The Domain system's objective is to integrate mainframe capability with local area networking and raster graphics capabilities at a cost appropriate to engineering and graphics applications.

The Architecture and Applications of the Apollo Domain

David L. Nelson and Paul J. Leach
Apollo Computer, Inc.

Apollo Computer's Domain system is a distributed processing system designed for both general-purpose and interactive graphics applications. It is a collection of powerful personal workstations and server computers interconnected by a high-speed local area network. Both workstations and server computers are referred to as nodes and are capable of running very large, complex applications. The personal workstation is provided with a high-resolution, bit-mapped display, so that each user can display the output of programs written in Fortran, Pascal, or C.

A Domain server-processor can act as a file or peripheral server, as well as a gateway to other networks. All workstations and server-processors share a common network-wide virtual memory system that allows groups of users to share programs, files, and peripherals. Figure 1 shows a Domain personal workstation node with display. Figure 2 is a schematic overview of the Domain network and software systems. This article describes the architecture of the Domain system and the benefits of this architecture to general-purpose applications.

Evolution of the architecture

It has frequently been observed that in their evolution, mainframes, minicomputers, and microcomputers trail each other by about seven years in performance and capability. At a technological improvement rate of 30 to 40 percent per year, seven years yields the approximate factor-of-ten separation between these three classes of machines. Consequently, a 1975-vintage superminicomputer costing $300,000 can be expected to perform comparably to a 1982 microcomputer-based machine costing about $30,000.

Presently, individual users can afford to purchase a single computer system that gives them a predictably high level of computing. At this point, however, users may desire more from their systems than simple computing: They may discover a need for communication. Communication is an essential part of a computing environment; it includes the sharing of databases, programs, expensive peripherals, and interfaces, as well as communication with other users.

As the computing environment becomes more comprehensive, communication plays an increasingly important role. Consequently, stand-alone, dedicated computers must include the ability to communicate rapidly with other computers. In most applications, the trend toward dedicated workstations goes hand in hand with the trend toward local area networking.

What are the desired architectural characteristics of such a dedicated/network form of computing? In principle, these characteristics are well understood:

- The computer must have a mainframe architecture capable of running large, complex application programs.
- The computer must have a highly interactive, bit-mapped graphics capability.
- The local area network must offer high performance and must be well integrated into the overall system design to facilitate ease of use.

Implementing these architectural features on a low-cost system has been the primary objective of the Domain.

Overview of the architecture

The Domain system consists of three main features: a processor with significant computational power and a large virtual address space; a high-speed local area network; and a high-resolution, bit-mapped graphics display subsystem. These three features interact to support system objectives in the following ways.

First, the high computational power of a Domain node combined with its support of a large virtual address space

0272-1716/84/0400-0058$01.00 © 1984 IEEE

permits tools traditionally used on mainframes and superminicomputers to be ported easily to the node and to operate with a performance that often exceeds the performance attained in a timesharing environment. In addition, a nodes's high computational power plus its display subsystem creates a user environment that increases productivity. In particular, the display can be divided into multiple windows that the user can control simultaneously.

The network and a network-wide distributed file system permit the professional to correspond with colleagues via electronic mail and to share programs, data, and expensive peripherals in much the same way as on larger shared machines, without the disadvantage of having to share computing power.

The Domain architecture creates an "integrated distributed" environment: It is distributed because users have their own nodes, and it is integrated because the system provides mechanisms and the user can select policies that permit a high degree of cooperation and sharing when so desired. (Locus, as described by Popek et al.,[1] and Eden, in Lazowska et al.,[2] are other examples of integrated distributed systems. The role of autonomy in distributed systems is discussed by Svobodova et al.[3])

The Domain system fosters cooperation and sharing by allowing users and programs to name and access all system entities in the same way, regardless of their location on the network. Consequently, all users and application programs view the system as an integrated whole rather than as a collection of individual nodes. Yet the Domain system is still a distributed system, not a loosely coupled multiprocessor. For example, the emphasis on node autonomy and the expectation of partial network failures led designers to develop the design criterion that suitably configured machines must always be able to run even when disconnected from the network.

Figure 1. An Apollo personal workstation node with a high-resolution (1024 × 1024 × 8 plane) color display.

