Interprocess Communication In

UNIX and Windows NT

Scott M. Lewandowski
Department of Computer Science
Brown University
Providence, RI 02912-1910
scl@cs.brown.edu

TABLE OF CONTENTS

1. Introduction

Interprocess communication (IPC) refers to the coordination of activities among cooperating processes. A common example of this need is managing access to a given system resource. To carry out IPC, some form of active or passive communication is required.

The increasingly important role that distributed systems play in modern computing environments exacerbates the need for IPC. Systems for managing communication and synchronization between cooperating processes are essential to many modern software systems. IPC has always played a prominent role in UNIX-variant operating systems, but it has been largely overlooked for systems running the Windows NT operating system. This paper will discuss some of the IPC options that are available to programmers using UNIX and describe the corresponding techniques available to programmers writing for Windows NT. Features and mechanisms specific to Windows NT will also be discussed. The conclusion will offer a summary of the available techniques as applicable to Windows NT.

2. Pipes

Pipes are a simple synchronized way of passing information between two processes. A pipe can be viewed as a special file that can store only a limited amount of data and uses a FIFO access scheme to retrieve data. In a logical view of a pipe, data is written to one end and read from the other. The processes on the ends of a pipe have no easy way to identify what process is on the other end of the pipe.

The system provides synchronization between the reading and writing process. It also solves the producer/consumer problem: writing to a full pipe automatically blocks, as does reading from an empty pipe. The system also assures that there are processes on both ends of the pipe at all time. The programmer is still responsible, however, for preventing deadlock between processes.

Pipes come in two varieties:

- **Unnamed.** Unnamed pipes can only be used by related processes (i.e. a process and one of its child processes, or two of its children). Unnamed pipes cease to exist after the processes are done using them.
- **Named.** Named pipes exist as directory entries, complete with permissions. This means that they are persistent and that unrelated processes can use them.

2.1 UNIX

Most UNIX systems limit pipes to 5120K (typically ten 512K chunks). The unbuffered system call write() is used to add data to a pipe. Write() takes a file descriptor (which can refer to the pipe), a buffer containing the data to be written, and the size of the buffer as parameters. The system assures that no interleaving will occur between writes, even if the pipeline fills temporarily. To get data from a pipe, the read() system call is used. Read() functions on pipes much the same as it functions on files. However, seeking is not supported and it will block until there is data to be read.

The pipe() system call is used to create unnamed pipes in UNIX. This call returns two pipes. Both support bidirectional communication (two pipes are returned for historical reasons: at one time pipes were unidirectional so two pipes were needed for bidirectional communication). In a full-duplex environment (i.e. one that supports bidirectional pipes) each process reads from one pipe

and writes to the other; in a half-duplex (i.e. unidirectional) setting, the first file descriptor is always used for reading and the second for writing.

Pipes are commonly used on the UNIX command line to send the output of one process to another process as input. When a pipe is used both processes run concurrently and there is no guarantee as to the sequence in which each process will be allowed to run. However, since the system manages the producer/consumer issue, both proceed per usual, and the system provides automatic blocking as required.

Using unnamed pipes in a UNIX environment normally involves several steps:

- Create the pipe(s) needed
- Generate the child process(es)
- Close/duplicate the file descriptors to associate the ends of the pipe
- Close the unused end(s) of the pipe(s)
- Perform the communication
- Close the remaining file descriptors
- Wait for the child process to terminate

To simplify this process, UNIX provides two system calls that handle this procedure. The call popen() returns a pointer to a file after accepting a shell command to be executed as input. Also given as input is a type flag that determines how the returned file descriptor will be used. The popen() call automatically generates a child process, which exec()s a shell and runs the indicated command. Depending on the flag passed in, this command could have either read or write access to the file. The pclose() call is used to close the data stream opened with popen(). It takes the file descriptor returned by popen() as its only parameter.

Named pipes can be created on the UNIX command line using mknod, but it is more interesting to look at how they can be used programatically. The mknod() system call, usable only by the superuser, takes a path, access permissions, and a device (typically unused) as parameters and creates a pipe referred to by the user-specified path. Often, mkfifo() will be provided as an additional call that can be used by all users but is only capable of making FIFO pipes.

2.2 Windows NT

Windows NT supports both named and unnamed pipes, although it refers to the latter as anonymous pipes.

The CreatePipe() function creates an anonymous pipe and returns two handles. One handle is a read handle to the pipe and the other is a write handle to the pipe; neither pipe can perform the opposite operation. When the pipe is created the server requests that it be implemented using a programmer-defined buffer size. To communicate using the pipe, the server must pass one of the handles to another process. Usually, this is done through inheritance; that is, the process allows the handle to be inherited by a child process. The process can also send the handle to an unrelated process using another form of interprocess communication, such as shared memory.

A server can send either the read handle or the write handle to the pipe client, depending on whether the client should use the anonymous pipe to send information or receive information. To read from the pipe, the pipe's read handle is used as a parameter to the call ReadFile(). The ReadFile() call returns when another process has written to the pipe. ReadFile() call can also return if all write handles to the pipe have been closed or if an error occurs before the read operation has been completed.

