UNIXWORLD
NETWORKED & MULTIUSER SYSTEMS

HETEROGENEOUS DISTRIBUTED APPLICATIONS

Confused by all the choices? Here's an overview of the client-server model to help you get started

By Eric W. Wasiolek

Eric W. Wasiolek is the product marketing manager for UNIX products at Excelan Corp., San Jose, Calif. He formerly worked with distributed applications at Informix.

UNIX has much of the technology that sophisticated business users covet: distributed processing, networked windowing, network mail, distributed file services, and more.

But the commercial market is firmly entrenched with non-UNIX systems from IBM, DEC, and Apple. As a result, it's critical for commercial UNIX applications, whether distributed or otherwise, to share information with non-UNIX systems.

Fortunately, UNIX applications may be built to connect to non-UNIX systems to address corporate America's emerging heterogeneous distributed applications requirements. OEMs, VARs, and software developers can use dissimilar systems communications technology available on the market today to turn standalone applications into heterogeneous distributed applications.

Actually, UNIX was among the first operating systems to incorporate a method to build distributed processing applications. It used Inter-Process Communications (IPC) pipes to send data between two processes on the same system and used BSD sockets to send data between two processes on the same or different systems. Only recently have other operating systems borrowed this technology—for example, named pipes and APPC (which is IBM's Application Program to Program Communications interface) in OS/2.

It was some six years ago that communications vendors extended the Berkeley socket model to non-UNIX platforms. The result was that, for the first time, developers could write applications that allowed processes on UNIX systems to communicate with programs on non-UNIX systems. Developers, for example, could write DOS to UNIX, DOS to VMS, or DOS to UNIX to VMS distributed applications.

A CHOICE OF INTERFACES

Today, many communications companies offer network programmatic interfaces in non-native environments (such as Netbios under XENIX or APPC under VMS). The result is that software vendors have, in most cases, a choice as to which programmatic interface to build their applications on.

Often, however, the programmer is confined to the interface available across certain systems. Despite recent developments in heterogeneous connectivity technology, the BSD socket model is still available across the greatest variety of systems. And most software vendors continue to build their heterogeneous distributed applications on faithful BSD sockets.

Today's UNIX users are faced with a dilemma. On the one hand, they increasingly enjoy the virtues of desktop computing with ever-increasing local processing power, more elegant windowing interfaces, and cheaper RAM and disk space. On the other hand, the mainstay of corporate information still resides on backroom mainframes and minis to which end users need access to do their jobs.

One simple answer is to place a 3270 or VT100 terminal on the desk beside the desktop personal system. But this is not cost-effective. Of course, you can buy software to allow the desktop system to emulate a 3270 or VT100 terminal. But this still necessitates that the user learn the operating system and application on the system where the corporate jewels reside. The user must also be able to locate the data on the mainframe or mini.

A much more sophisticated solution allows users to run a local application while accessing remote information. This is possible through the familiar interface of their favorite desktop, without specifying the location of the data. The solution is enabled by extending the distributed processing model of computing to span dissimilar systems and by the construction of heterogeneous distributed applications.

DISTRIBUTED APPLICATIONS

Distributed applications technology is founded on the principle of distributing the processing of the application across different CPUs, whether they are on the same (as in multiprocessor systems) or different systems. This computing principle enables greater CPU utilization across machines. With it, there is less idle CPU time, minimal network traffic, and each process may be specifically written to best utilize the resources of the system on which it executes.

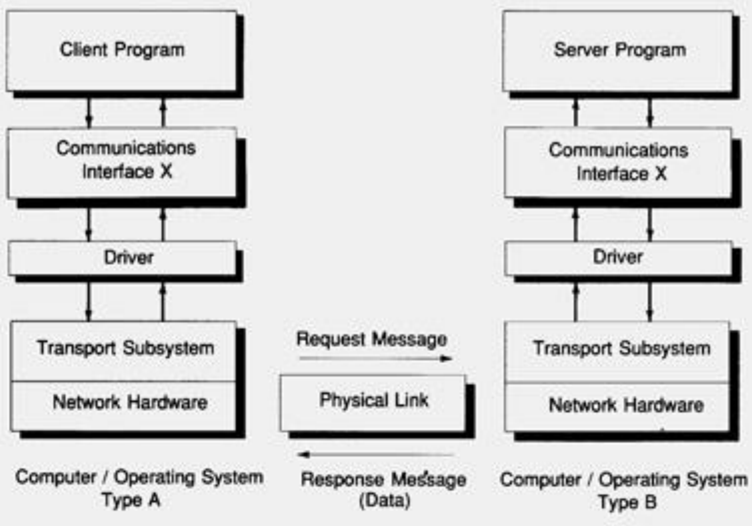
A distributed application is typically divided into a client program and a server program. There may be many instances of both the client program and server program on a single network. In most cases, client programs reside on desktop computers, while server programs are typically found on multiprocessing host computers.

The client program accepts user input, constructs a request message for server resources, and sends the message to the server program, which executes the request, and returns the requested information. (See Figure 1.) The client program displays the information to the user.

