Digital Technical Journal

DIGITAL UNIX CLUSTERS

OBJECT MODIFICATION TOOLS

EXCURSION FOR WINDOWS OPERATING SYSTEMS

NETWORK DIRECTORY SERVICES

digital™

Volume 8 Number 1
1996

The *Digital Technical Journal* is a refereed journal published quarterly by Digital Equipment Corporation, 30 Porter Road LJO2/D10, Littleton, Massachusetts 01460. Subscriptions to the *Journal* are $40.00 (non-U.S. $60) for four issues and $75.00 (non-U.S. $115) for eight issues and must be prepaid in U.S. funds. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Orders, inquiries, and address changes should be sent to the *Digital Technical Journal* at the published-by address. Inquiries can also be sent electronically to dtj@digital.com. Single copies and back issues are available for $16.00 each by calling DECdirect at 1-800-DIGITAL (1-800-344-4825). Recent issues of the *Journal* are also available on the Internet at http://www.digital.com/info/dtj. Complete Digital Internet listings can be obtained by sending an electronic mail message to info@digital.com.

Digital employees may order subscriptions through Readers Choice by entering VTX PROFILE at the system prompt.

Comments on the content of any paper are welcomed and may be sent to the managing editor at the published-by or network address.

ISSN 0898-901X

Documentation Number EY-U025E-TJ

Book production was done by Quantic Communications, Inc.

Cover Design
The "hot" colors on our cover reflect the kind of performance delivered by 64-bit Digital UNIX TruCluster systems. A four-node cluster made up of AlphaServer 8400 5/350 systems interconnected with the high-speed MEMORY CHANNEL and running the Oracle Universal Server with Oracle Parallel Server recently achieved record TPC-C performance of 30,390 tpmC. The design of the Digital UNIX TruCluster system is the opening topic in this issue.

The cover was designed by Lucinda O'Neill of Digital's Design Group.

http://www.digital.com/info/dtj

Contents

Editor's Introduction

Digital recently announced record-breaking 30,390 tpmC performance on a Digital UNIX cluster of 64-bit RISC AlphaServer systems. In this issue, engineers from the UNIX team describe the key technologies that enable these near supercomputer performance levels as well as provide the cluster characteristics of high availability and scalability. Also presented in this issue are advanced UNIX programming tools for maximizing performance, X server software that supports the Microsoft family of operating systems, and new network directory services that simplify management.

First defined by Digital in the early 1980s, clusters are highly available, scalable multicomputer systems built with standard parts and offering the advantages of single-computer systems. Wayne Cardoza, Fred Glover, and Sandy Snaman compare clusters with other types of multicomputer configurations and describe the major components of Digital's newest cluster implementation, TruCluster systems, for the 64-bit UNIX environment. The cluster interconnect, called MEMORY CHANNEL, is critical to the cluster's outstanding performance. MEMORY CHANNEL implements clusterwide virtual shared memory and reduces overhead and latency by two to three orders of magnitude over conventional interconnects.

Also developed for the Digital UNIX environment (version 4.0) are two program analysis and optimization tools—OM and Atom. The tool technology originated in Digital's Western Research Laboratory, where researchers focused on providing performance diagnosis and improvements for large customer applications. Software developers Linda Wilson, Craig Neth, and Mike Rickabaugh from the UNIX Development Environment Group describe the object modification tools and the flexibility they provide over traditional tools that are implemented in the realm of compilers. In addition to demonstrating practical application of the tools, the authors examine the process of transferring technology from research to development.

For mixed operating system environments, Digital developed Windows-based X server software, called eXcursion, to allow the windows of a remote host running UNIX or OpenVMS to display on a desktop running the Microsoft Windows operating system. The latest version of eXcursion, described here by John Freitas, Jim Peterson, Scot Aurenz, Chuck Guldenschuh, and Paul Ranauro, is wholly rewritten to maximize graphics performance and to support the full range of Windows platforms: Windows, Windows 95, and Windows NT. This new version is based on the X Window System version 11, release 6 protocol from the X Consortium.