To write to the pipe, the pipe's write handle is passed as a parameter to the WriteFile() function. The WriteFile() call does not return until it has written the specified number of bytes to the pipe or an error occurs. If the pipe buffer is full and there are more bytes to be written, WriteFile() does not return until another process reads from the pipe, which frees up buffer space.

Asynchronous read and write operations are not supported by anonymous pipes. An anonymous pipe exists until all pipe handles, both read and write, have been closed. A process can close its pipe handles by using the CloseHandle() function. All pipe handles are also closed when the process terminates.

Named pipes in Windows NT have a unique name that distinguishes them from other named pipes in the system's list of named objects. A server specifies a name for a pipe when it creates it by calling CreateNamedPipe(). Clients specify the pipe name when they call CreateFile() to connect to an instance of the named pipe.

Like filenames in Windows NT, the form \\ServerName\pipe\PipeName should be used when specifying the name of a pipe to any of the pipe functions. Here, the server could be a remote computer or the local host, which can be specified using a period. Pipes cannot be created on remote machines, but they can be used on remote machines once created.

The first time a pipe server calls CreateNamedPipe(), it specifies the maximum number of instances of the pipe that can exist simultaneously. The server can call CreateNamedPipe() repeatedly to create additional instances of the pipe, as long as it does not exceed the maximum number of instances. If the function succeeds, each call returns a handle to the server end of a named pipe instance.

As soon as the pipe server creates a pipe instance, a pipe client can connect to it by calling CreateFile() or CallNamedPipe(). If a pipe instance is available, CreateFile() returns a handle to the client end of the pipe instance. If no instances of the pipe are available, a pipe client can use the WaitNamedPipe() function to wait until a pipe becomes available. A server can determine when a client is connected to a pipe instance by calling ConnectNamedPipe(). If the pipe handle is in blocking-wait mode, ConnectNamedPipe() does not return until a client is connected.

When a client and server finish using a pipe instance, the server should first call FlushFileBuffers(), to ensure that all bytes or messages written to the pipe are read by the client. FlushFileBuffers() does not return until the client has read all data from the pipe. The server then calls DisconnectNamedPipe() to close the connection to the pipe client. This function makes the client's handle invalid, if it has not already been closed. Any unread data in the pipe is discarded. After the client is disconnected, the server calls the CloseHandle() function to close its handle to the pipe instance. Alternatively, the server can use ConnectNamedPipe() to enable a new client to connect to this instance of the pipe.

Several functions unique to Windows NT are available for working with pipes. The call PeekNamedPipe() can be used to read from a pipe without removing data from it. The TransactNamedPipe() function writes a request message and reads a reply message in a single operation, enhancing network performance. It can be used with duplex pipes if the pipe handle of the calling process is set to read mode. A process can retrieve information about a named pipe by calling GetNamedPipeInfo(), which returns the type of the pipe, the size of the input and output buffers, and the maximum number of pipe instances that can be created. GetNamedPipeHandleState() reports on the read and wait modes of a pipe handle, the current number of pipe instances, and additional information for pipes that communicate over a network. SetNamedPipeHandleState() sets the read mode and wait modes of a pipe handle. For pipe clients communicating with a remote server, the function also controls the maximum number of bytes to collect or the maximum time to wait before transmitting a message (assuming the client's handle was not opened with write-through mode enabled).

3. Sockets

A socket is a software abstraction that is used to create a channel-like interface between processes. The channel allows bi-directional communication between processes, but does little to structure the data being transmitted. In general, the serving machine in a system that uses socket communication follows several steps to initiate communication:

- Create a socket
- Map the socket to a local address
- Listen for client requests

When the client process wishes to begin a transaction with the server, it follows similar steps:

- Create a socket
- Determine the location (system name and port number) of the server
- Begin sending and/or receiving data

3.1 Types of Sockets

Communication between processes via sockets can be viewed as streams or datagrams. Stream-based communication views transmitted data as a sequence of bytes, whereas datagram-based communication deals with small discrete packets of information. These packets typically contain header and trailer information in addition to the data being transmitted. Choosing a socket type is important, because each has different performance and reliability characteristics, and two communicating sockets must be of the same type.

- **Stream.** Stream sockets are reliable; that is, data is guaranteed to be delivered in the same order it was sent. An underlying mechanism that the programmer need not be concerned with handles duplicate data, performs error checking, and controls flow. When using stream sockets, a connection is established. This means that two processes must logically agree to communicate before information can be transferred between the two of them. This connection is maintained by both processes throughout the communication session.
- **Datagram.** Datagram sockets provide unreliable communication between the two processes, meaning that data could be received out of order. Datagram sockets provide no notion of a connection, so each datagram is sent and processed independently. This has a host of implications, one of the most significant of which is that individual datagrams can take different paths to their ultimate destination. Another important implication is that there is no flow control. Datagram packets are typically quite small and usually are fixed in size. No error correction is mandated, and when optional error connection is used, it is weak.

