.

Because of these issues and the fact that the DDEML under Windows NT was not designed to support access to the same library instance from multiple threads, I had to drop support for NetDDE in CommChat.

What Do the Interfaces Have in Common?

All interfaces have to deal with a common number of issues. Let us first define those issues, and then examine how different interfaces deal with the issues.

The first question is name resolution: How does a communication application address another machine? Normally, each machine is assigned a unique name—how does that (logical) name map to a (physical) machine? If a machine is not identified by a name, how does a process determine the address of that machine?

Second, if the first problem is solved, how can a machine concurrently serve different server applications? In other words, if BEAKER runs two communication applications—say, a database server and a mail server—how does BEAKER determine which network packets belong to an ongoing database query, and which packets belong to a mail conversation?

Third, if the second problem is solved, how can the same network-aware application service communications with more than one client? Let us assume that BEAKER is a database server from which both GNORPS and SCHTONK retrieve data concurrently. How does the database server application running on BEAKER sort out network packets that belong to GNORPS and SCHTONK, respectively?

Notice that these three questions define three levels of "granularity" in network communications: The first issue involves addressing a remote machine, the second issue involves addressing a particular application on that machine, and the third issue involves distinguishing between different clients talking simultaneously to the same server application on that machine.

The fourth issue is asynchronous vs. synchronous communications. This is really a non-issue for CommChat because CommChat is multithreaded. However, in a single-threaded application, you will need to deal with the possibility that a network request may take an arbitrary amount of time to complete; thus, all interfaces offer some kind of support for asynchronous network I/O, that is, function calls that return immediately but complete later on.

Also, some interfaces allow connectionless communication; that is, it may be possible for two machines to send data back and forth without explicitly establishing a communication first. The default is usually connected communication, in which every packet that is sent belongs to a predefined connection that defines both sides of the communication.

Note that the preceding discussion implies that each communication interface was designed to communicate with remote computers only. That is not true; the interfaces can establish and maintain communications between remote machines as well as between distinct processes on the same machine. However, in the remainder of this article, I will focus only on communications that take place between distinct machines.

Name Resolution

Named pipes solve the name resolution problem in a fairly stringent way: A client can establish a connection only if the other side (the server) has created the server end of the pipe. The server must be addressed by a static name that the server machine was assigned when it was started.

In the sockets interface, a communication channel is opened using a remote address whose representation depends on the underlying transport protocol. The **gethostbyname** and **gethostbyaddress** services map a machine name to a network address and vice versa. The actual assignment of network addresses may take different forms, depending on the protocol used.

In many cases, a network address (for example, an IPX node address or an IP address; see "[Power Outlets in Action: Windows Sockets](#)" for details) is nothing but a logical construct; the network software must also know how to map this address to a physical network address. A physical network address is, in most cases, a unique numeric value that is hard-coded into the network adapter card. The address resolution strategy depends very strongly on the network protocol and will not be discussed here.

NetBIOS probably has the most flexible approach to name resolution because an application can register a computer name dynamically at run time. As soon as a successful **AddName** NetBIOS call is submitted, the NetBIOS software broadcasts a request to add the name to the network. If another computer has already registered that name, the **AddName** call fails. When a machine tries to establish a connection with another computer, it again broadcasts the target machine name over the network. The machine that has that name registered answers by providing its own network adapter address so that the communicating machine can address the target machine without broadcasting in the future.

Sorting Out Communications

Now that the first problem—addressing machines—is solved, the next problem involves sorting out multiple communications on the same machine.

You will recall that named pipes solve this problem through unique pipe names—for example, `\\beaker\pipe\pipe1` serves a different communication than `\\beaker\pipe\pipe2`. Sockets discriminate between communications using port numbers. In NetBIOS, on the other hand, each application that wants to communicate over the network registers a unique name under which it can be addressed.

NetDDE sorts out communications using logical NetDDE shares. Each NetDDE communication is maintained via an NDDE\$ logical share on the server machine (for example, GNORPS establishes a NetDDE communication with SCHTONK by connecting to the logical share `\\SCHTONK\NDDE$`). DDEML then discriminates between server applications through unique topics registered by each server application.

Instancing Communications

Now that we know how several processes on the same machine can use the network concurrently, how can the same network application service multiple clients at the same time? Let us assume, once again, that BEAKER runs a database server application. Every reasonable database server must be able to service multiple clients concurrently; thus, the same code is executed for each instance of a communication. Because the code does not change, it doesn't really make sense to launch a separate process for each client, so the application must be able to discriminate between multiple clients in the same code.