Two network directory services that reduce complexity and increase choices for network managers are the subjects of our next papers. The first is designed for multiple networked environments; Integrated Directory Services (IDS) software integrates multiple services into one directory-service-independent system. Margaret Olson, Laura Holly, and Colin Strutt outline the problems that have limited the use of directory services and the different design approaches the team considered to simplify directory services use and make it more attractive. They then describe the IDS extensible, object-based framework, which comprises an application programming interface and a service provider interface. Next, Rich Rosenbaum and Stan Goldfarb present the Common Directory Interface (CDI) for DECnet/OSI. Implemented as shared libraries in the Digital UNIX and OpenVMS operating systems, CDI is designed to give network managers a choice of directory services. The authors describe the libraries and the registration tool set of management operations that is layered on a specialized API.

Coming up in the *Journal* are papers about a new log-structured clusterwide file system called Spiralog, the 64-bit OpenVMS operating system, speech recognition software, and the UNIX clusters message-passing system and its use for program parallelization.

Jane C. Blake
Managing Editor

Foreword

Don Harbert
Vice President, UNIX Business

Digital not only invented clusters but continues to set the standard by which all other cluster systems are measured. The VAXcluster success and that of Digital's latest UNIX cluster systems derive from superb engineering that builds on the system definition put forth in the early 1980s by the VAX engineering team: an available, extensible, high-performance multicomputer system built from standard processors and a general-purpose operating system, with characteristics of both loosely and tightly coupled systems.*

We in the UNIX community are proud of our VAXcluster heritage and have engineered our products to provide the same kinds of benefits to customers that VAXcluster systems provide.† In the opening paper for this issue of the *Journal*, members of the Digital UNIX engineering team describe the multicomputer system for the Digital UNIX environment, called TruCluster, which, like the VAXcluster system, is designed for high availability, scalability, and performance.

The technology, of course, is different, and the environment is open. The fundamental concepts are nevertheless the same. The TruCluster system is a loosely coupled, general-purpose system connected by a high-performance interconnect. It maintains a single security domain and is managed as a single system.

Cluster services remain available even when other members are unavailable. Like VAXcluster systems, TruCluster systems implement a distributed lock manager, which provides synchronization for a highly parallelized distributed database system. The technology for the lock manager, however, is newly implemented for the UNIX environment. Also completely new is the interconnect technology for TruCluster systems. MEMORY CHANNEL is a reliable, high-speed interconnect based on a design by Digital partner Encore Computer Corporation. MEMORY CHANNEL addresses the unique needs of clusters by implementing clusterwide virtual shared memory; the interconnect reduces overhead and latency by two to three orders of magnitude.‡ Because MEMORY CHANNEL uses the industry-standard PCI, designers can implement the network at very low cost. We believe this interconnect technology puts Digital years ahead of the competition.

The TruCluster system is the latest example of Digital's intent to remain a technology leader in the UNIX market. We began by developing the first high-performance, 64-bit general-purpose operating system, DEC OSF/1, shipping in March 1993. The first Digital UNIX cluster release, DECsafe Available Server Environment, followed soon thereafter in April 1994. The announcement in April 1996 of TruCluster systems with MEMORY CHANNEL

* Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker, "VAXclusters: A Closely-Coupled Distributed System," *ACM Transactions on Computer Systems*, vol. 4, no. 2 (May 1986): 130–146.

† Digital has renamed VAXcluster systems to OpenVMS Cluster systems.

‡ Richard B. Gillett, "Memory Channel Network for PCI," *IEEE Micro* (February 1996): 12–18.

again places Digital far ahead of the competition technologically. The performance of these available cluster systems now approaches that of very expensive supercomputers. System performance has been measured at the record-breaking rate of 30,390 tpmC on four AlphaServer 8400 systems running Digital UNIX and the Oracle Universal Server with Oracle Parallel Server. The previous performance record, 20,918 tpmC, was held by the proprietary Tandem Himalaya K10000-112; Digital's open system cluster performance record is 1.5 times the Tandem performance record at one-third the system cost.