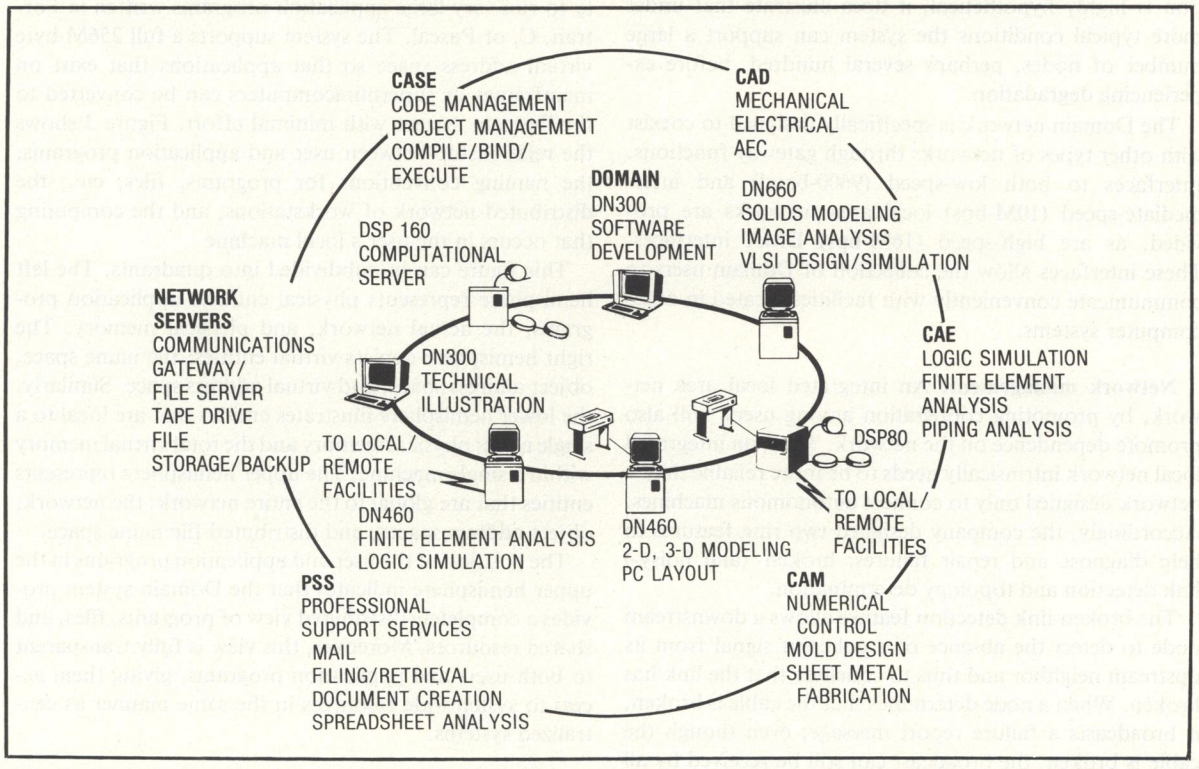


Figure 2. The Apollo Domain has personal workstation nodes and server nodes (inner circle), and software systems (outer circle) to support sharing, cooperation, and communications for a wide variety of professional applications.

The network

In the Domain architecture, the local area network occupies the place usually given to the backplane interconnect in other architectures. The network is the system integration point and the primary architectural feature held constant across new hardware implementations, instead of a particular CPU instruction set. This design allows Domain to choose, at any given time and for a given cost/performance goal, the best implementation of nodes using state-of-the-art technology. Because the architecture specifies only what the nodes "see" through the network, it does not constrain the implementation of the nodes themselves.

Network hardware. The network is a 12M-bps token-passing ring. Other ring implementations are discussed by Gordon et al.[4] and Wilkes and Wheeler.[5] Individual nodes transmit data onto the ring by first acquiring the token, which normally travels around the ring at a very high rate. The system guarantees that only one token exists on the ring at any given time; consequently, in normal operation the network never experiences collisions (or retries due to collisions) that would impair efficiency. (See Saltzer, Clark, and Pogran[6] for a discussion of token rings versus contention-based access arbitration.)

At 12M bps, the network can deliver in excess of 1000 pages per second. Since (when accessed randomly) the Winchester disks used with the nodes provide approximately 20 pages per second, the worst-case configuration can involve up to 50 pairs of nodes, where one member of the pair is requesting pages over the network as fast as its partner can deliver them from its disk. Although this situation is highly hypothetical, it does illustrate that under more typical conditions the system can support a large number of nodes, perhaps several hundred, before experiencing degradation.

The Domain network is specifically designed to coexist with other types of networks through gateway functions. Interfaces to both low-speed (9600-baud) and intermediate-speed (10M-bps) local area networks are provided, as are high-speed (16M-bps) DMA interfaces. These interfaces allow the collection of Domain users to communicate conveniently with facilities located in other computer systems.

Network management. An integrated local area network, by promoting cooperation among users, will also promote dependence on the network. Thus, an integrated local network intrinsically needs to be more reliable than a network designed only to connect autonomous machines. Accordingly, the company designed two ring features to help diagnose and repair failures: broken- (and noisy-) link detection and topology determination.

The broken-link detection feature allows a downstream node to detect the absence of a coherent signal from its upstream neighbor and thus to conclude that the link has broken. When a node determines that the cable is broken, it broadcasts a failure report message; even though the cable is broken, the broadcast can still be received by all other active nodes on the network, since they are downstream from the detecting node and the break.

Determining the network topology means determining the order in which the nodes are joined to form the ring. The network controller provides the mechanism for construction of a topology map. With a broken-link report and a network topology map, link failures can be isolated to the nodes at either end of the link.

To aid in recovery from link failures, the network can be installed in a star-shaped configuration as described by Saltzer and Pogran.[7] A star-shaped configuration is electrically a ring but topologically a star; in such a configuration, each subring, or loop, is brought back to a central point. The star configuration permits the loop containing the failure to be switched out so that operation can be restored to the rest of the network.

Good network maintenance and troubleshooting facilities are essential to the success of a large local network. The star-shaped ring has been crucial to managing the Domain network. Presently, the network at Apollo Computer corporate headquarters contains over 240 nodes; in general, more than 200 of these nodes are active at any one time. Nodes belonging to every facet of the company are attached to the same ring, including those in the research and development, finance, marketing, customer service, and manufacturing departments. Without the ability to make the almost continual network changes inherent to a large network (moving old nodes, adding new ones, and adding new network subloops), network maintenance would be much more difficult.