Named pipes address this issue by allowing multiple instances of the same pipe to be active. The Windows Sockets library understands the concept of a pending connections queue: On the server side, a call to **accept** removes one pending connection attempt from the queue, so that multiple connections are allowed on a single socket. Note that transmission control protocol/Internet protocol (TCP/IP) defines a connection using four components—the addresses of the two communicating machines and their respective port numbers—so the library can discriminate between communications rather easily.

NetBIOS addresses this issue by returning a unique local session number (LSN) for each established communication. Under NetDDE, the server application is responsible for administering multiple connections. As soon as the server has initialized DDE with a message-processing function, that function receives an XTYP_CONNECT message whenever a new client attempts to connect to the server. It is up to the server application to keep track of the multiple communications.

Asynchronous vs. Synchronous Commands

The third area in which I would like to compare the interfaces is synchronous vs. asynchronous

communications. Recall that the issue here is that single-threaded applications may be blocked for an arbitrary amount of time while waiting for a network command to complete.

You have three choices to cope with this problem: (1) allow the I/O call to take whatever time it takes to complete, potentially making the application look as if it is blocked indefinitely to the user; (2) delegate a blocking I/O call into a secondary thread; (3) use asynchronous I/O in a single thread. All interfaces we have looked at provide asynchronous I/O as an option.

Named pipes can be opened in either nonblocking or overlapped mode to achieve asynchronous processing (see ["Garden Hoses at Work"](#) for details), and sockets can either be opened in overlapped mode or used with the asynchronous Windows extensions. Note that the sockets hierarchy provided by MFC—the **CAsynchSocket** and **CSocket** classes—lend themselves rather naturally to single-threaded, asynchronous processing (please see the article ["Power Outlets in Action: Windows Sockets"](#) for details).

NetBIOS calls can be submitted in asynchronous mode, where an asynchronous callback routine that is invoked as soon as the asynchronous command completes is supplied to the network control block (NCB). Asynchronous NetBIOS commands were a big issue under 16-bit Windows—although the nonpreemptive nature of the operating system practically enforced the usage of asynchronous over synchronous commands, it also provided a number of challenges concerning the callback (for example, an asynchronous callback routine had to be explicitly page-locked in memory).

NetDDE leads itself very naturally to asynchronous I/O, as I mentioned in the section "To NetDDE or Not to NetDDE?" earlier in this article.

So what should you do?

The controversy over whether to use asynchronous single-threaded or synchronous multithreaded I/O assumes a fairly religious flavor at times. CommChat demonstrates multithreaded communications while NDDECHAT (shipped with the Platform SDK) accomplishes pretty much the same thing using NetDDE in a single-threaded application. The CHATSRVR and CHATTER sample applications that are shipped with MFC also demonstrate a chat-like application using asynchronous sockets in a single-threaded environment.

Your choice depends on several factors:

- **Portability.** What additional platforms do you want your application to run on? All NetBIOS implementations support asynchronous communications, but not all platforms that support NetBIOS support multiple threads, so if you need to be compatible with applications written for 16-bit Windows and have already voted for NetBIOS, you might decide for asynchronous I/O. On the other hand, if your application also targets, say, UNIX® workstations, you are more likely to encounter multithreaded platforms than systems that allow asynchronous communications on sockets. In that case, you might want to opt for synchronous, multithreaded, socket-based communications. Other platforms may lead to different decisions.
- **Ease of application design vs. machine load.** It is my personal theory that synchronous, multithreaded I/O is much easier to code than asynchronous I/O because generally, at *some* point, the results of an asynchronous I/O operation must be synchronized with the main "thread" of computation. Thus, designing asynchronous I/O is pretty much a matter of figuring out how to most effectively fill in the "holes" between the submission and the completion of a request. In contrast, in a synchronous multithreaded application design, a primary thread may simply delegate the I/O into the background, do whatever it is about to do, and at some point later synchronize with the result.

Also, some implementations of asynchronous I/O interfaces restrict the functions that can be called in an asynchronous callback routine.

On the other hand, multithreading isn't free. If your application is designed to serve multiple communications concurrently, a liberal use of threads may seriously affect the load of the computer. Asynchronous I/O, on the other hand, may be relatively cost-effective. For example, in Windows NT, the I/O system and the kernel architecture provide

asynchronous I/O in a very effective manner.