For Digital, clusters of high-performance 64-bit systems are to a great extent at the heart of its commercial and technical server strategy. Digital UNIX has been defined and engineered for the server business, specifically, for the high-performance commercial and large-problem/scientific environment. To be successful in the open system market, however, a company must reach outside itself to jointly engineer products with leading software suppliers that have the software customers need to be competitive. Therefore, the first TruCluster implementation is designed with Digital's partners—major software companies—to meet the requirements for high performance and functionality in the commercial database server market.

The competitive challenge now is to maintain Digital's significant lead in providing outstanding cluster performance, availability, and affordability. From a technological perspective, the immediate and achievable goal is to increase the number of cluster nodes from 4 to 10 or 20 nodes. Within this range, Digital maintains a simple cluster system model that offers the performance advantages of clustering and avoids the disadvantages, such as the management problems and qualification headaches, of more complex topologies. Further, the Digital UNIX organization will focus on a new cluster file system, configuration flexibility, management tools, and a cluster alias that allows a single-system view for clients and peers. The overall goal of this work is to evolve toward a more general computing environment.

The kinds of tools that both simplify and enhance performance are exemplified by the program analysis and optimization tools presented in this issue. Built on Digital UNIX version 4.0 and announced in April, these tools help software developers extract maximum performance from the system. The story of the tools development is an excellent example of the direct application of research to products. The power of the OM object modification tool and the analysis tool with object modification (Atom) was recognized by developers even as research progressed; in fact, semiconductor designers developed Atom tools to evaluate new Alpha chip implementations. The result of this close cooperation between research and development is advanced programming tools for customers.

These efforts in the UNIX organization are manifestations of Digital's commitment to open systems. Other areas of engineering where this commitment is apparent are also represented in this issue. For example, eXcursion software is key to integration between Microsoft's Windows family of products and Digital's UNIX and OpenVMS products. This wholly revised version both adds new functionality and conserves system resources. Another major area of strength for Digital is its networks products. Networks engineers describe two examples of network services that increase users' choices and extend system functionality, i.e., the Integrated Directory Services (IDS) and the Common Directory Interface.

Digital's strategy is to continue to engineer products that provide outstanding performance and price/performance in open environments. In all areas of engineering—systems, services, networking—our goal is to set the standard by which all others are measured.

Design of the TruCluster Multicomputer System for the Digital UNIX Environment

Wayne M. Cardoza
Frederick S. Glover
William E. Snaman, Jr.

The TruCluster product from Digital provides an available and scalable multicomputer system for the UNIX environment. Although it was designed for general-purpose computing, the first implementation is directed at the needs of large database applications. Services such as distributed locking, failover management, and remote storage access are layered on a high-speed cluster interconnect. The initial implementation uses the MEMORY CHANNEL, an extremely reliable, high-performance interconnect specially designed by Digital for the cluster system.

The primary goal for the first release of the TruCluster system for the Digital UNIX operating system was to develop a high-performance commercial database server environment running on a cluster of several nodes. Database applications often require computing power and I/O connectivity and bandwidth greater than that provided by most single systems. In addition, availability is a key requirement for enterprises that are dependent on database services for normal operations. These requirements led us to implement a cluster of computers that cooperate to provide services but fail independently. Thus, both performance and availability are addressed.

We chose an industry-standard benchmark to gauge our success in meeting performance goals. The Transaction Processing Performance Council TPC-C benchmark is a widely accepted measurement of the capability of large servers. Our goal was to achieve industry-leading numbers in excess of 30,000 transactions per minute (tpmC) with a four-node TruCluster system.

The TruCluster version 1.0 product provides reliable, shared access to large amounts of storage, distributed synchronization for applications, efficient cluster communication, and application failover. The focus on database servers does not mean that the TruCluster system is not suitable for other applications, but that the inevitable design decisions and trade-offs for the first product were made with this goal in mind. Although other aspects of providing a single-system view of a cluster are important, they are secondary objectives and will be phased into the product over time.

This paper begins with a brief comparison of computer systems and presents the advantages of clustered computing. Next, it introduces the TruCluster product and describes the design of its key software components and their relationship to database applications. The paper then discusses the design of the MEMORY CHANNEL interconnect for cluster systems, along with the design of the low-level software foundation for cluster synchronization and communication. Finally, it addresses application failover and hardware configurations.