Architectural features

A predominant feature of the Domain system is its ability to run very large application programs written in Fortran, C, or Pascal. The system supports a full 256M-byte virtual address space so that applications that exist on mainframes or superminicomputers can be converted to the Domain system with minimal effort. Figure 3 shows the relationship between user and application programs; the naming conventions for programs, files, etc.; the distributed network of workstations; and the computing that occurs in the user's local machine.

This figure can be subdivided into quadrants. The left hemisphere represents physical entities: application programs, the actual network, and physical memory. The right hemisphere depicts virtual entities: file name space, object address space, and virtual address space. Similarly, the lower hemisphere illustrates entities that are local to a single node: physical memory and the total virtual memory within a single machine. The upper hemisphere represents entities that are global to the entire network: the network, object address spaces, and distributed file name space.

The position of the user and application programs in the upper hemisphere indicates that the Domain system provides a completely distributed view of programs, files, and shared resources. Moreover, this view is fully transparent to both users and application programs, giving them access to systemwide resources in the same manner as centralized systems.

Object storage system. The structure of virtual memory is based on the object: 32-bit, byte-addressable virtual ad-

dress spaces accessible from anywhere in the network. Objects are generalizations of data files, executable modules, interprocess communication (IPC) mailboxes, bit maps, directories, and other system entities.

The system assigns to each object a 64-bit, unique identifier string, or UID, which it creates by concatenating the unique node ID of the node generating the object with a time stamp from the node's timer. The UID is the mechanism by which the object is located. See Leach et al.[8] for a complete description of the use of UIDs in the Domain distributed file system.

What are the advantages of object-based systems? In conventional timesharing systems, separate mechanisms are frequently used to implement similar system functions. For example, programs can be managed by a paging system, whereas data files are accessed and handled through a file system. Thus, two distinct system mechanisms exist to handle similar system entities.

In contrast, the Domain system deals exclusively with objects, without regard to their physical location on the network or their specific function (program, file, etc.). The object abstraction simplifies the overall design of the system by casting all system entities into a common framework and managing them with a common set of mechanisms.

Objects are very large (over four billion bytes), linear virtual address spaces. Only the allocated, nonzero portion of an object's address space actually exists on a physical disk; these portions are managed by the disk management system. Allocation can be sparse; that is, the allocated regions of an object can be arbitrarily fragmented throughout the address space and need not be contiguous.

At any given time, the permanent storage for an object is entirely on one node; thus, for every object there is a particular node that acts as the object's "home node." This node is the site of all operations performed on the object.

Each distinct type of system resource is controlled by a type manager. Type managers exist for disk files, mapped objects, serial I/O lines, IPC mailboxes, and other object types. Each object has a type UID that identifies which type manager is to handle it.

User programs gain access to system resources by calling the stream manager, the system's device-independent I/O mechanism. When called, the stream manager determines the object's type and redirects control of the operation to the proper type manager. Stream manager operations are similar to Unix system calls for manipulating files and peripherals, as described by Ritchie and Thompson.[9]

Translation. The relationship between virtual and physical memory in a single node is maintained by memory management hardware designed to enhance the central processor to the level of a mainframe computer. Memory management functions include

- dynamic translation of virtual addresses to physical addresses,
- pagination of physical memory into 1024-byte pages,
- read, write, and execute protection,
- collection of statistics on page usage, and
- transparent support for page faults that occur in virtual memory systems.

Paging. The system allocates objects in units of 1024-byte pages. Each node has a paging server process that handles remote requests to read and/or write pages of objects on that node. When a page belonging to an object is referenced by another node on the network, the paging server dynamically transfers, or demand pages, it to the requesting node. Since the local area network operates at 12M bps, it can transfer a page to any node on the network in less than a millisecond—a small delay compared with the access time of moving-head devices.

When combined with adequate virtual memory management hardware, this underlying network paging system can provide a user process with a global view of objects residing in the network.

Mapping. A unique aspect of the Domain system is its network-wide single-level store mechanism. With SLS, a program gains access to an object by requesting (via the object's UID) that it be mapped into its address space. Subsequently, the object is accessed with ordinary machine instructions using virtual memory demand paging, as described in Leach et al.[10] Multics, described by Organick,[11] and IBM System/38, described by French et al.,[12] are examples of the use of the SLS concept in centralized systems; Domain is the first application to a distributed system.

This mapping between object space and process address space is the fundamental primitive of the Domain architec-

