Named Pipes, Sockets and other IPC

Mujtaba Khambatti {Mujtaba.Khambatti@asu.edu}

Arizona State University

Abstract

The purpose of this paper is to discuss interprocess communication in the context of Windows 2000 and UNIX (Berkeley UNIX, System V, and SunOS). Special emphasis will be given to the system mechanisms that are involved with the creation, management, and use of named pipes and sockets. There will also be a discusssion of remote procedure calls and signals in the context of the UNIX environment.

1. Table of Contents

Abstract 1

1. Table of Contents 2

2. Named Pipes in Windows 2000 3

2.1 Named Pipes Background 3

2.2 Named Pipes Definition 3

2.3 Architectural Description 3

2.4 An Overview of the LANMAN Architecture 3

2.4.1 LANMANWorkstation 3

2.4.2 LANMANServer 3

2.5 Relation with other IPC 4

2.6 Impersonation 4

3. TCP/IP Sockets over Windows 2000 5

3.1 Windows Sockets Background 5

3.2 Definition of Socket 5

3.3 Windows Sockets 2 and TCP/IP 5

3.4 Uses for Sockets 6

3.5 Named Pipes Vs TCP/IP Sockets 6

4. Comparison between IPC over Unix and IPC over Windows 2000 7

4.1 UNIX IPC Background 7

4.1.1 A note on UNIX Orientation for IPC 7

4.2 Types of UNIX IPC and Definitions 7

4.2.1 Local IPC 7

4.2.2 Remote IPC 7

4.2.3 Differences in IPC models in versions of UNIX 8

4.3 Streams in UNIX 8

4.3.1 Pipe Streams in UNIX 8

4.3.2 Named Pipe Streams in UNIX 8

4.3.2.1 Using Named Pipes in UNIX 8

4.4 Sockets in UNIX 9

4.4.1 Uses of Sockets in UNIX 9

4.4.2 UNIX Socket Programming Interface 9

4.5 Remote Procedure Calls in UNIX 10

4.6 Signals in UNIX 10

5. References 11

2. Named Pipes in Windows 2000

2.1 Named Pipes Background

Named pipes provide a high-level connection-oriented messaging by using pipes. Connection-oriented messaging requires that the communication occur over a virtual circuit and maintain reliable and sequential data transfer. A pipe is a portion of memory that can be used by one process to pass information to another. A pipe connects two processes so that the output of one can be used as input to the other. This technique is used for passing data between client and server. Named pipes are based on OS/2 API calls, which have been ported to the *WNet* APIs. Additional asynchronous support has been added to named pipes to pass data between client/server applications. Named pipes are included to provide backwards compatibility with *Microsoft® LAN Manager* and related applications [SYBEX, SoloRuss, NetArch, PlatSDK1].

2.2 Named Pipes Definition

A *named pipe* is a named, one-way or duplex pipe for communication between the pipe server and one or more pipe clients. The term *pipe*, as used here, implies that a pipe is used as an information conduit. Conceptually, a pipe has two ends. A one-way pipe allows the process at one end to write to the pipe, and allows the process at the other end to read from the pipe. A two-way (or duplex) pipe allows a process to read and write from its end of the pipe. Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network. Any process can act as both a server and a client, making peer-to-peer communication possible. As used here, the term *pipe server* refers to a process that creates a named pipe, and the term *pipe client* refers to a process that connects to an instance of a named pipe [SYBEX, SoloRuss, NetArch, PlatSDK1].

2.3 Architectural Description

The Named Pipes ties in with the I/O subsytem and can appear to the user as nothing but another file system. This is because Named pipes are written as file system drivers. Local processes can also use named pipes. As with all other file systems, remote access to named pipes is accomplished through the Common Internet File System (CIFS) redirector. A redirector intercepts file input/output (I/O) requests and directs them to a drive or resource on another networked computer. The redirector allows a CIFS client to locate, open, read, write, and delete files on another network computer running CIFS [PlatSDK1].

2.4 An Overview of the LANMAN Architecture

The Microsoft® LAN Manager is used for networking. Two modules, *LANMANWorkstation* and *LANMANServer*, accomplish this. These two components, with the help of several more we will identify, provide most of the functionality of the OS/2 version of LAN Manager available today. Both of these modules execute as 32-bit services [Clark93].