3.2 Berkeley Sockets

In the early 1980s, the Berkeley socket interface was introduced and it has formed the basis for UNIX sockets programming ever since. Berkeley sockets allow general file system type access routines to be used to send and receive information over a network connection. Berkeley sockets are completely protocol independent, so they need not use TCP/IP for communication although this is most common.

Berkeley sockets support asynchronous I/O, meaning that the process can request that the kernel alert it when a specified socket descriptor is ready for I/O. In most UNIX implementations, the notification from the kernel to the user process occurs via the SIGIO signal. Because a given process can have only a single signal handler for each signal, to use asynchronous I/O with more than one socket, another technique is used. Otherwise, the process would not know which socket is ready for I/O. Such multiplexing is implemented via the select() system call.

Using multiplexed I/O solves the problem of delaying data from non-active sockets while an active socket is blocking waiting for an I/O transaction to take place. There are several potential solutions to this problem in UNIX:

- **Polling.** If all sockets are set to be nonblocking, the process can execute a loop that checks each socket to see if there is something to be read. If there is, it handles it; otherwise, it goes to sleep until the next time that the socket is scheduled to be checked. Polling is inefficient because most of the time there is no work to be done, yet CPU time is wasted.
- **Use multiple processes.** The parent process could use fork() to establish a child process to handle each socket. With this technique, each process can read from the port and block since the system will handle scheduling the processes most efficiently. However, the child process must then return the read data to the parent process via some other form of IPC.
- **Asynchronous I/O.** This is not a good solution in UNIX since signals are very expensive to catch. In addition, if more than one socket is using asynchronous I/O, there is a need to determine which socket the signal corresponds to.
- **Use select().** Using select() allows a user process to request that the kernel wait for one of several events occurs and to wake the process up at that time. The select() call is quite flexible: asking it to return immediately effectively is a poll request, and timeouts can optionally be specified.

The use of select() is the preferred technique when using Berkeley sockets because it is fairly efficient and simple to work with.

3.3 Windows Sockets: WinSock v2.2

The Microsoft implementation of Windows sockets is derived from Berkeley sockets as included in BSD v4.3. Like Berkeley sockets, Windows sockets are protocol independent and support asynchronous operation. The most recent release of Windows sockets, which is embodied by the WinSock v2.2 API, includes native support for the following protocols:

- IPX/SPX: sockets serve as an alternative to Novell's Event Control Block (ECB) architecture, which forces a programmer to learn the details of the protocol.
- NetBEUI: replaces the NetBIOS interface
- AppleTalk: to allow communication with Macintosh computers
- ISO TP/4: widely used in Europe, so is important for internationalization efforts

The availability of all of these protocols via sockets simplifies application programming since using a different protocol for communication does not necessitate learning its idiosyncrasies or learning a new API. Although Windows sockets give a programmer protocol independence, protocol transparency is not provided since setting up a socket connection requires specifying the protocol to be used, among other parameters. Later an alternative technique for providing protocol transparency using Win32 will be discussed.

3.4 Comparing Windows and Berkeley Sockets

3.4.1 Porting Code from Berkeley Sockets to Windows Sockets

Windows sockets were designed to facilitate easy porting of Berkeley socket code to Windows sockets. Windows sockets support the Berkeley API (with minor exceptions) as well as functionality specific to Win32.[1] Code written using Berkeley sockets can usually be ported to Windows sockets with only small changes. One such change is the need to call WSAStartup(), which accepts as an argument the version of the sockets DLL an application requires; this provides

[1] Such nonportable functions have the prefix of "WSA" in their names.

simple versioning, which is quite essential since the WinSock specification changes frequently. As with Berkeley sockets, socket() opens a new socket. In Windows, socket() returns an unsigned int, which behaves like a HANDLE. It is important to note that the int is unsigned; Berkeley sockets are represented by plain ints. This means that it is not correct to check for success by checking for non-negativity of the socket returned. Alteratively, WSASocket() can be used; this allows the programmer additional flexibility, such as being able to select between overlapped and non-overlapped sockets (socket() only creates non-overlapped sockets as of WinSock v2.2). The bind() call works basically the same under Berkeley and Windows sockets, as does name resolution. Windows NT provides %SYSTEM32%\drivers\etc\lmhosts, which is equivalent to /etc/hosts in UNIX. Other calls that function nearly identically include listen(), accept(), and connect().

After a socket connection is established, data needs to be transferred. Using the standard read() and write() calls from Berkeley sockets is the most straightforward to accomplish this communication using Windows sockets. The programer need to be aware of the fact that TCP/IP as implemented on Windows 95 and Windows NT will fragment packets, meaning that if 1024 bytes of data are to be received, one cannot just call recv() and ask for 1024 bytes; instead data should be “accumulated”. According to the WinSock documentation, send() also has this restriction, although this behavior rarely, if ever, manifests itself when using send().