Brief Comparison of Computing Systems

Contemporary computing systems evolved from centralized, single-node time-sharing systems into several distinct styles of multinode computer systems. Single-node systems provided uniform accessibility to resources and services and a single-management domain. They were limited with respect to scalability, however, and system failures usually resulted in a complete loss of service to clients of the system.

Multinode computer systems include symmetric multiprocessing (SMP) systems and massively parallel processors (MPPs). They also include network-based computing systems such as the Open Software Foundation Distributed Computing Environment (OSF DCE), Sun Microsystems Inc.'s Open Network Computing (ONC), and workstation farms.[1,2] Each of these systems addresses one or more of the benefits associated with clustered computing.

SMP configurations provide for tightly coupled, high-performance resource sharing. In their effective range, SMP systems provide the highest-performance single-system product for shared-resource applications. Outside that range, however, both hardware and software costs increase rapidly as more processors are added to an SMP system. In addition, SMP availability characteristics are more closely associated with those of single systems because an SMP system, by definition, is composed of multiple processors but not multiple memories or I/O subsystems.

MPP systems such as the Intel Paragon series were developed to support complex, high-performance parallel applications using systems designed with hundreds of processors. The individual processors of an MPP system were typically assigned to specific tasks, resulting in fairly special-purpose machines.

The DCE and ONC technologies provide support for common naming and access capabilities, user account management, authentication, and the replication of certain services for improved availability. Workstation farms such as the Watson Research Central Computer Cluster deliver support for the parallel execution of applications within multiple computer environments typically constructed using off-the-shelf software and hardware.[3] ONC, DCE, and farms provide their services and tools in support of heterogeneous, multivendor computing environments with hundreds of nodes. They are, however, much further away from realizing the benefits of a single-system view associated with clustered computing.

In the continuum of multinode computer systems, the advantage of the cluster system is its ability to provide the single-system view and ease of management associated with SMP systems and at the same time supply the failure isolation and scalability of distributed systems.

Cluster systems have clear advantages over large-scale parallel systems on one side and heterogeneous distributed systems on the other side. Cluster systems provide many cost and availability advantages over large parallel systems. They are built of standard building blocks with no unusual packaging or interconnect requirements. Their I/O bandwidth and storage connectivity scale well with standard components. They are inherently more tolerant of failures due to looser coupling. Parallel or multiprocessor systems should be thought of as cluster components, not as cluster replacements.

Cluster systems have a different set of advantages over distributed systems. First they are homogeneous in nature and more limited in size. Cluster systems can be more efficient when operating in more constrained environments. Data formats are known; there is a single-security domain; failure detection is certain; and topologies are constrained. Cluster systems also are likely to have interconnect performance advantages. Protocols are more specialized; interconnect characteristics are more uniform; and high performance can be guaranteed. Finally, the vendor-specific nature of cluster systems allows them to evolve faster than heterogeneous distributed systems and will probably always allow them to have advantages.

There are numerous examples of general-purpose clusters supplied by most computer vendors, including AT&T, Digital, Hewlett-Packard, International Business Machines Corporation, Sequent Computer Systems, Sun Microsystems, and Tandem Computers. Digital's OpenVMS cluster system is generally accepted as the most complete cluster product offering in the industry, and it achieves many of the single-system management attributes.[4] Much of the functionality of the OpenVMS cluster system is retained in Digital's TruCluster product offerings.

Structure of the TruCluster System

Digital's TruCluster multicomputer system is a highly available and scalable structure of UNIX servers that preserves many of the benefits of a centralized, single computer system. The TruCluster product is a collection of loosely coupled, general-purpose computer systems connected by a high-performance interconnect. It maintains a single security domain and is managed as a single system. Each cluster node may be a uniprocessor or a multiprocessor system executing the Digital UNIX operating system. Figure 1 shows a typical cluster configuration.