2.4.1 LANMANWorkstation: This module is really in two pieces (see Figure 1). The LANMANWorkstation component provides the user-mode interface. The other component is the RDR, or *Redirector*. This component is a *File System Driver* (FSD) that actually does the interaction with the lower layers of the protocol stack. *Multiple UNC* (Universal Naming Convention) *Provider* (MUP) is an interesting entity that runs in kernel-mode memory. The most productive way of thinking of MUP is as a resource locator. The types of resources it locates are UNC names. A UNC name is a naming convention for describing servers, and sharepoints on those servers, on a network [Clark93]. A typical UNC name would appear as:

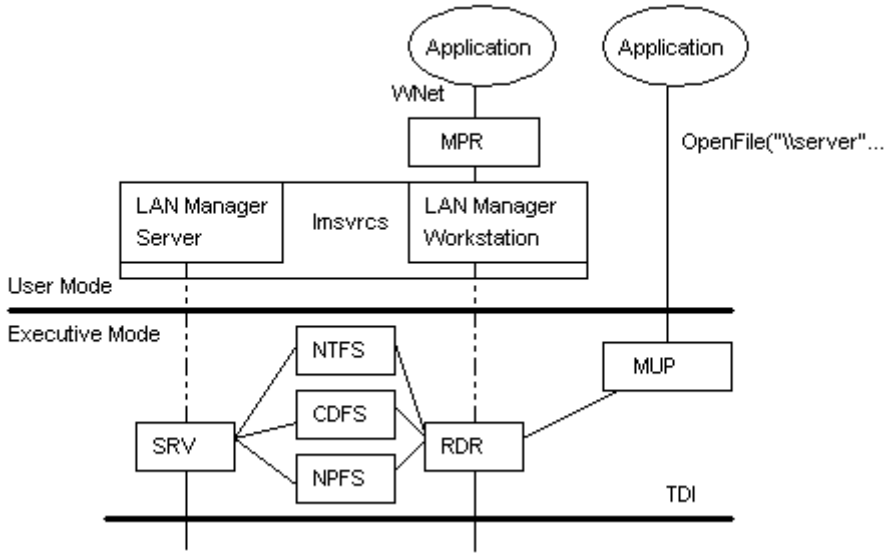
\\server\share\subdirectory\filename

2.4.2 LANMANServer: This module is much like the LANMANWorkstation module. It is a service that runs in the *lmsvrcs* process. Unlike the workstation component, it is not dependent on the MUP service, since the server is not a UNC provider. It doesn't attempt to connect to other machines, but other machines connect it to. Like LANMANWorkstation, it is composed of two parts: the LANMANServer component and the *SRV* component. The

SRV component handles the interaction with the lower levels and also directly interacts with the other file system devices to satisfy command requests such as file read and write.

Above the redirector and server components live the applications. As with our other layers, we want to provide them with a single unified interface to develop to, independent of the lower-layer services. This is done through two mechanisms. We have already looked at the first—MUP.

The other is the MPR, or *Multi-Provider Router*. The MPR is much like the MUP. This layer takes in WNet commands, finds the appropriate redirector based on the handle, and passes the command to that redirector for communication onto the network. In addition to I/O calls such as Open and Close, Win32™ contains a set of APIs called the WNet API. These are APIs that were ported over from Windows 3.1 network calls. Most of these calls deal with establishing remote connections [Clark93].



1 Figure 1: The LANMAN Workstation module divided into two parts [Clark93]

2.5 Relation with other IPC

The six IPC mechanisms provided by Windows NT are: named pipes, mailslots, NetBIOS, Windows Sockets, Remote Procedure Calls (RPC), and Network Dynamic Data Exchange (NetDDE). Named pipes and mailslots provide backward compatibility with existing LAN Manager installations and applications. This is also true of the NetBIOS interface. Windows Sockets is a Windows-based implementation of the widely used sockets programming interface created by the University of California at Berkeley. RPC is compatible with the Open Software Foundation/Distributed Computing Environment (OSF/DCE) specification for remote procedure calls. NetDDE allows standard DDE connections to be redirected across the network as was possible with Windows for Workgroups [SYBEX].