Some Berkeley socket applications use the _read() and _write() calls since UNIX treats sockets as files. Windows sockets support these functions, but some additional work on the part of the programmer is required. The socket must be opened as non-overlapped and must be converted to a file handle by calling _open_osfhandle(). This will return a file handle (i.e. an int) which can then be used with _read() and _write().

When the program is through with the socket, it should call the nonportable function closesocket() on it, as opposed to the Berkeley socket standard close(). When the entire transaction is completed, WSACleanup() should be called to shut down the DLL. WinSock keeps a reference count for each socket being used so that when the count drops to zero it can be freed so the programmer need not worry about this detail.

So far, the discussion of sockets has focused on applications that rely on blocking sockets. Sockets can also be put into a nonblocking mode using ioctl() on UNIX or WSAIoctl() under WinSock v2.2. WSAIoctl() supports the use of overlapped execution for more time-intensive requests. When using Berkeley sockets, traditionally, one uses select() to detect the occurrence of an event. Since select() blocks until an event occurs, it is useful for determining when to call a function to act on received data. Although select() works using Windows sockets, it is not recommended because it is cumbersome to use, due largely to the fact that it deals with an array of sockets and not an array of strings like the Berkeley implementation does. More importantly, it is very inefficient, since it triggers context switches. Therefore, several alternatives are available to programmers using WinSock v2.2:

- **Use blocking overlapped sockets**, and specify a timeout.
- **Set send() and recv() timeouts** using setsockopt() on top of overlapped sockets. If this option is used, send() and recv() will block only for the user-specified period of time, not indefinitely.
- **WSAAsyncSelect()** can be used to put a socket into a nonblocking mode and request notification of events that occur on it.
- **Request asynchronous notification** by using WSAEventSelect(), which also makes the socket nonblocking. This differs from using WSAAsyncSelect() since it does not post a message to a window; it instead causes a Win32 object to signal.

3.4.2 Features Specific to Windows Sockets

There are two levels of extensions to Berkeley sockets that are included in Windows sockets. The first are the WinSock v1 extensions, which include several WSA functions that offer message posting and asynchronous execution. The second level of extensions is the WinSock v2 extensions, which add numerous enhancements, many of which address overlapped I/O.

3.4.2.1 Overlapped I/O

Windows sockets natively support overlapped I/O to provide superior data transfer capabilities. In the WinSock v1.1 standard, overlapped I/O was optional and limited to use under Windows NT. WinSock v2.2 supports overlapped I/O as the default under both Windows NT and Windows 95.[2] Since files are opened as Win32 overlapped I/O handles, you can pass sockets to ReadFile() or ReadFileEx() as well as the corresponding write functions without any modifications. If overlapped I/O is used, the programmer must use event handles, I/O completion routines, or I/O completion ports. However, this overhead is a small price to pay for the superior performance that can be achieved, especially on the server-side of a connection. Tests indicate that performance using overlapped I/O consistently is approximately 100K/sec better than with other techniques.

WSAAsyncSelect() is used to put a socket into nonblocking mode and to post messages to a window when events occur. This is an important feature because it features a great deal more functionality and automation than select(). There are many events that can be checked for; some of the most common are shown below.

lEvent Value	Meaning
FD_READ	Data is available for reading
FD_WRITE	The socket can be written to
FD_OOB	Urgent data needs to be read
FD_ACCEPT	A client request for a connection has arrived
FD_CONNECT	A client's connect attempt has completed
FD_CLOSE	The partner station has closed the socket
FD_QOS	The socket's quality of service has been changed

WSAAsyncSelect() does not need to be reactivated after each event is handled by the appropriate handling function. Detailed error messages are returned via the standard Win32 interface. Only one call to WSAAsyncSelect() can be active at any given time. Subsequent calls override previous calls, so if multiple events are to be handled their values can be ORed together. A typical sequence of events in a system using WSAAsyncSelect() is shown below, for both client and server sides of the connection.

Server Side	Client Side
Create a sockets and bind the your address into it	Create a socket
Call WSAAsyncSelect() and request FD_ACCEPT notification	Call WSAAsyncSelect() and request FD_CONNECT notification
Call listen() and go to other tasks	Call connect() or WSAConnect(); they will return right away so you can go on to doing something else
When the window is notified that a message came in, accept it using accept() or WSAAccept()	When you receive the FD_CONNECT notification, request FD_WRITE/READ/OOB/CLOSE notification

[2] The implementation of overlapped I/O in Windows 95 does not support I/O completion ports.