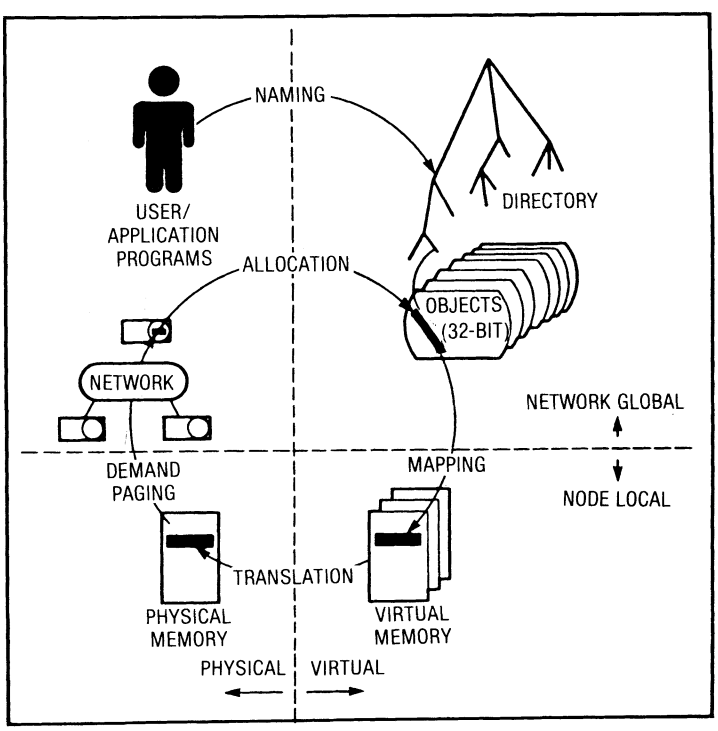


Figure 3. Relationship of the system resources in the Apollo Domain: physical, virtual, local, and network. The user and applications programs are in the upper hemisphere where they have a completely distributed view of these resources.

ture; it provides a single-level storage view of programs, files, and other system entities. Objects can be mapped to process address space with various protection rights, they can be shared among multiple processes, and they can be extended by CPU instructions that simply write into unallocated regions.

Mapping proceeds independently of whether the object is local or remote. Thus, the single-level store mechanism provides a uniform, network-transparent way to access objects. As a result, the user can execute a program without being concerned about its location or the location of the files it uses. For example, it is possible to execute on node A a program that resides on node B, reads input from node C, and creates output on node D.

Together, the network paging system and extensive memory management hardware provide this simple and elegant mechanism to manage network global objects. Experience has shown that this network virtual memory system eliminates the need for many conventional network functions, such as file transfer protocols. It also allows users to share single copies of programs and data files.

Concurrency control. The Domain single-level store implementation is distinguished from single-level store on a centralized system by its need to handle multiple main memory caches, one for each node on the network. This need leads to the problem of synchronizing the caches, that is, to locate and retrieve the most up-to-date copy of an object's page on a page fault, and to avoid using stale pages (pages still in one node's cache that have been recently modified by another node).

One objective of synchronization is to give programs a consistent view of the current version of an object, since there can be many updaters. (See Kohler[13] for a survey.) A second objective is to offer a synchronization algorithm that is simple and that requires a small database, because as part of the implementation the database must be permanently resident in main memory.

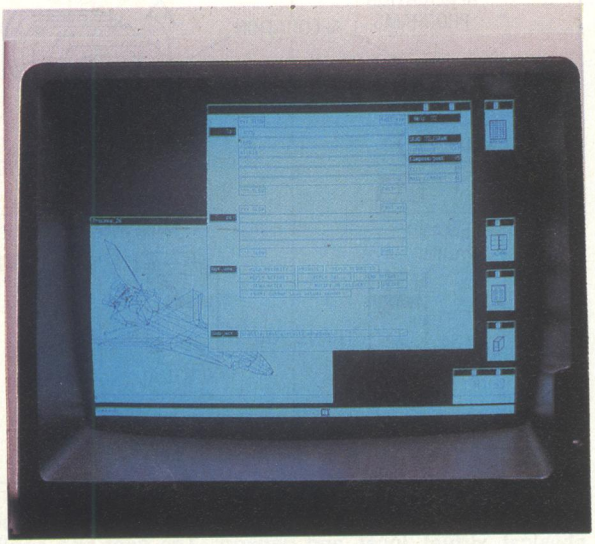


Figure 4. Multiple windows allow interaction with, and control of, several concurrently executing tasks.

These objectives appear, for practical purposes, to be mutually exclusive. Hence, the SLS implementation guarantees neither consistency nor the use of the current version. Rather, a more relaxed condition is met whereby applications are provided with operations and information from which they can build mechanisms that make the stronger guarantees. In addition, an application can use the virtual memory provided by the SLS and thereby gain a measure of freedom from the aforementioned constraints on its size and the size of its database.

Concurrency control in the Domain architecture is based on each version of an object; the system detects a concurrency violation when an attempt is made to use more than one version of an object. To implement concurrency-violation detection, each object has a time-stamp version number that records the time the object was last modified. This value is maintained at the object's home node and is the means by which remote reads and writes are synchronized.

Using object version numbers for concurrency control leads to the following conditions:

(1) Concurrency violations can only occur in multinode situations. If an object is used by only one node, that node is the only source of version number changes and hence always sees a consistent view of the current version.

(2) Even in multinode situations, concurrency violations will not occur if an object is used as read-only; all nodes will have the same version number and see the same version.

(3) If no concurrency violations occur, all users of an object see a consistent view of the object. However, even then, some could see an out-of-date version. Therefore, if multiple objects are used, there is no guarantee of interobject consistency.

(4) An object must be read before being written; thus, if two nodes read from the same version of an object and then each writes the object, the first writer will update the version number, causing the second writer to get a concurrency violation. However, if the nodes' read/write pairs can be serialized, no violations will occur.

The Domain system also provides a readers/writers locking mechanism at the higher level; however, users are free to construct their own synchronization mechanism at this level if they do not wish to use Domain's.