2.6 Impersonation

One of the most popular area to use Named Pipes has been database client-server communication. Until before Windows 2000, there was a serious problem with the security of this communication. The Windows 2000 operating system now provides special APIs that increase security for named pipes. Using a feature called impersonation, the server can change its security identity to that of the client at the other end of the message. This is done with the **ImpersonateNamedPipeClient()** API [PlatSDK2]. A server typically has more permissions to access databases on the server than a client requesting services. When the request is delivered to the server through a named pipe, the server changes its security identity to the security identity of the client. This limits the server to only those permissions granted to the client rather than its own permissions, thus increasing the security of named pipes.

3. TCP/IP Sockets over Windows 2000

3.1 Windows Sockets Background

The Windows Sockets specification defines a binary-compatible network programming interface for Microsoft Windows. Windows Sockets are based on the UNIX® sockets implementation in the Berkeley Software Distribution (BSD, release 4.3) from the University of California at Berkeley. The specification includes both BSD-style socket routines and extensions specific to Windows. Using Windows Sockets permits your application to communicate across any network that conforms to the Windows Sockets API. On Win32, Windows Sockets provide for thread safety.

Many network software vendors support Windows Sockets under network protocols including Transmission Control Protocol/Internet Protocol (TCP/IP), Xerox® Network System (XNS), Digital Equipment Corporation's DECNet™ protocol, Novell® Corporation's Internet Packet Exchange/Sequenced Packed Exchange (IPX/SPX), and others. Although the present Windows Sockets specification defines the sockets abstraction for TCP/IP, any network protocol can comply with Windows Sockets by supplying its own version of the dynamic link library (DLL) that implements Windows Sockets. Examples of commercial applications written with Windows Sockets include X Window servers, terminal emulators, and electronic mail systems [WinSock].

3.2 Definition of Socket

A socket is a communication endpoint — an object through which a Windows Sockets application sends or receives packets of data across a network. A socket has a type and is associated with a running process, and it may have a name. Currently, sockets generally exchange data only with other sockets in the same "communication domain," which uses the Internet Protocol Suite. Both kinds of sockets are bi-directional: they are data flows that can be communicated in both directions simultaneously (full-duplex).

Two socket types are available:

- **Stream sockets:** Stream sockets provide for a data flow without record boundaries — a stream of bytes. Streams are guaranteed to be delivered and to be correctly sequenced and unduplicated.
- **Datagram sockets:** Datagram sockets support a record-oriented data flow that is not guaranteed to be delivered and may not be sequenced as sent or unduplicated [WinSock].

3.3 Windows Sockets 2 and TCP/IP

One of the primary goals of Windows Sockets 2 has been to provide a protocol-independent interface fully capable of supporting emerging networking capabilities, such as real-time multimedia communications. Windows Sockets 2 is an interface, not a protocol. As an interface, it is used to discover and utilize the communications capabilities of any number of underlying transport protocols. Because it is not a protocol, it does not in any way affect the *bits on the wire*, and does not need to be utilized on both ends of a communications link.

Windows Sockets programming previously centered on TCP/IP. Some of the programming practices that worked with TCP/IP do not work with every protocol. As a result, the Windows Sockets 2 API added new functions where necessary to handle several protocols. Windows Sockets 2 has changed its architecture to provide easier access to multiple transport protocols. Consequently, Windows Sockets 2 support is not limited to TCP/IP protocol stacks as is the case for Windows Sockets 1.1.

There are new challenges in developing Windows Sockets 2 applications. When sockets only supported TCP/IP, a developer could create an application that supported only two socket types: connectionless and connection-oriented. Connectionless protocols used SOCK_DGRAM sockets and connection-oriented protocols used SOCK_STREAM sockets. Now, these are just two of the many new socket types. Additionally, developers can no longer rely on socket type to describe all the essential attributes of a transport protocol [PlatSDK3].