Request FD_READ/OOB/CLOSE socket so you know when the client sends data or closes the connection	When FD_WRITE is reported, the socket is available for the client to send requests to the server
When you receive FD_READ/OOB, call ReadFile(), read(), recv(), recvfrom(), WSARecv(), WSARecvFrom() to get the data	When the data comes from the server, the window received FD_READ/OOB, and you can respond by calling ReadFile(), read(), recv(), recvfrom(), WSARecv(), or WSARecvFrom().
Respond to FD_CLOSE by calling closesocket()	The client should normally close the connection, but should be ready to handle an unexpected FD_CLOSE from the server

WSAAsyncSelect() is a very powerful function that adds a great deal of functionality to a program and allows the programmer to work at a higher level of abstraction. Windows servers using WSAAsyncSelect() do not need to be multithreaded for good performance, since using the function gives implicit multithreading. Certainly being able to write virtually multithreaded code without worrying about deadlock, synchronization, and the like is a welcome benefit. When using WSAAsyncSelect() it is also not necessary to allocate buffer space before an asynchronous receive request is posted since the system buffers the data. This also prevents excessive allocation of memory since when memory does need to be allocated the precise quantity is known.

An addition to WinSock v2 is the WSAEventSelect() call. It is quite similar to WSAAsyncSelect() but it uses signalling instead of message posting. WSAEventSelect()accepts a handle to an event as a parameter. The call immediately returns, and then the program should wait on the event handle using a standard Win32 mechanism such as WaitForSingleObject(). WinSock v2 does provide a special function, WSAWaitForMultipleEvents(), which allows a timeout to be specified and allows the programmer to select from waiting until all events have signalled or just for any one event to signal. If the programmer wishes to approximate the Berkeley sockets select() function, but in a more efficient way than using the Windows sockets select() call, s/he could use WSAEnumNetworkEvents(), which fills an array with the FD_* codes of completed operations. It is very important to note that WSAAsyncSelect() and WSAEventSelect() are mutually exclusive; that is, using one cancels the other. As with WSAAsyncSelect(), the solution is to OR FD_* codes together.

3.4.2.2 Protocol Transparency

Since a single binary program image does not work with all protocols but protocol transparency is a desirable feature, Microsoft included he Service Registration API starting with Windows NT v3.5. Using the Service Registration API, servers using Windows sockets can register their presence on the network with a centralized information brokerage. In addition, they can ask what parameters the underlying support layers should be passed when using socket functions, allowing code to be written with no assumptions about the networking environment in which the program will ultimately run. WinSock v2.2 now provides some of this functionality to the programmer transparently.

Using the Service Registration API, server class information can be registered. A server name can be associated with the registered class identifier. When the server is started, it can use this information to determine what sockets should be opened and to what ports and machine addresses they should be bound. The API also allows the server to advertise its presence on the network to facilitate discovery and subsequent connection by clients. Unfortunately, using this useful facility requires complex variable-sized data structures to be populated—not an easy chore. Taking a look at the specifics of the operation of the Service Registration API is quite interesting. The basic operations it supports are:

- **Registration of services and classes.** A class or service is registered by associating it with a globally unique identifier (GUID); the GUIDs are the same ones used to identify COM

components and RPC interfaces. When this registration is performed, the communication protocols to be supported can be specified

- **Helping the server application read the registration information.** When the server first comes up, it needs to read the stored registration information. With this information it knows what to pass to the socket() or WSASocket() calls. After setting up its sockets and entering the listening state, the server can tell the registration database that it is online.
- **Helping the client application read the registration information.** When the client starts up, it too needs to retrieve registration information using the Service Registration API. With this information it can find a target server to connect to, and the client knows what information to pass to the relevant function calls.

3.4.2.3 Socket Abstractions

Microsoft Foundation Classes (MFC) are widely used for Windows software development today. It is no surprise that there are classes designed specifically to accommodate the use of sockets. There are two classes offered by MFC for sockets:

- **CAsyncSocket.** This class is a thin wrapper around the WinSock v2.2 API. It provides asynchronous communication that relies on WSAAsyncSelect() heavily. All socket event notifications are handled internally by a CSocketWnd instance so the programmer does not need to deal with them. The events to be monitored are indicated using AsyncSelect().
- **CSocket.** The CSocket class is very similar to CAsyncSocket but it provides the illusion of synchronous communication. In other words, the functions block and do not return until the operation completes, although in reality asynchronous communication is still used.

These classes are significant to MFC programmers because in addition to encapsulating much of the busywork of using sockets into the socket classes, they work with MFC features such as object serialization quite transparently. For example, to send a CArchive-derived object over a socket connection one can use the << operator.

3.4.2.4 Quality of Service

WinSock v2.2 also offers quality of service (QOS) capabilities. The basic QOS mechanism descends from a RFC flow specification. Each flow specification describes a set of characteristics about a proposed unidirectional flow through the network. An application may associate a pair of flow specifications with a socket (one for each direction) at the time a connection request is made using WSAConnect(), or at other times using WSAIoctl(). Flow specifications indicate what level of service is required for the application at hand and provide a feedback mechanism for applications to use in adapting to network conditions.

An application may establish its QOS requirements at any time via WSAIoctl() or when the connection is started using WSAConnect(). For connection-oriented sockets, it is often most convenient for an application to use WSAConnect(); QOS values supplied at connect time supersede those that may have been supplied earlier via WSAIoctl(). If the call to WSAConnect() completes successfully, the application knows that its QOS request has been honored by the network, and the application is then free to use the socket for data exchange. If the connect operation fails because of limited resources an appropriate error is returned and the application may scale down its service request and try again or determine that network conditions are not acceptable and exit.