The communications subsystem acts as a vehicle to deliver request messages issued by the client program, and to return information supplied by the server program. To the application, the communications subsystem is a black box. The application writes messages to, and reads messages from, the communications interface, which is its sole interaction with the mechanism for delivering messages between dissimilar systems.

However, the method by which the message is delivered is not irrelevant. On the contrary, the specific implementation of the communications subsystem directly affects the overall quality of the resulting distributed application.

FIGURE 1: Messages are passed between the client and server programs running on different types of computers running different operating systems. The messages are passed through a common communications interface by an underlying intelligent transport subsystem.



CHOOSING THE COMMUNICATION METHOD

It is through the communications interface that the client program passes messages to the server program (and conversely). Moreover, it is the range of dissimilar systems across which the interface is offered that limits the range of systems across which the application may be distributed.

Several network programmatic interfaces are available. These include BSD Sockets, TLI/TPI, Netbios, named pipes, CL/1, and APPC. Each was developed in a different environment, and each offers connectivity between a different set of dissimilar systems.

Netbios is actually a session level (ISO level 5) interface definition. It lets you establish a reliable data stream (virtual circuit) and send packets, called network control blocks, between client and server programs across a network. Netbios is the standard network programming interface in the DOS world and is also available under XENIX.

APPC is on the LU6.2 protocol available on IBM mainframes as a part of IBM's Systems Network Architecture (SNA). As mentioned earlier, APPC is also available on OS/2 PS/2 systems. And some implementations are available for DEC VAX VMS systems, providing OS/2 to VMS to MVS connectivity.

CL/1 from Network Innovations is a high-level applications interface that allows Macintosh applications to access a variety of SQL-based databases and File Transfer Access Method (FTAM) files located on DEC VAX/VMS systems.

Named pipes are an extension of the UNIX IPC mechanism. They are available under OS/2's LAN Manager.

Finally, TLI/TPI is AT&T's UNIX V.3 programmatic streams interface and is available on any computer running UNIX V.3. Although TLI is increasingly popular as a UNIX distributed applications programming interface, it will not suit heterogeneous distributed applications until it is available under DOS, VMS, and OS/2.

As mentioned, BSD sockets are UNIX's first program-to-program communication mechanism. It is available as a standard part of the kernel on all BSD-derived systems, including Sun OS, and is also available on DOS, VMS, UNIX System V, XENIX, and OS/2.

OTHER METHODS

Also available are higher level network programming interfaces, including X.11, Postscript, RPC, XDR, NFS, RFS, SMB (MS-NET), XENIX-Net, and LM/X (LAN Manager). These interfaces let you send output to remote terminals, execute remote commands, and make remote file system calls. Each of these interfaces is itself a heterogeneous distributed application. Hence, any application that uses one of these interfaces is an application built on top of a heterogeneous distributed application. The result is that certain applications may be built with less effort. But they will be limited to the capabilities of the higher level interface and will be slow relative to building the same application directly on sockets, for example.

A software developer converts a standalone application into a heterogeneous distributed application in stages.

Unfortunately, no single interface spans all the systems that can reside in a large corporation. Hence, the software developer must choose between interfaces or offer multiple versions of the heterogeneous distributed application on different interfaces.

A software developer converts a standalone application into a heterogeneous distributed application in stages.

- Divide the application into two: (1) a client program that accepts input from and displays results to the user, and (2) a server program that accesses resources on the host.
- Modularize the client and server into an operating system interface portion, a communications interface portion, and the program body. By following this procedure, you isolate the core code from the different operating systems and communications interfaces to which that code may be ported.
- Establish message communications between the client and server processes on the same machine. You accomplish this, for example, by writing to socket descriptors whose addresses reside on the same system.
- Move the client or server's executable code to another machine of identical type; then establish message communications between a client running on one system across a network link to a server running on another system of identical type. You do this by reassigning socket descriptors.

At this point, the developer is running a homogeneous distributed application. He or she has, for example, a client process on one Sun system communicating (through a socket library over Ethernet using the TCP/IP protocol) to a server process listening through a socket library on a remote Sun system.

- Last, convert the homogeneous distributed application into a heterogeneous distributed application. This stage may require technology from the handful of independent communications vendors who specifically connect dissimilar systems. These vendors provide standard communications interfaces in non-native environments.

These firms include Excelan and Kinetics, the Wollongong Group, CMC, Micom-InterLan, 3COM/Bridge, Retix, and Touch Communications. Private consultants are also active in the field.

ACCOMPLISHING THE FINAL STEP

How do you accomplish this final step? First, you must port the client and server programs to the operating systems of the desktop and host computers on which the vendor wishes to market the application. In porting the operating systems interface module, the developer may take advantage of the special features of the particular operating system.

If, for example, the vendor is porting a client program to a Macintosh, the program will probably be more marketable if the mouse and icon libraries are utilized to yield the icon-driven interface to which Mac users are accustomed. The above-mentioned code modularization allows significant modifications of the user interface operating system-specific portion without affecting the core client code.

You must then port the client and server programs to a communications interface available across the range of systems over which the application is to be distributed. Suppose, for example, the software vendor wishes to have client implementations on DOS and OS/2, and server implementations on VAX/VMS and Sun systems.