3.4 Uses for Sockets

Sockets are highly useful in at least three communications contexts:

- Client/Server models
- Peer-to-peer scenarios, such as chat applications
- Making remote procedure calls (RPC) by having the receiving application interpret a message as a function call

3.5 Named Pipes Vs TCP/IP Sockets

In a fast local area network (LAN) environment, Transmission Control Protocol/Internet Protocol (TCP/IP) Sockets and Named Pipes clients are comparable in terms of performance. However, the performance difference between the TCP/IP Sockets and Named Pipes clients becomes apparent with slower networks, such as across wide area networks (WANs) or dial-up networks. This is because of the different ways the interprocess communication (IPC) mechanisms communicate between peers.

For named pipes, network communications are typically more interactive. A peer does not send data until another peer asks for it using a read command. A network read typically involves a series of peek named pipes messages before it begins to read the data. These can be very costly in a slow network and cause excessive network traffic, which in turn affects other network clients.

It is also important to clarify if you are talking about local pipes or network pipes. If the server application is running locally on the computer running an instance of Microsoft® SQL Server™ 2000, the local Named Pipes protocol is an option. Local named pipes runs in kernel mode and is extremely fast.

For TCP/IP Sockets, data transmissions are more streamlined and have fewer overheads. Data transmissions can also take advantage of TCP/IP Sockets performance enhancement mechanisms such as windowing, delayed acknowledgements, and so on, which can be very beneficial in a slow network. Depending on the type of applications, such performance differences can be significant.

TCP/IP Sockets also support a backlog queue, which can provide a limited smoothing effect compared to named pipes that may lead to pipe busy errors when you are attempting to connect to SQL Server.

In general, sockets are preferred in a slow LAN, WAN, or dial-up network, whereas named pipes can be a better choice when network speed is not the issue, as it offers more functionality, ease of use, and configuration options.
For more information about TCP/IP, see the Microsoft Windows NT® documentation [TCPvsPIP].

4. Comparison between IPC over Unix and IPC over Windows 2000

4.1 UNIX IPC Background

The UNIX operating system developers define interprocess communication (IPC) as a means by which two or more running programs (processes) can communicate by exchanging data [LawBesaw]. The communication we are interested in is conducted by ordinary read and write calls, occasionally supplemented by I/O control requests, so that it resembles - and, where possible, is indistinguishable from - I/O to files. Moreover, we wish to commence communication in ways that resemble the opening of ordinary files.

4.1.1 A note on UNIX Orientation for IPC

The UNIX architecture provides a framework in which standard file, terminal, and communications I/O all operate in a similar fashion. Operations are performed on a file by means of calls such as:

```
descriptor = open(filename, readwritemode);
read(descriptor, buffer, length);
write(descriptor, buffer, length);
close(descriptor);
```

When a program opens a file, the call creates an area in memory called a *file control block* (FCB). Information about the file is stored in the FCB. The *open* call returns a small integer called a *file descriptor*. The program uses this descriptor to identify the file in any subsequent operations. As the user reads from or writes to the file, a pointer in the file descriptor keeps track of the current location in the file.

4.2 Types of UNIX IPC and Definitions

At first lets try and identify the types of IPC that is available over UNIX and define them very briefly. There are two types of IPC in UNIX: Local and Remote [EDavis]. These IPC mechanisms achieve the purpose of process to process communication either between processes on the same machine (local IPC) or on processes on separate machines (remote IPC). While the Windows 2000 model incorporates the use of Network protocols, file systems and redirectors to achieve this cross-machine data exchange, UNIX has a similar approach with some differences. We will take a look at the modules that UNIX uses to accomplish IPC.

4.2.1 Local IPC

This can exist between one or more processes on a single host. Examples of the mechanisms of Local IPC are: signals, pipes (unnamed, named, and STREAM pipes), shared memory, message queues, semaphores, UNIX-domain sockets, file/record locking, memory-mapped files. Signals are nothing but software interrupts; Pipes allow for half-duplex (uni-directional), reliable, FIFO, byte-stream IPC between parent and child(en) on same machine; Sockets allow for full duplex (bi-directional), reliable local and remote IPC between potentially unrelated processes; Shared Memory is implemented via the mmap system call and Synchronization requires file and record locking.