Naming. Since objects can be demand-paged from remote disks at rates comparable to local disks, it is desirable to present to the user a homogeneous view of the collection of objects, independent of their physical location. To meet this objective, the Domain system allows users to reference objects by means of a network-wide, hierarchical name space whose structure is similar to file structures found in modern timesharing systems. All programs, files, etc. are referenced across the entire system using a name space that consists of a multilevel directory tree, with directories at the nodes and other objects at the leaves. The user refers to an object with a text string, or pathname. The Domain system's naming server then translates the pathname into the object's UID.

Experience has shown that a user's local naming server can efficiently translate object names anywhere on the net-

work without perceptible response-time delays. The reason for this is that the directory structure is itself implemented as a collection of objects so that they can be mapped efficiently into the naming server's address space and referenced as if they were local.

Therefore, the designers implemented a system in which each user sees a distributed database of objects with the same appearance and access-time efficiency as that of a centralized, timesharing system. The performance of the local area network, together with the memory management hardware, provides the basis for this view.

The user environment

Within each node, the user environment is managed by a process known as a display manager. The display manager allows the user to view and control separate activities both concurrently and independently by dividing the screen into windows whose size, shape, and placement are under user control. (See Figure 4.) Windows are viewports to different types of "pads," similar to the ones invented by Lantz and Rashid.[14] An edit pad displays an object that the user can modify, given the proper access rights. Input and transcript pads provide a process with a virtual terminal; the process writes output and reads keyboard input from these pads. Multiple windows can be overlaid on top of each other, completely or in part, so that the user environment is analogous to a desk upon which various pieces of paper are placed, except that the "pieces of paper" are actively performing some function or displaying some graphic output in a window.

Multiple processes can run concurrently, and each process sends its output to its own pad. User programs can also use display manager windows or the entire display for graphics operations.

The integration of multiple windowing and the Domain virtual memory system means that users can simultaneously perform design, analysis, development, and communication functions in a significantly larger environment than that possible with conventional stand-alone workstations. In addition, the Domain system offers a related set of tools that support technical professionals in their day-to-day activities. These generic service tools include:

(1) A calendar utility that can automatically schedule, confirm, and signal the onset of daily, weekly, and monthly events. (A calendar display is shown in Figure 5.)

(2) A highly visual electronic filing cabinet that manages "files," "folders," and "drawers."

(3) A mail program to send, reply, and forward to one, several, or a mailing list of recipients. (Figure 6 displays a typical mail program.)

(4) A document editor/processor that incorporates and expands upon the features associated with traditional word processing systems. It permits interactive manipulation of multiple-font text and graphics. (See Figure 7 for an example of a document editor.)

(5) A report editor/generator that allows a user to manipulate rows and columns of a spreadsheet and performs mathematical functions such as adding up columns of numbers and calculating percentage increases from one row to the next.

Together, these generic service functions maintain the traditional terminology of the workplace—calendars, worksheets, drawers, writing pads, and file cabinets—while providing a highly sophisticated and integrated set of electronic tools and services.

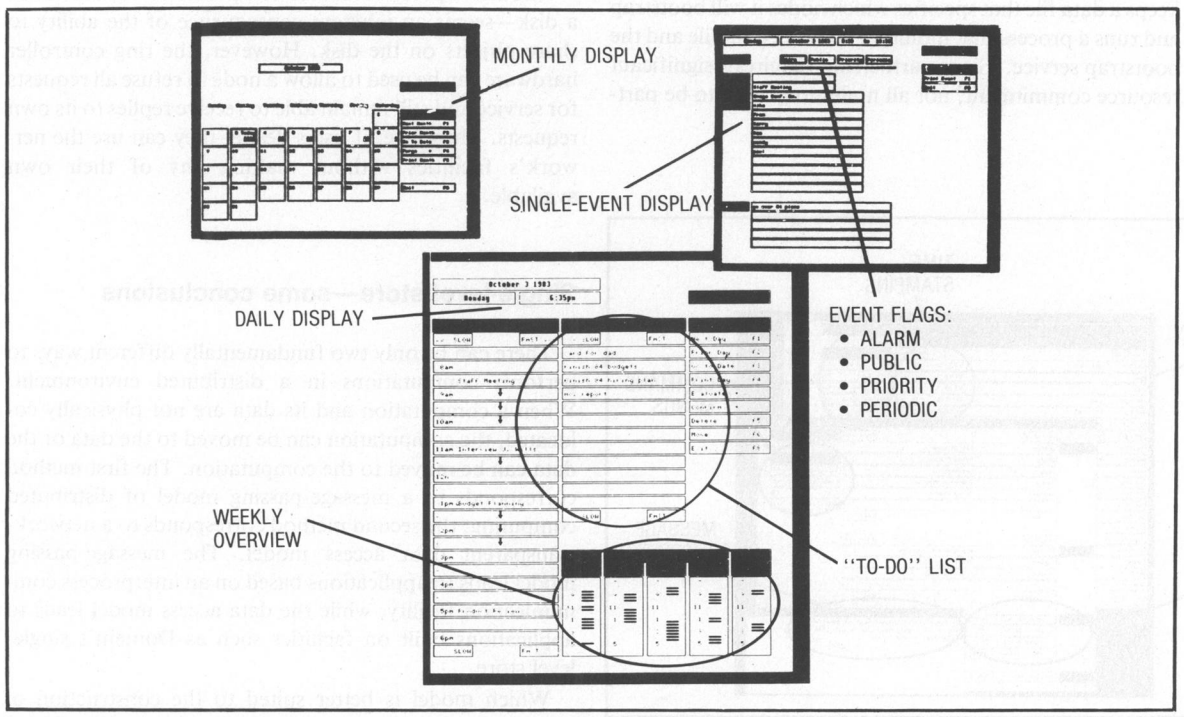


Figure 5. Calendar utility display.

