POINT OF CONTACT

Networking technologies for tying together multivendor hardware are maturing. Now it's time to turn to the challenge of integrating different types of software.

By Robert A. Gingell

A growing number of computing environments are being fashioned today out of networks of heterogeneous systems. The variety that characterizes these networks owes to several factors, among them the needs to:

* preserve investments in computing resources as new technology is assimilated.
* maintain economic flexibility by avoiding encumbering relationships with isolated vendors.
* share resources where value is a function of common access (such as corporate databases).
* share resources which are prohibitively expensive to duplicate (such as supercomputers).
* share resources which require special environments (such as noisy line printers).
* share human resource services (operators and system administrators, for example).
* distribute and dedicate inexpensive (or performance-critical) resources to specific, specialized functions. File servers, ''user-interface servers'' (graphical workstations), and data monitoring and collecting equipment are examples.
* provide an integrated environment in which the functions of each resource can be harnessed directly both by users and other resources.

As the first major portable operating system, UNIX actually helped to establish many of these requirements. Indeed, it might be said that many of the seeds for heterogeneous networks were sewn by UNIX users, who for years have chosen freely from among a variety of hardware vendors.

An analogy can be drawn between the emerging networked computing environments and component stereo systems. In the instance of stereos, a task (making music) is distributed over a variety of components, each dedicated to performing some part of the overall job. Systems typically are built over time through several purchases. As a result, they generally consist of components from several different vendors (given that most consumers try to take advantage of the best available price/performance deals). As a whole, then, a stereo system can be seen as an evolving compromise between desired performance, the state of technology at various points in time, and the budgetary constraints of the user.

Component stereo gear would not even be possible today had vendors not agreed on interconnection standards. Comparable efforts have been made in the computing arena, but the task has been complicated by the existence of differing standards for various networking facets (for example, the Ethernet and Token Ring *physical medium* standards, and the TCP/IP, DECnet, ISO, and X.25 *transport control* standards).

Compounding the computer interconnection challenge is the enormous variety of network *applications*. With stereo systems, there is only one application to worry about--the making of music. Computer vendors, though, must shape standards that can accommodate and evolve with innumerable applications. It is here that the obstacles lie to creating an integrated, heterogeneous, ''component computing'' environment.

DISTRIBUTED OPERATING SYSTEMS

The two major approaches used to integrate networks of computer systems can be classified as *distributed operating systems* and *network services*. In effect, distributed operating systems achieve a high degree of integration by transforming a network of components into what amounts to a loosely-coupled multiprocessor. Examples include Apollo's DOMAIN [5], UCLA's distributed UNIX (known as LOCUS) [6], and DEC's VAX/VMS on VAXclusters. By using the same software architecture on every hardware system in a network, these operating systems

essentially limit the problems of heterogeneity to hardware. In some cases, even hardware heterogeneity can be limited to a degree by incorporating appropriate assumptions about the environment.

Many organizations want to have an integrated network to create a more productive computing environment out of previously autonomous installations [4]. These users *already* have an extensive hardware and software investment in systems ranging from PC's to mainframes. The imposition of a single operating system, even if it could support a wide range of hardware, is unacceptable for these users because it invalidates their current software investments. Further, this approach constrains the evolution of *any* networked computing environment to match the pace of their single software base. Users need a more general and flexible means for integrating heterogeneous systems.

NETWORK SERVICES

By contrast, the network services approach described the facilities available to applications in a network in the form of standardized, system-independent interfaces called *services*. This approach to providing resource sharing is the one employed in the ARPAnet and the networking applications supplied with 4.2BSD. Incorporated in the Berkeley applications set are utilities for handling standard ARPAnet protocols as well as some standard UNIX facilities customized for accessing other UNIX systems in a network.

A number of these tools are limited with respect to accommodating heterogeneity, though. An example can be found in the *C Advisor* column in the May 1985 issue of UNIX REVIEW. The sample code in that column consists of a pair of programs intended to exchange binary data through a pipe. As the column illustrates, the data is interpreted correctly when the pipeline runs on a single machine (say, a VAX):

```
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

But when the pipeline must be partitioned across machines of different architectures (a Sun and a VAX, for instance), surprising results can be produced:

```
sun% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296 117440512
sun%
```

This discrepency results from the byte-order differences between the two architectures. Distressingly, **lint** will not complain about this problem since it tests for the portability of *programs* rather than the *data* they produce or consume. Such problems are also beyond the scope of the 4.2BSD kernel-supplied communications facilities used by **rsh** since they operate strictly at the *transport level*, meaning they focus exclusively on data *movement*, not data *interpretation*.

As experience with networking has grown, though, network architects have come to recognize that applications development support requires more than data transport. Thus, the International Standards Organization's seven-layer model for network architectures defines a *presentation* layer for the resolution of *data representation* problems [2]. Other network architects, meanwhile, have recognized the semantic similarities between procedure calls and the subprotocol embedded for synchronous command and response by many application protocols like **ftp** and **telnet**. This has led to the notion of Remote Procedure Call (RPC) protocols [10].

Factoring out this application-independent subprotocol to a separate layer provides application designers with a common foundation for use in constructing network applications. In addition to being easier to build, programs produced in this way are able to use the familiar paradigm of a procedure call to acquire network services. This contributes to the development of intimate environments by allowing programs to treat remote services almost as if they were local library routines. Many RPC mechanisms also embed facilities for producing architecture-independent data representations for passing parameters and results to and from remote procedures.

There currently are two popular RPC systems available for UNIX. One is an implementation of Xerox's Courier RPC system [3][11], based on the XNS protocol architecture. The second is Sun Microsystems' RPC package [9], which is largely transport-independent. Partly because of this, it has gained not only a broad UNIX following but also has been ported to such other environments as MS-DOS and IBM's MVS [4].

An interesting aspect of Sun's implementation is that its data representation standard, the External Data Representation (XDR), is packaged separately. The separation of data representation from the RPC discipline leaves the

XDR facilities open for interchanges that are not RPC-based. In fact, the previously referenced *C Advisor* ended up solving the problem of data exchange between heterogeneous hardware by writing and reading the binary numbers via XDR facilities. This was done by modifying the programs so that they made calls on the XDR libraries rather than reading or writing directly through standard I/O facilities. It also would have been possible to create filters for each system and then extend the pipeline to include them. For example:

```
sun% writer | sun_long_to_xdr | rsh vax 'xdr_to_vax_long | reader'
0 1 2 3 4 5 6 7
sun%
```

This approach of ''stacking'' processing components is similar, though not identical, to Dennis Ritchie's *stream* mechanisms [7].

NETWORK SERVICES FOR INTEGRATED, HETEROGENEOUS ENVIRONMENTS

The simplified application environment provided by a foundation like RPC makes it possible to build some very sophisticated network services. However, the architecture-independent mechanisms of RPC and the data-representation mechanisms of XDR hardly are panaceas for the problems of heterogeneity. Indeed, care must be taken when building a service on top of RPC and XDR to define the service with interfaces every bit as architecture-independent as the mechanisms that will be used to access it. Examples of such interfaces can be found in Sun's Network File System (NFS) [8].

The NFS implements a service by which machines in the same network can share file systems. By providing *access* rather than a *copy* service (like **ftp** and **rcp**), the NFS leverages all of the resources in a network environment. The key is that it provides a common space of file objects on which all the resources can operate. The NFS itself consists of a protocol and appropriate implementations of it for each of the systems participating in a given network. The ''protocol'' is a description of the procedures, arguments, and return values to be used in each of these implementations.

Even though most NFS implementations are UNIX-based, and the abstract file system the NFS provides has UNIX-like features, the NFS should be thought of as a service for which UNIX-based clients and servers can be implemented--not as a network-wide implementation of the UNIX file system. This is an important viewpoint to take when designing an architecture-independent service. A consequence of this design practice, though, is that those functions whose semantics are accepted only on isolated native operating system environments are completely omitted from the network service interface. UNIX, for instance, allows processes to read directories as normal files, but this by no means is a universal attribute of file systems. The NFS service therefore prohibits the direct reading of directory files, instead providing ''database-like'' operations on directories by which various hosts can search for and retrieve file entries.

The implementation of a service partitions naturally into two halves: one which *calls* the procedures defined by the service interface (*client*), and one which *implements* the entry points (*server*). This permits asymmetric service implementation across the members of a network--an important point of flexibility in networks that include systems of diverse capabilities. For instance, the lack of multitasking in MS-DOS prevents the system's use as a platform for implementing many *servers*, but that hardly keeps MS-DOS-based systems from participating as *clients* of many services.

This ability to accommodate asymmetry gives the network services approach a clearcut advantage by permitting resources to participate in networks to the extent that they can. The alternative would be to set ''entrance requirements'' that would surely exclude certain resources altogether.

As ''entrance requirements'' are minimized, the likelihood that any given system will be able to participate grows. The effect of increasing hetereogeneity this provides should motivate developers to specify service interfaces that *stateless* servers can implement. Stateless servers need not retain any information about their clients between transactions. By contrast, statefull servers do maintain information, creating a (perhaps implicit) requirement on service participants to keep each piece of information current. To do so even in the face of network or other failures increases implementation complexity and may exclude some potential service participants.

Thus, service designers need to weigh the value of features that require statefull implementations against the costs of implementing them. In some cases, it may be possible to factor a single service requiring statefull implementation into several distinct subservices, several of which may be stateless--thus permitting layered and

asymmetric participation.

Beyond developing new functions, service designers must consider that evolving technology and changing user needs will require parallel modifications in existing services. Since this means that the interfaces supplied by services will change over time, users must be armed with version-control techniques. It simply is not feasible to instantaneously update all the members of a network, so real systems will have to be able to accommodate *version heterogeneity* as well as component diversity. The asymmetric qualitities of network services can be applied here again, since servers and clients can be updated independently so long as at least one matching server exists for each version of each interface required by each client.

Implementations of the same service specification on different systems create yet another problem: *service heterogeneity*. This refers to the fact that services generally come in varying flavors that differ in ways not defined by their interfaces--such as performance or cost. For example, a **troff** service may be provided by both a small microcomputer and a supercomputer. Both flavors may be the ''same'' in that they both can process a document for typesetting, but while one will complete the task in hours for a low cost, the other will be able to complete the job in seconds for a significantly higher cost.

Users of an integrated environment generally do not want to worry about all the details of what, where, and how a service is to be performed. More often than not, they prefer to express needs on a higher level, such as: ''I want my **troff** done sometime today'', or ''I need this **troff** done right now, no matter what the cost''. This illustrates the need for flexible, often transparent, *binding* techniques between servers and clients. This requirement actually transcends the issue of heterogeneity since good binding strategies can be used to address--among others--the system-wide issues of reliability and load-sharing through replicated servers.

Finally, it is important that any integrated environment be able to focus all of its various processing resources on a single problem. Although this involves binding, the problem is far more involved than simply selecting the ''right'' server. It means that, in addition to being able to specify execution activities, users must be able to designate the ''context'' in which these activities take place. For UNIX users, this context includes such items as:

* The current working directory.
* The controlling terminal.
* The file creation mask.
* Ignored signals.
* The shell environment.

Most other operating systems have similar notions, but generally express them differently. An ''execution service'' needs to translate system-dependent items in such a way as to preserve the integrity of user-created work environments.

CONCLUSION

Although the problems confronting users of increasingly heterogeneous systems have spawned some radically different responses, all approaches contain a common thread: the accommodation of heterogeneity is really a process of building interfaces at the points at which all systems are *homogeneous*. The key lies in not only selecting an appropriate form for an interface, but in locating it at the right spot. It also is important that interfaces be distributed over sets of services which make only minimal ''entrance requirements'' because it is this that gives users the flexibility to accommodate hardware and software heterogeneity as their computing environments evolve.

The task facing us all is to specify and implement of network services by which heterogeneous environments can be integrated. In many cases, these services will have to provide facilities that previously have been accessible only through program libraries (such as workstation graphics and window system libraries). The accommodation of heterogeneity will require remote interfaces that make such services generally available across networks [1].

ACKNOWLEDGEMENTS

A number of colleagues at Sun Microsystems offered helpful comments and insights during the preparation of this article--notably Bob Lyon, Gary Sager, and Bill Shannon.

BIOGRAPHY

Rob Gingell is Manager of New Systems in the Systems Software Group at Sun Microsystems. Prior to coming to Sun, he worked at Case Western Reserve University on research in computer design and modelling tools, computer graphics, and later in development of operating systems, networks and campus computing facilities.

REFERENCES

[1] J.A. Gosling, ‘‘SUNDEW: A Distributed and Extensible Window System’’, *Usenix Association: Winter Conference Proceedings*, pp. 98-103 (Jan. 1986).

[2] Mark Hall, ‘‘The Potpourri of Networks’’, *UNIX Review*, pp. 25-26 (April 1985).

[3] Steven F. Holmgren, ‘‘The Untapped Potential of Remote Procedure Calls’’, *UNIX Review*, pp. 35-42, 92 (May 1985).

[4] S.N. Kahane, S.G. Tolchin, M.J. Schneider, D.W. Richmond, P. Barta, M.K. Ardolino, and H.S. Goldberg, ‘‘Windows in the Hospital or a Workstation-Based Inpatient Clinical Information System in the Johns Hopkins Hospital’’, *Usenix Association: Winter Conference Proceedings*, pp. 45-61 (Jan. 1986).

[5] D.L. Nelson and P.J. Leach, ‘‘The Evolution of the Apollo Domain’’, *Hawaii International Conference on Systems Science* (1984).

[6] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, ‘‘LOCUS: A Network Transparent, High Reliability Distributed System’’, *Proceedings of the Eighth Symposium on Operating Systems Principles*, pp. 169-177 (Dec. 1981).

[7] D.M. Ritchie, ‘‘A Stream Input-Output System’’, *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Part 2, pp. 1897-1910 (Oct. 1984).

[8] G.R. Sager and R.B. Lyon, ‘‘Distributed File System Strategies’’, *UNIX Review*, pp. 28-33, 94 (May 1985).

[9] Sun Microsystems, ‘‘Networking on the Sun Workstation’’, Part No. 800-1177-01 (May 1985).

[10] James E. White, ‘‘A High-Level Framework for Network-Based Resource Sharing’’, *AFIPS Proceedings of the National Computer Conference*, pp. 561-570 (June 1976). Also available as RFC:RFC707.TXT at the ARPAnet Network Information Center.

[11] Xerox Corporation, ‘‘Courier: The Remote Procedure Call Protocol’’, XSIS 038112 (Dec. 1981).