4.2.2 Remote IPC

This can exist between one or more processes on different hosts. Typically this involves some network protocols. There are different mechanisms for IPC over UNIX:

- Internet-domain sockets
- Transport Layer Interface (TLI)

Remote Procedure Calls (RPC)

4.2.3 Differences in IPC models in versions of UNIX

Below we compare the IPC model that exists between the BSD and System V [Klefstad] versions of Unix. Since there are a couple of differences between the 2 systems lets consider an intersection of the IPC mechanisms available on these two versions of UNIX for our discussions in comparing the IPC with Windows 2000.

BSD UNIX	System V
Signals	Signals
Pipes	Pipes
Sockets	TLI (Transport Layer Interface similar to BSD sockets)
Shared Memory	Shared Memory
Synchronization	FIFO (known as named pipes – also in BSD now)
	Stream Pipes
	Semaphores
	Message Queues

4.3 Streams in UNIX

Different mechanisms come into play to implement IPCs in UNIX. An example of one of them is a *stream,* which is a full-duplex connection between a process and a device or another process consisting of several linearly connected processing modules [LawBesaw]. A stream is analogous to a Shell pipeline, except that data flows in both directions. The end modules in a device stream become connected automatically when the process opens the device; streams between processes are created by a *pipe* call. Intermediate modules are attached dynamically by request of the user's program. They are addressed like a stack with its top close to the process, so installing one is called `pushing' a new module. Stream modules are part of the operating system kernel, but because they transmit messages, and streams can connect processes, it is plausible to transmit a stream through a user program.

4.3.1 Pipe Streams in UNIX

Pipes in Unix are commonly used to avoid writing temporary files to communicate between programs [BobStear]. The most common use of a pipe is probably:

```
ls -Fla | more
```

which takes the *stdout* from the *ls* command and transfers it to the *stdin* of the *more* command, allowing the reading of the file list one page at a time. If there were no pipe construct available and no file called *listing*, the process would be something like this:

```
ls -Fla > listing
more < listing
rm listing
```

4.3.2 Named Pipe Streams in UNIX

Named pipes (also known as FIFOs) are used to communicate between programs just like regular pipes are, but are usable by programs which use file names rather than just *stdin* and *stdout*. Furthermore, multiple named pipes may be open at the same time, rather than just the two normally allowed by shell commands [BobStear].

4.3.2.1 Using Named Pipes in UNIX

The following example shows how to create the compressed version of the merge of three sorted files which currently only exist as compressed files, without creating an uncompressed file on disk [BobStear].

First create the named pipes to use:

```
mkfifo in1 in2 in3
```

Then attach the uncompressed version of the compressed files to the pipes:

```
zcat jan.Z >> in1 &
zcat feb.Z >> in2 &
zcat mar.Z >> in3 &
```

Next, merge the uncompressed versions and compress the result, writing only the compressed version on the disk:

```
sort -m -1 +2 in1 in2 in3 \ compress > quarter1.Z
```

Finally, remove the pipes:

```
rm in1 in2 in3
```

There are several noteworthy things about the example above. First, *mkfifo* is just a front-end to the *mknod* command with a *p* parameter. Second, all of the writing processes must be startedin the background before the reading process is started in the foreground. (Compare the *zcat* commands with the *sort* command.) Third, after they have been created, FIFOs are treated just the same as regular files. (That is why the *zcat* command has to write to the end of the files via '>>' rather than the beginning via '>'.)

Also note that with the correct permissions, different users can be the source(s) and sink(s) for the data. This technique can be used anywhere you wish to create (uncompress, program, etc.) an input file without actually taking up disk space with it.

4.4 Sockets in UNIX

Berkeley's 4.2 BSD system10 introduced *sockets* (communication connection points) that exist in domains (naming spaces). The design is powerful enough to provide most of the needed facilities.

4.4.1 Uses of Sockets in UNIX

The use of a socket for communication often follows the client/server model. One method of communication between server and client processes is to design the server following these steps:

1. Create the socket.
2. Assign a name to the socket.
3. Attach a connection to the socket.
4. Transfer data via the socket.
5. Clean up the socket after use.

The connection to a socket also uses a socket. The connecting socket used for a client follows similar steps, except that the assignment of a name is not always necessary.