Connection-Oriented vs. Connectionless Communications

In CommChat (as in many applications that exploit communication techniques), it makes sense to establish a communication explicitly between two machines before any data is transferred. However, at times it is not really necessary to open a communication first. Imagine that you want to copy a file from a remote share to your hard drive. As a user, you can do this in two ways:

1. Establish a permanent connection between the two machines—for example, by using file transfer protocol (FTP) or connecting to a remote share. You can then copy the file from the logical connection. The remote connection can be terminated when it is no longer used.
2. In your copy command, use universal naming convention (UNC) names, which allow you to access remote data without explicitly establishing a connection. For example:

```
copy \\BEAKER\public\bizarre.txt c:\tmp
```

These two options are analogous to connection-oriented vs. connectionless communications. Some interfaces allow you to transfer data without first establishing a connection, roughly similar to the second option above. This method may be useful when small amounts of data are transferred.

To establish connectionless communication using sockets, you omit the **listen** call on the server side and the **connect** call on the client side. Instead, you use the **sendto** and **recvfrom** calls to transmit data to, and receive data from, another socket without establishing a connection first.

Under NetBIOS, the datagram services provide the same capability. Named pipes do not support connectionless communications, but the **CallNamedPipe** function does something along those lines. The Win32 API also defines *mailslots*, which allow an application to broadcast a message to all machines that have created a specific mailslot without explicitly establishing a connection.

And Now for the Differences

We've discussed how the interfaces deal differently with common issues (so we can disregard the differences by abstracting them away in the **CCommunication** class hierarchy). Now let's look at features that are specific to interfaces.

Network Semantics

NetBIOS is the only one of the four interfaces that defines more than a method for formulating network communications in function calls—it also defines some network semantics. The NetBIOS name table mechanism (in particular, the group name mechanism) provides a built-in way to logically group computers and resolve names.

A *broadcast* is a simultaneous network transmission to multiple target processes. Sockets allow transmissions of broadcast datagrams on networks that support broadcasting (such as the Internet, which defines local broadcast addresses). Named pipes do not support broadcasting, but the mailslot feature in the Win32 API is frequently used to accomplish something similar to broadcasting.

The broadcasting and name table architecture of NetBIOS is immensely powerful. Emulating that functionality using other interfaces would require a good amount of coding of LAN Manager management software.

A word of warning: Broadcasts should not be used too liberally, especially in large networks, because each broadcast significantly affects the network load. All protocols have to broadcast messages over the network at some time (for example, TCP/IP uses broadcasts implicitly to associate a network address with a physical network adapter address), but NetBIOS is pretty much the only interface that explicitly defines broadcasting semantics. If you use broadcasting,

do so with great care—your network administrator might be taking a voodoo class right now. .

Security Concerns

In Windows NT, named pipes and NetDDE shares are securable objects and can, therefore, be easily protected against misuse. An application such as CommChat can easily be misused by users who happen to know the port number and Internet address (for sockets) or the name CommChat has registered in the name table (for NetBIOS). Such a user could simply write a small application that connects to CommChat and fakes a known user. With CommChat, this is not a big deal because it is up to the user on the target machine to respond to a communication request, but assume that you used the **CCommunication** class library to write an automated data server that, say, downloads sensitive data to users without supervision. With named pipes and NetDDE, you could rely on the built-in security mechanisms to restrict access to the data (unauthorized users wouldn't even be able to open a connection under this scheme), whereas in a solution that uses NetBIOS or sockets, it would be the application's responsibility to implement a security scheme to prevent misuse of the data. Please refer to the article series beginning with "[Windows NT Security in Theory and Practice](#)" for details on how to use the Windows NT security system.

Communication Tracking

Both sockets and NetBIOS provide predefined functionalities to retrieve the name of the connected machine, given only a connection. In the sockets specification, the **getpeername** service provides this functionality; NetBIOS does this using the NCB **CallName** field.

Using named pipes, a client needs to explicitly communicate its name to the server. Note that the **GetNamedPipeHandleSate** function and several security functions can be used to retrieve information about the connected user.

Platform Compatibility

If your main focus is on writing applications for the Windows environment, you might lean towards NetDDE or Windows Sockets because both of these interfaces have built-in mechanisms that integrate well into the message-based Windows environment. If you write console applications more often than graphical user interface (GUI) applications, all interfaces except NetDDE will serve you well, although with sockets, you will be restricted to the "standard" sockets functionality (no Windows extensions).