Because the vendor (in this example) has already ported the communications module of the client and server programs to BSD sockets between two Sun systems, his work is virtually done. The vendor needs merely to acquire the BSD socket technology for DOS, OS/2, and VMS from a communications technology vendor.

He then makes minor modifications to ensure the socket communications module works properly and specifies to his customers that they get the appropriate communications run-time module (say, sockets for DOS) to run their application on that system type.

OUT IN THE MARKET

Based on the technology just described, there are several heterogeneous distributed applications on the market. Most are implemented on socket libraries.

Oracle Corp., for example, markets a heterogeneous distributed database. It offers users on DOS, UNIX, XENIX, VMS, or IBM MVS the ability to transparently share information dispersed anywhere across the network. Oracle's application is implemented on Excelan's dissimilar systems communications technology and socket library for DOS, XENIX, UNIX, and VMS.

Moreover, an Oracle DOS user may concurrently access remote database information and files on a Novell server, thanks to the communication technology's ability to run both IPX and TCP/IP protocols concurrently across Ethernet.

There are many other distributed applications on the market. Locus Computing offers X Windows products for both DOS and UNIX V.3/386 systems. These allow DOS or UNIX desktop users to view and interact with programs located on any dissimilar TCP/IP host running X.11 protocols.

PRODUCTS DIFFER WIDELY

Although it appears at first glance that the application interacts only with the communications subsystem through the communications programming interface, there is more to the story. The implementation of the underlying dissimilar communications technology directly affects the overall robustness, performance, applications independence, code maintainability, ease of development, and future migration capability of the heterogeneous distributed application.

The implementation also limits the range of dissimilar systems over which the application is distributed. Internal implementations of dissimilar systems communications technologies differ widely. How is the harried software developer to deal with this problem?

In selecting the programming interface, first determine whether it is available on the system types you wish to interconnect. Then, review the extent to which the interface adheres to the interface standard. For example, is the socket library implemented on DOS as it is under BSD UNIX? Your ease of development and the degree of total interoperability will be directly related to this degree of standardization.

Is the library consistently implemented across all operating system types? Does the communications subsystem reliably send messages between the required dissimilar systems? The answer to the last question should be yes even for systems that the communications vendor doesn't accommodate with products.

For example, does the communications subsystem interoperate with the kernel-based TCP/IP provided in all BSD systems (for which the dissimilar communications vendor does not specifically offer products)? Are messages delivered with the required speed? For example, is the physical network medium fast, and is the communications subsystem efficient?

THE POTENTIAL BOTTLENECK

Because communications times are significantly slower than local disk or CPU speeds, the communications link is the potential bottleneck in any distributed application. The ultimate test, and in large part what is meant by location transparency in distributed applications, is whether a remote operation appears to the user to have the same access time as a local operation.

Is the communications subsystem internally implemented such that it is minimally affected when operating system revisions are made? This consideration is most important since a heterogeneous distributed application uses a communication subsystem implemented under many different operating systems.

Many perverse problems can arise for several reasons: certain operating systems are typically revised every six months, and perhaps the application is distributed across several different operating systems. If the communications subsystem is not shielded from host operating system changes, there is a good chance that, too often, a communications subsystem under one of the operating systems will cease to function properly.

In selecting the programming interface, determine whether it is available on the system types you wish to interconnect.

Such failures create a maintenance problem that is impossible to solve. Certain vendors have taken great care to avoid this pitfall by implementing protocols on a board subsystem or within a host-based framework—both of which are separated from operating system kernels through a driver.

The latter implementation ensures a consistent implementation of protocols regardless of operating system type or revision. In contrast, some vendors implement their host-based protocols in the operating system kernel.

OTHER IMPORTANT QUESTIONS

You should ask other questions about the communications subsystem. Is it implemented so that you can easily add new types of system connections? Is there a provision, by virtue of the way the subsystem is implemented, to allow clients and servers to communicate between systems located on dissimilar networks? And can the communications subsystem communicate with, and eventually transition to, OSI protocols when they become prevalent?

In general, the communications subsystem will not be able to connect new systems quickly unless it is itself modular (the bus interface, operating interface, and protocol stacks should be contained in different hardware and software modules). The ability to allow applications to span dissimilar networks requires minimally that the subsystem can run different protocol stacks simultaneously.

Note also that for applications to span dissimilar networks that run both different protocols and different cables, the communications vendor must offer point-to-point communications through gateways.

The subsystem must be able to smoothly evolve into an OSI implementation without the purchase of new hardware. The ability to run multiple protocol stacks simultaneously allows existing hardware investments to evolve to OSI solutions with software updates.

No single communications technology interconnects on a single network all the systems that populate today's organizations, and offers a single consistent programmatic interface across all those systems. However, BSD sockets on Ethernet running TCP/IP best approximates that complete offering. OSI, perhaps in some combination with Systems Application Architecture (SAA)—that battle has yet to be waged—intends to offer the future's complete heterogeneous connectivity solution.

In the meantime, for present needs that must be met, it is best to build products on real, existing technologies, fill in the gaps with other spanking-new communications technologies where necessary, and wait for OSI migration. □