The Domain user environment also offers Aux, a Unix System III-compatible software environment with Berkeley 4.2 enhancements. Aux is fully integrated into the Domain user environment. The Aux environment allows Unix users unfamiliar with the Domain system to use the Unix shell, run Unix utilities, and receive the expected responses to Unix key sequences. In short, users feel as though they are using a Unix system, and they can develop expertise with the Domain system at their own pace.

Users can intermix Domain and Aux programs, even within a single pipeline, without any awareness of the programs' origins. Because the Aux and Domain user environments are fully integrated, Aux and Domain programs can be executed from either Aux or Domain environments, and files written in Domain and Aux environments can be read in either environment. Thus, Aux provides the user with the benefits of the Unix software and user environment on a dedicated computer that supports high-performance network and raster graphics capabilities unavailable on most other Unix systems.

Diskless nodes

The Domain architecture makes it relatively easy to implement diskless nodes (nodes that can run without a local disk). Once a diskless node is successfully running the operating system, it has full access to the distributed file system. Thus, the only extra support needed for diskless nodes is a bootstrapping mechanism.

When a diskless node is first powered up, it runs software stored in nonvolatile memory. This software sends a broadcast packet to the diskless node bootstrap service. A node that is willing to act as a partner for diskless nodes keeps a data file that specifies which nodes it will bootstrap and runs a process that monitors both the data file and the bootstrap service. (Since partnership requires a significant resource commitment, not all nodes may wish to be partners; thus, a node must explicitly declare its availability for partnership.)

Partner nodes supply an operating system environment to the diskless nodes they boot. This environment includes a system paging file, hooks into the network name space, and a directory to store the names of node-specific objects.

Eventually, one or more servers reply to the diskless node's request, and a node is selected to be the diskless node's partner. A copy of the operating system image is sent to the diskless node, followed by the pieces of the system environment. Once bootstrapped, diskless nodes make requests for remote objects (which for diskless nodes include their environment objects) in the same way as nodes with disks.

Remote use of resources

The first implementation of the Domain virtual memory system made no distinction between the paging demands of local and remote processes. Consequently, local users constantly competed with remote users for use of their local physical memory. Remote users would steal pages from the local user's process working set; as a result, the local user's paging rate would skyrocket. The remote user, however, received little benefit, since only infrequently used pages existed in the local node's memory.

The system has been revised to allow a local user to restrict the number of physical memory pages available for remote use. Experiments have shown that limiting the size of the pool to a small fraction of the total available physical memory—for example, five percent—has dramatically reduced the working set pressure from remote users yet has left them with little or no performance impact.

Another aspect of resource competition—for the use of a disk—seems an inherent consequence of the ability to share objects on the disk. However, the ring controller hardware can be used to allow a node to refuse all requests for service and still remain able to receive replies to its own requests. Therefore, if users desire, they can use the network's facilities without making any of their own available.

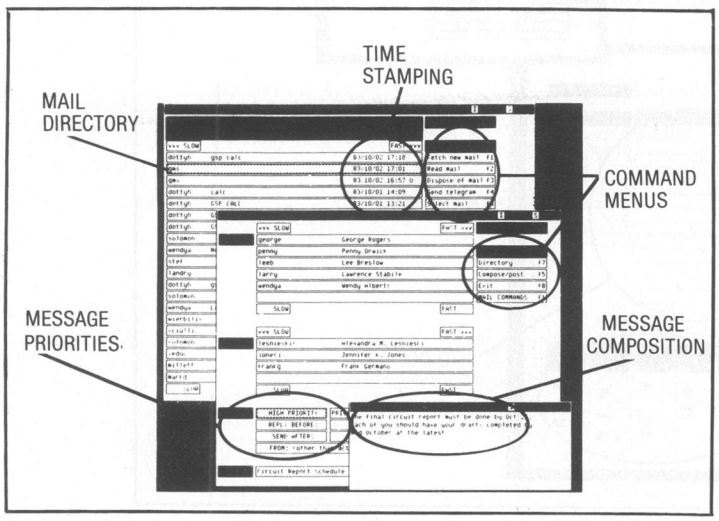


Figure 6. Mail program display.

Single-level store—some conclusions

There can be only two fundamentally different ways to perform computations in a distributed environment. When a computation and its data are not physically colocated, the computation can be moved to the data or the data can be moved to the computation. The first method corresponds to a message-passing model of distributed computing; the second method corresponds to a network-transparent data access model. The message-passing model leads to applications based on an interprocess communication facility, while the data access model leads to applications built on facilities such as Domain's single-level store.

Which model is better suited to the construction of distributed applications? The Domain architecture has pushed the network single-level store concept to the limit

and after almost three years of experience can provide some data for general observations on those limits.