After every connection attempt, transport providers update the associated flow specification structures to indicate the most recent network conditions. This update from the service provider about current network conditions is especially useful when the application's QOS request consisted of the default values, which any service provider should be able to agree to (meaning

that application had no real information about the network). Applications expect to be able to use this information to guide their use of the network, including any subsequent QOS requests. Information provided by the transport in the updated flow specification structure may be little more than a rough estimate that only applies to the first hop as opposed to the complete end-to-end connection, so the application should take appropriate precautions when interpreting this information.

Connectionless sockets may use WSAConnect() to establish a specified QOS level to a single designated peer. Otherwise connectionless sockets make use of WSAIoctl() to stipulate the initial QOS request, and any subsequent QOS renegotiations.

The flow specifications for WinSock v2.2 divide QOS characteristics into the following areas:

- **Source** Traffic Description. The manner in which the application's traffic will be injected into the network. This includes specifications for the token rate, the token bucket size, and the peak bandwidth. Even though the bandwidth requirement is expressed in terms of a token rate, this does not mean that service provider must actually implement token buckets; any traffic management scheme that yields equivalent behavior is permitted.
- **Latency.** Upper limits on the amount of delay and delay variation that are acceptable.
- **Level of service guarantee.** Whether or not an absolute guarantee is required as opposed to best effort. Providers which have no feasible way to provide the level of service requested are expected to fail the connection attempt.
- **Cost.** This is for the future when a meaningful cost metric can be determined.
- **Provider-specific parameters.** The flow specification can be extended in ways that are particular to specific providers.

3.4.2.5 Other Useful Extensions

Another useful extension to Berkeley sockets is the TransmitFile() call. It can be used to easily send an open file over a socket connection. The call allows all or part of a file to be sent over the network, maintains state using the standard notion of a seek pointer, and allows optimization of the transmission by allowing the programmer to override the default transmission packet size. A very useful feature of this call is that it takes as a parameter a struct that can hold header and trailer information. This is useful if, for example, the name that the file should assume on the receiving end of the connection should be transmitted, or it would be helpful to indicate that this file was the last in a series of transfers.

There are numerous WSAAsyncXXX() functions provided which offer superior performance and integration in an environment in which asynchronous communication is being used. These functions take advantage of capabilities provided via Win32 that allow communication and non-communication functions alike to more efficiently operate in multithreaded environments than their standard counterparts as implemented in UNIX do. Asynchronous socket programming is considerably more difficult and error prone than synchronous programming is, but large performance gains can be reaped, and often the benefits outweigh the added development effort.

3.4.2.6 Problems

Despite all of these advantages that Windows sockets have over Berkeley sockets, they are not perfect. Although CAsyncSocket is fairly efficient and imposes only a marginal performance penalty, using all of its asynchronous capabilities requires some challenging programming. On the other hand, CSocket is very simple to use, but this simplicity comes at the cost of high performance losses. A significant problem with these classes is that they do not support overlapped I/O, meaning that they cannot be used with I/O completion routines or I/O completion

ports. Finally, these classes still rely on WinSock v1.1; they do not take advantage of the new features or enhanced efficiency of WinSock v2.2 yet.

4. Shared Memory

Shared memory allows multiple processes to share virtual memory space so changes made by one process are instantly reflected in other processes. This affords the fastest possible IPC but is not necessarily the easiest to coordinate between the two processes. Shared memory is unique in that it allows random access of data, whereas most other IPC mechanisms mandate sequential access to data. Typically, one process creates a shared segment, and needs to set access permissions for the segment. It is then mapped into the process's address space after the process attaches to it. Usually the creating process initializes the memory, but after this other processes that have been given access permission to the segment can use it freely. When another process does use it, it is mapped into its address space. Oftentimes semaphores are used by the original process to be sure that only one other process is allowed to access the memory at any one time, although a readers/writers lock may be a more suitable solution. When no process needs the memory segment anymore, the creating process can delete it.

4.1 Creating a Shared Memory Segment

When programming for UNIX, a new shared memory segment can be created using the shmget() call. Using shmget() also generates the required data structures. If a shared memory segment was created by another process previously, using shmget() allows a process to gain access to the existing segment since shared memory segments are identified by unique identifiers returned by the call. When creating a new segment, the programmer can specify a key to associate with an existing shared memory segment, the size of the segment requested, and the creation and access conditions. Note that creating a shared segment only reserves space for it; the process still cannot access it.

Using a shared memory region under Windows NT requires two distinct steps, much as it does in UNIX: creating a kernel object specifying the size of the shared area and then mapping this file-mapping object into the process's address space.