4.4.2 UNIX Socket Programming Interface

Most TCP/IP implementations offer a programming interface that follows a single model, the *socket programming interface*. The socket interface is a de facto standard because it is almost universally available and is in widespread use. The original socket interface was written for a Unix operating system. It was first introduced in 1982 with BSD Unix, and was designed for use with several communications protocols. AT&T introduced the *Transport Layer Interface* (TLI) for Unix System V. TLI is used to interface to the OSI transport layer, TCP, and other protocols.

4.5 Remote Procedure Calls in UNIX

The *Remote Procedure Call* (RPC) framework was designed from the start to support general client/server application development. It has evolved into the *Open Network Computing* (ONC) standard.

RPCs are implemented as a compiler feature supported by platform-specific RPC run-time libraries that allow the program to make remote-procedure calls in the same manner as a local procedure call would be made. The run-time library is responsible for "finding" the remote procedure, establishing the connection, and handling the communication.

The *eXternal Data Representation* (XDR) standard is an important part of the RPC architecture. XDR includes a datatype definition language and a method of encoding datatypes in a standard format. This enables data to be exchanged between different types of computers.

Sun Microsystems published a RFC describing Remote Procedure Calls in 1988. Sun maintained control of the protocol until 1995, when new versions were published. At that point, Sun's Open Network Computing Remote Procedure Call and its supporting protocols were turned over to the Internet Engineering Task Force (IETF) and submitted to the Internet standards process.

4.6 Signals in UNIX

Berkeley UNIX provides a set of signals that may be delivered to a process for various reasons, such as an interrupt key being typed or a bus error occurring. These signals--examples of which are SIGIO and SIGALRM--are defined in the file <signal.h>. Typically, the default action is that the delivery of the signal causes a process to terminate. This default can be changed so that a signal is caught or ignored. If the signal is caught, a signal handler is declared as the location where control is transferred at the time of interrupt. The arrival of a signal is thus similar to a hardware interrupt. When a signal is delivered to a process, the program state is saved, the delivered signal is blocked from further occurrence, and program control is transferred to the designated handler. If the handler returns normally, the signal is once again enabled and program execution resumes from the point where it was interrupted. If a signal that is currently blocked arrives, it is queued for later delivery.

5. References

[BobStear] B.Stearns, Unix named pipes (FIFOs), Computer Review, Winter Quarter 1995, http://www.uga.edu/~ucns/tti/Computer_Review/Winter95/UNIX.html

[Clark93] G.Clark, Networking in Microsoft Windows NT, Microsoft Corporate Technology Team, Technical Articles, MSDN Library, January 2001.

[EDavis] E.Davis, UNIX Network Programming, http://www.nas.nasa.gov/~edavis/unix_net_prog.html

[Klefstad] R.Klefstad, UNIX Network Programming, Introduction to UNIX Local and Remote Interprocess Communication, http://www.ics.uci.edu/~klefstad/s/147/lectures/unix-networking.txt

[LawBesaw] L.Besaw, Berkeley UNIX System Calls and Interprocess Communication, January 1987.

[NetArch] Named Pipes and Mailslots, Windows 2000 Server Resource Kit Online Books, MSDN Library, January 2001.

[NTUnix] MS Windows NT Server From a UNIX Point of view, Microsoft TechNet, http://www.microsoft.com/technet/winnt/Winntas/technote/ntunixvw.asp

[PlatSDK1] Named Pipes, Interprocess Communications: Platform SDK, MSDN Library, January 2001.

[PlatSDK2] Named Pipe Security, Interprocess Communitions: Platform SDK, MSDN Library, January 2001.

[PlatSDK3] Overview of Windows Sockets 2: Platform SDK, MSDN Library, January 2001.

[SYBEX] Creating Named Pipes, Partial Books, MSDN Library, January 2001.

[SoloRuss] Inside Microsoft Windows 2000, 3rd edition, David A. Solomon and Mark E. Russinovich, Microsoft Press.

[TCPvsPIP] Named Pipes vs. TCP/IP Sockets, Optimizing Database Performance (SQL Server), MSDN Library, January 2001.

[WinSock] Windows Sockets: Background, MSDN Library, January 2001.