Even though Domain's implementation specifically focuses on the efficiency of network paging, there are cases where IPC can be more efficient; the naming server is an example. Using the SLS technique, the naming server must send three messages to look up a name in a directory: one message to lock the directory, one to read it, and one to unlock it. Using the IPC model, the naming server has the potential to carry out the lookup sequence with one message.

Network-transparent access to data can also complicate the problem of object encoding and abstraction. For example, Domain's network registry uses a simple object-replication technique to store a small database used to identify network users for access control purposes. This database has a manager that abstracts from representational details; however, there is one manager per node, bringing about a network-wide knowledge of the database's internal structure.

As a result, the structure is difficult to modify, since any change involves synchronizing installation of the modified manager on each node. If the manager ran only on the nodes storing the registry, and if all other nodes used IPC to communicate with the manager, installing new versions of managers would be considerably simpler. This example shows that the IPC model is required to achieve effective information hiding in a distributed environment; data abstraction alone is insufficient.

Despite this example, there are many cases where an application's set of demands leads to the choice of SLS over IPC as an implementation tool. To make SLS preferable, the application must meet the following criteria:

(1) It requires a significant amount of data but simple synchronization, so that synchronization overhead is small.

(2) Its data structure semantics are very close to the representation, so that it is not necessary to confine knowledge of the representation to only a few nodes. Some examples are text files, which are often logically arrays of bytes, both conceptually and representationally, and executable program formats, which are defined for the most part by the CPU's instruction set.

(3) Its computation must not need to be trusted (for protection purposes) by other nodes. Trust cannot be achieved because, with current techniques, trust depends on the computation being moved to the data.

When these criteria hold, the limitations to single-level store described earlier do not apply; on the other hand, if the criteria do not hold, an IPC-based application is

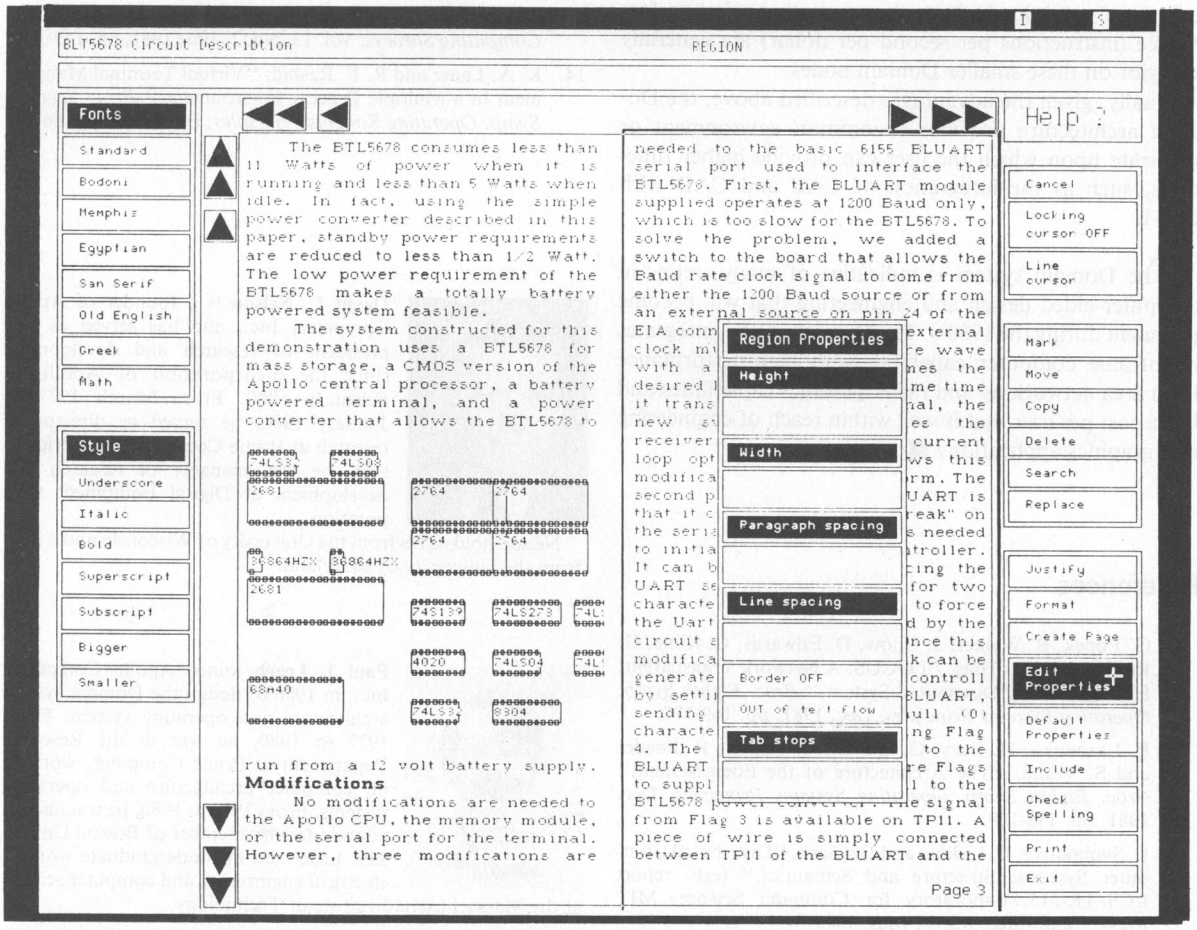


Figure 7. The document editor handles multicolumn, multifont, mixed text and graphics with justification, headers, footers, autowrap, and automatic line spacing capabilities.

needed. SLS is useful because a large number of applications do have the simpler requirements that SLS can satisfy, and because SLS provides the following two advantages: automatic, system-managed caching of frequently used programs and data; and a simpler program model whereby programs access data using ordinary string, record, and array access provided by the programming language instead of extra-lingual, message-passing methods.