To carry out the first step the CreateFileMapping() call is used. This call usually allows a file to be accessed as if in memory (not unlike the UNIX mmap() call) but can be used to create a place in one of the system paging files. This place, once reserved, can be accessed by name via the kernel. When using CreateFileMapping in this way, the file handle passed in is a special constant representing that a paging file should be used. Other parameters that CreateFileMapping() takes include flags for security attributes, access specifiers (which are normally set to allow page read and write access when using shared memory), the size of the segment to be mapped, and name that the segment can be referred to as. If the call returns successfully, a handle to a file-mapping kernel object is available to the calling process. It is not necessary to use the paging file for the mapping; however, doing so prevents processes from needing to agree on using a common file name and eliminates the possibility of having temporary files to cleanup. The choice of whether or not to use a pagefile or some other file effectively allows Windows NT to mimic both types of mapping possible under UNIX.[3] However, UNIX simplifies the case of general (not file mapped) shared memory by completely removing the burden of dealing with files.

[3] Windows NT requires the decision to be made earlier in the process, however; the second type of mapping has not yet been presented.

4.2 Controlling a Shared Memory Segment

Once a UNIX process has created a shared memory segment, it can control it using the shmctl() system call. This call can potentially affect existing shared memory segments as well as the system-maitained shared memory control structures. The call takes a key representing a previously created segment, a command (specified via a constant), and a buffer containing a struct that effectively serves as the parameters for the specified command. Possible commands include:

- IPC_STAT: returns the values of the associated data structure to a process having read access to the segment
- IPC_SET: allows a process with an effective UID of superuser or the UID of the process used to create the segment to modify the segment's associated user and group identifiers, as well as change the access permissions
- IPC_RMID: remove the system data structure, effectively indicating that the process is done with the shared segment and that if no other process is still using it then it can be deleted
- SHM_LOCK: allows the superuser to lock a shared segment in memory
- SHM_UNLOCK: allows the superuser to unlock a shared segment locked into memory

Windows NT does not offer similar features to user-level processes. This higher degree of management by the kernel is useful at times but does place some limits on the flexibility of shared memory for Windows NT.

4.3 Shared Memory Operations

UNIX features two operations to work with shared memory: shmat() and shmdt(). These allow processes to attach and detach from shared memory segments, respectively.

When a process attached to a shared memory segment using shmat(), the referenced shared memory is mapped into the calling process's data segment. The call takes an identifier of an existing shared memory segment, a flag that specifies the access permissions for the shared segment and to request special conditions (such as address alignment), and a suggestion for placement of the segment. This last argument is passed in as a void*, which is usually set to null, indicating that the system should choose the addresses that it feels best. However, the process could pass in a suggested address. The memory is then mapped into the nearest available page address.

When a process wishes to detach from a segment, it can use the shmdt() system call. The call takes only a reference to an attached memory segment.

When programming for UNIX, programmers can use the mmap() system call to map a file into a process's virtual address space. Mapping files has several advantages over using traditional shared memory segments. In particular, standard system calls can be used to manipulate the memory as desired (instead of the specialized calls that shared memory segments require) and files are persistent. The mmap() call takes several parameters. It accepts a user-specified address to try to map the file into, much as the shmat() call does. In addition, the length of the mapped memory area can be specified along with the access privileges that should apply. Various flags can be used to set attributes for the mapping, such as whether the map should be shared or private, whether the address needs to be fixed, and whether or not swap space should be reserved for the mapping. Although the file will be unmapped when the process terminates, the process can unmap it at any time using munmap(), which accepts the address and size of the memory block as parameters.

Windows NT provides equivalent functionality through the use of the OpenFileMapping() and UnmapViewOfFile() system calls.

The OpenFileMapping() call returns a handle to a file-mapping kernel object that was previously created using CreateFileMapping(). As parameters it takes the level of access that is desired for the segment, whether or not child processes should inherit the handle, and the name of the memory region to open (as specified when CreateFileMapping() was called). After the handle to the kernel object is acquired, a pointer to the memory that it represents is required. The MapViewOfFile() call can be used to get this pointer. MapViewOfFile() takes the handle returned by CreateFileMapping(), the desired level of access to the memory, and the size of the region to map as parameters. It returns a pointer to the beginning of the memory segment.

When the shared map is no longer required, it should be released using UnmapViewOfFile(). It takes the address of the memory (as returned by MapViewOfFile()) as a parameter. The programmer should also be careful to close the handle to the file-mapping kernel object.

5. Mailslots

Mailslots, available only in Windows NT, provide one-way communication between processes. A mailslot is a pseudofile; it resides in memory, and standard file functions are used to access it. The data in a mailslot message can be in any form. When all handles to a mailslot are closed, the mailslot and all the data it contains are deleted, so mailslots cannot be used for long-term storage.

A mailslot server is a process that creates and owns a mailslot. When the server creates a mailslot, it receives a mailslot handle. This handle must be used when a process reads messages from the mailslot. Only the process that creates a mailslot or obtains a handle to can read from the mailslot. All mailslots are local to the process that creates them; a process cannot create a remote mailslot. A mailslot client is a process that writes a message to a mailslot. Any process that has the name of a mailslot can put a message there. New messages follow any existing messages in the mailslot.