Your choice of network interfaces will also be determined by compatibility with existing software. For example, as I mentioned in "[Power Outlets in Action: Windows Sockets,](#)" a socket communication can bind to a number of well-known ports, which correspond to predefined services that other machines may have installed (for example, FTP). If your application wants to be able to address these functionalities on a remote machine, it would need to work with sockets because neither NetBIOS nor named pipes allow you to access ports explicitly.

System Support for Starting Up Servers

With all interfaces, a server application must register the communication objects it wishes to make available for clients to use. Unless the code that creates server objects resides in a service (or in an application that is automatically started, for example, an application that is in the startup group), a server application must be explicitly started before a client can establish a communication.

NetDDE is the only one of the four interfaces that registers objects (NetDDE shares) along with the applications that create the shares; thus, NetDDE does not require that you explicitly start the application on the server machine. The NetDDE-related services can implicitly launch a server application as soon as a client application tries to communicate via a NetDDE whose associated server has not been started yet. This is very convenient because it does not require you to write a service or put an application into the startup group.

Assigning Protocols to Interfaces

We have discussed three interfaces (named pipes, sockets, and NetBIOS) and swept one (NetDDE) under the carpet. As I mentioned earlier, except for slight semantic differences, these interfaces are exchangeable, and their usage is pretty much independent of the underlying protocol.

In a very few cases, your application must be aware of the underlying protocol and be able to control it to accomplish a data transmission. Imagine that your computer is attached to a network that has some machines that run exclusively on TCP/IP and other machines that have only IPX installed. Let us further assume that your computer (call it GNORPS) has both protocols installed. How would GNORPS be able to specifically communicate with SCHTONK, knowing that SCHTONK has only TCP/IP installed, and address ROMPER, which runs IPX only, with the same interface?

Only NetBIOS and Sockets allow you to dynamically assign a protocol to a specific communication: NetBIOS via the LANA_NUM field in the NCB, and Sockets via the **SOCKADDR** structure. Neither named pipes nor NetDDE provides that functionality.

NetBIOS is probably the most flexible of the interfaces because it allows application programs to select network adapter cards for specific communications.

Summary

The following table summarizes everything we have discussed in this article.

Feature	Named Pipes	Sockets	NetBIOS	NetDDE
Name management	Defined by pipe name syntax	Depends on network addressing scheme	Defined by name table service	Defined by DDE and network architecture
Sorting out multiple communications	Unique pipe names	Unique port numbers	Unique names in name table	Unique topic names
Same service serving multiple clients	Multi-instanced pipes	Unique communications via address/port/address/port tuples	Unique LSNS	Application-defined
Asynchronous option	Overlapped I/O on a pipe	Overlapped I/O or Windows extensions	Asynchronous NCD calls	Through "message-loop"
Connectionless communication option	Approximated through CallNamedPipe function	Yes, via datagrams	Yes, via datagram services	No
Broadcasts	Approximated via mailslots	Supported if supported by the network transport	Part of the NetBIOS specification	No
Security support	Native under Windows NT	Not provided	Not provided	Native under Windows NT
Communication tracking	No	Yes, via getpeername	Yes, via name table	Yes

		function		
Fits into the Windows messaging model?	No	Yes, via Windows extensions	No	Native
Service provision	No, unless pipe is created in a service	No, unless socket is created in a service	No, unless NetBIOS name is registered in a service	Yes (NetDDE service starts server applications on demand)
Dynamic assignment of protocols or network adapters to communications	Not provided	Yes, via SOCKADDR interpretation	Yes, via LANA_NUM field in the NCB	Not provided
Header file(s) to include (Visual C++ 2.0)	%	WINSOCK.H	NB30.H	NDDEAPI.H DDEML.H
Libraries to link to (Visual C++ 2.0)	%	WSOCK32.LIB	NETAPI32.LIB	NDDEAPI.LIB

Conclusion

The networking world is rather confusing—it provides a significant number of interfaces, protocols, network models, and services, and almost any combination of these. As an increasing number of these interfaces and protocols become integrated, writing network-aware applications or network software becomes less of a matter of choosing an appropriate interface than writing to a standardized superset (such as the **CCommunication** class hierarchy I presented).

However, the details in which the network interfaces differ may make it worthwhile to choose one interface over the other, depending on the particular task that your network-aware application needs to accomplish.

For further reading on networking under Windows, I very strongly recommend *Windows Network Programming* by Ralph Davis (Reading, Mass.: Addison-Wesley, 1993). This book provides a very comprehensive and thorough coverage of networking under Windows and Windows NT.

[Send feedback to Microsoft](#)

[© 2003 Microsoft Corporation. All rights reserved.](#)