User benefits

The style of computing suggested by the Domain system has numerous advantages compared with traditional forms of computing such as intelligent terminals linked to superminicomputer timesharing systems. First of all, each workstation is potentially a stand-alone system capable of solving large, complex problems. The entry cost of such a system is considerably lower, since no central components need to be purchased. The system can be expanded incrementally to a very large scale; each increment is relatively small, so that at any point the system optimally fits the client's needs.

In addition, users enjoy a predictable and constant level of performance that allows them to schedule their time without regard to computer availability. Although the performance level is high, gateway functions to large mainframe computers can be added to facilitate the optimum balance of work loads. Note, however, that price/performance (instructions per second per dollar) are generally superior on these smaller Domain nodes.

Finally, given the advantages described above, the Domain architecture provides a common environment or substrate upon which the user can provide higher functions, such as the integration of CAE and CAD with CAM.

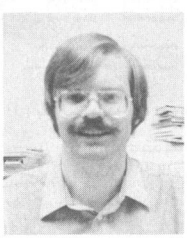
The Domain system is indicative of many trends in computer-aided design and engineering that will become prevalent during the 1980's. The Domain system integrates mainframe computer capability with high-performance local area networking and raster graphics capabilities and has a cost per user that is well within reach of engineering and graphics applications. ■

References

1. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proc. Eighth Symp. Operating Systems Principles,* Dec. 1981, pp. 169-177.
2. E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler and S. Vestal, "The Architecture of the Eden System," *Proc. Eighth Symp. Operating Systems Principles,* Dec. 1981, pp. 148-159.
3. L. Svobodova, B. Liskov, and D. Clark, "Distributed Computer Systems: Structure and Semantics," tech. report LCS/TR-215, Laboratory for Computer Science, MIT Press, Cambridge, Mass., Mar. 1979.
4. R. L. Gordon, W. Farr, and P. H. Levine, "Ringnet: A Packet Switched Local Network with Decentralized Control," *Computer Networks,* Vol. 3, North Holland, New York, 1980, pp. 373-379.
5. M. V. Wilkes and D. J. Wheeler, "The Cambridge Digital Communication Ring," *Proc. Local Area Comm. Network Symp.*, May 1979, pp. 47-61.
6. J. H. Saltzer, D. D. Clark, and K. T. Pogran, "Why a Ring," *Proc. Seventh Data Comm. Symp.*, Oct. 27-29, 1981, pp. 211-217.
7. J. H. Saltzer and K. T. Pogran, "A Star-Shaped Ring Network with High Maintainability," *Proc. Local Area Comm. Network Symp.,* The Mitre Corp., Bedford, Mass., May 1979, pp. 179-190.
8. P. J. Leach, B. L. Stumpf, J. A. Hamilton, and P. H. Levine, "UIDs as Internal Names in a Distributed File System," *Proc. First Symp. Principles of Distributed Computing*, Ottawa, Canada, Aug. 1982, pp. 34-41.
9. D. M. Ritchie and K. Thompson, "The UNIX Time-sharing System," *Comm. ACM,* Vol. 17, No. 7, July 1974, pp. 365-375.
10. P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf, "The Architecture of an Integrated Local Network," *IEEE Journal on Selected Areas in Comm.*, Nov. 1983, pp. 842-856.
11. E. I. Organick, "The Multics System: An Examination of Its Structure," MIT Press, Cambridge, Mass., 1972.
12. R. E. French, R. W. Collins, and L. W. Loen, "System/38 Machine Storage Management," *IBM System/38 Technical Developments,* IBM General Systems Division, 1978, pp. 63-66.
13. W. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *Computing Surveys,* Vol. 13, No. 2, June 1981, pp. 149-184.
14. K. A. Lantz and R. F. Rashid, "Virtual Terminal Management in a Multiple Process Environment," *Proc. Seventh Symp. Operating Systems Principles*, Dec. 1979, pp. 86-97.

David L. Nelson is a founder of Apollo Computer, Inc., and has served as vice president of research and development since the incorporation of Apollo in February 1980. From March 1977 to January 1980, he served as director of research at Prime Computer, Inc. Prior to that, he was manager of research and development at Digital Equipment Corporation.

Nelson holds a BS from the University of Wisconsin and a PhD from the University of Maryland.



Paul J. Leach joined Apollo Computer, Inc., in 1980 to design the Domain system architecture and operating system. From 1977 to 1980, he was in the Research Department at Prime Computer, working on computer architecture and operating systems. From 1979 to 1980, he was also an adjunct faculty member of Boston University. Leach did his undergraduate work in electrical engineering and computer science at the Massachusetts Institute of Technology.

The authors' address is Apollo Computer, Inc., 15 Elizabeth Drive, Chelmsford, MA 01824.