Processes that create mailslots act as servers to other processes that send messages to it by writing a message to its mailslot. Incoming messages are always appended to the mailslot and are saved until the mailslot server has read them. A process can be both a mailslot server and a mailslot client, so two-way communication is possible using multiple mailslots. This is useful for implementing a simple message-passing facility within a domain

Mailslot clients can send a message to any number of mailslot servers located anywhere on the network with a single call, as long as the mailslots all share the same name. The message can also be restricted to the local machine or a specific machine on the network. Messages broadcast to all mailslots on a domain can be no longer than 400 bytes, whereas messages sent to a single mailslot are limited only by the maximum message size specified by the mailslot server when it created the mailslot.

6. Conclusions

6.1 Review

Both UNIX and Windows NT provide efficient pipe implementations that are excellent in their own unique ways. The UNIX implementation is very simple to use whereas the Windows NT implementation is quite complex (in comparison), but affords the programmer substantially more flexibility. The abundance of information available at runtime allows powerful servers handling multiple clients to be created. The possibility for pipes to alternate between being synchronous and asynchronous is also a nice feature. Windows NT also natively supports multiple pipes, does not restrict the use of unnamed pipes to related processes, and allows bi-directional communication using a single pipe. Under both operating systems, pipes provide a reliable means for IPC, and the two parties can verify receipt of data in real-time.

Windows sockets, embodied by the WinSock v2.2 API, offer a good programming interface for peer-to-peer communications. The WinSock API is high-level and easy to program to, and it highly optimized for the Windows NT platform. It offers the highest data transfer rate of any peer-to-peer mechanism, especially when used with overlapped I/O. Providing protocol independence as well as protocol transparency when used with the Service Registration API, it is superior to standard Berkeley sockets. If offers all of the functionality of the Berkeley socket package, usually in the same way; porting code from Berkeley sockets to Windows sockets is trivial with a couple of minor exceptions. In addition to natively supporting protocols not offered by Berkeley sockets, Windows sockets bring a host of performance and ease of use advantages to the programmer, including the Service Registration API, overlapped I/O, the choice of synchronous or asynchronous I/O, quality of service support, and completion ports.

Both UNIX and Windows NT offer suitable mechanisms for creating shared memory segments. For the case of general shared memory, UNIX is easier to work with since the programmer need not worry about file mappings and the sequence of system calls o be used is somewhat more straightforward. UNIX claims an even larger advantage when mapping files into memory. UNIX requires only one mmap() call to complete the entire mapping operation whereas Windows NT requires a sequence of calls. Windows NT also mandates that the programmer cleanup after all shared memory operations, but UNIX handles this for the programmer when reference counts drop to zero or processes terminate.

Mailslots are a Windows NT-specific IPC mechanism that allows clients to send messages to repositories on its local computer, on another computer, or to all repositories with the same name on all computers on a given network segment. They serve as a standardized mechanism for sending a broadcast-style message. Mailslot messages are sent using datagrams for network efficiency, meaning that receipt of a message is not guaranteed. The closest approximation to a mailslot in UNIX is a named pipe.

6.2 Selecting an IPC Mechanism for Windows NT

Each of the IPC mechanisms discussed has its advantages and disadvantages; each is the optimal solution for a particular problem.

- **Anonymous pipes** provide an efficient way to redirect standard input or output to child processes on the same computer. **Named pipes** provide a simple programming interface for transferring data between two processes, whether they reside on the same computer or over a network.
- **Windows Sockets** are a protocol-independent interface capable of supporting current and emerging networking capabilities, such as quality of service monitoring, robust asynchronous communication, I/O completion ports for superior performance, and protocol-specific network features. In most cases, applications currently using files for communication can be easily adapted to use sockets.
- **Shared memory** is an efficient way for two or more processes on the same computer to share data, but the programmer must provide synchronization between the processes.
- **Mailslots** offer an easy way for applications to send and receive short messages. They also provide the ability to broadcast messages across all computers in a network domain. Since datagrams are used, they should not be considered a reliable means of IPC.

Choosing which of these technologies is up to the individual application programmer. After accounting for the balance of performance and ease of use desired and the technical requirements of the application, a selection can usually be made easily. In general, however, Windows sockets usually provide the best blend of performance, scalability, and ease of use.

7. References

In addition to numerous websites, MSDN articles, specification documents (particularly for Berkeley sockets and WinSock v2.2), and code samples, the following books served as references for this paper.

Beveridge, Jim and Robert Wiener. *Multithreading Applications in Win32.* Reading, Massachusetts: Addison-Wesley Developers Press, 1997.

Coulouris, George, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design, 2e.* Reading, Massachusetts: Addison Wesley, 1994.

Davis, Ralph. *Win32 Network Programming.* Reading, Massachusetts: Addison-Wesley Developers Press, 1996.

Gray, John Shapley. *Interprocess Communications in UNIX.* Upper Saddle River, New Jersey: Prentice Hall, 1997.

Stevens, W. Richard. *UNIX Network Programming.* New Delhi: Prentice-Hall of India, 1996.