Each cluster member is isolated from software and hardware faults occurring on other cluster members. Thus, the TruCluster system does not have the tightly coupled, "fail together" characteristics of multiprocessor systems. Cluster services remain available even when individual cluster members are temporarily

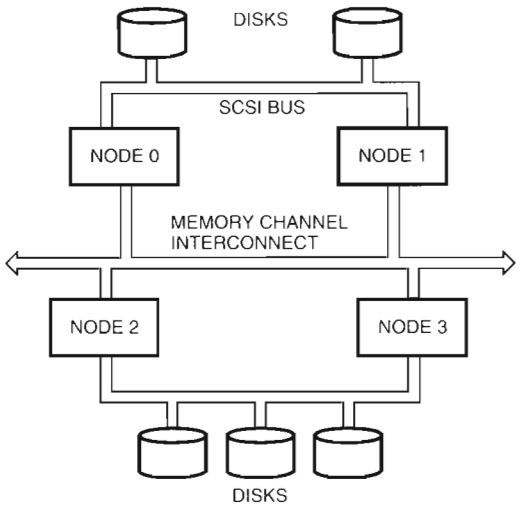


Figure 1
Configuration of a Four-node Cluster System

unavailable. Other important availability objectives of the TruCluster server include quick detection of component and member failures, on-line reconfigurations to accommodate the loss of a failed component, and continued service while safe operation is possible.

The TruCluster product supports large, highly available database systems through several of its key components. First, the distributed remote disk (DRD) facility provides reliable, transparent remote access to all cluster storage from any cluster node. Next, the distributed lock manager (DLM) enables the elements of a distributed database system to synchronize activity on independent cluster nodes. Finally, elements of Digital's DECsafe Available Server Environment (ASE) provide application failover.[5] In support of all these components is the connection manager, which controls cluster membership and the transition of nodes in and out of the cluster. Figure 2 is a block diagram showing the relationships between components.

Each major component is described in the remainder of this paper. In addition, we describe the high-performance MEMORY CHANNEL interconnect that was designed specifically for the needs of cluster systems.

Distributed Remote Disk Subsystem

The distributed remote disk (DRD) subsystem was developed to support database applications by presenting a clusterwide view of disks accessed through the character or raw device interface. The Oracle Parallel Server (OPS), which is a parallelized version of the Oracle database technology, uses the DRD subsystem.

The DRD subsystem provides a clusterwide namespace and access mechanism for both physical and logical (logical storage manager or LSM) volumes. The LSM logical device may be a concatenated, a striped, or a mirrored volume. DRD devices are accessible from any cluster member using the DRD device name. This location independence allows database software to treat storage as a uniformly accessible cluster resource and to easily load balance or fail over activity between cluster nodes.

Cluster Storage Background

Disk devices on UNIX systems are commonly accessed through the UNIX file system and an associated block device special file. A disk device may also be accessed through a character device special file or raw device that provides a direct, unstructured interface to the device and bypasses the block buffer cache.

Database management systems and some other high-performance UNIX applications are often designed to take advantage of the character device special file interfaces to improve performance by avoiding additional code path length associated with the file system cache.[6,7] The I/O profile of these systems is characterized by large files, random access to records, private data caches, and concurrent read-write sharing.

Overall Design of the DRD

The DRD subsystem consists of four primary components. The remote raw disk (RRD) pseudo-driver redirects DRD access requests to the cluster member serving the storage device. The server is identified by information maintained in the DRD device database (RRDB). Requests to access local DRD devices are passed through to local device drivers. The block shipping client (BSC) sends requests for access to remote DRD devices to the appropriate DRD server and returns responses to the caller. The block shipping server (BSS) accepts requests from BSC clients, passes them to its local driver for service, and returns the results to the calling BSC client. Figure 3 shows the components of the DRD subsystem.

The DRD management component supports DRD device naming, device creation and deletion, device relocation, and device status requests. During the DRD device creation process, the special device file designating the DRD device is created on each cluster member. In addition, the DRD device number, its corresponding physical device number, the network address of the serving cluster member, and other configuration parameters are passed to the DRD driver, which updates its local database and communicates the information to other cluster members. The DRD driver may be queried for device status and DRD database information.

Clusterwide Disk Access Model

During the design of the DRD subsystem, we considered both shared (multiported) and served disk models. A multiported disk configuration provides good failure recovery and load balancing characteristics. On the

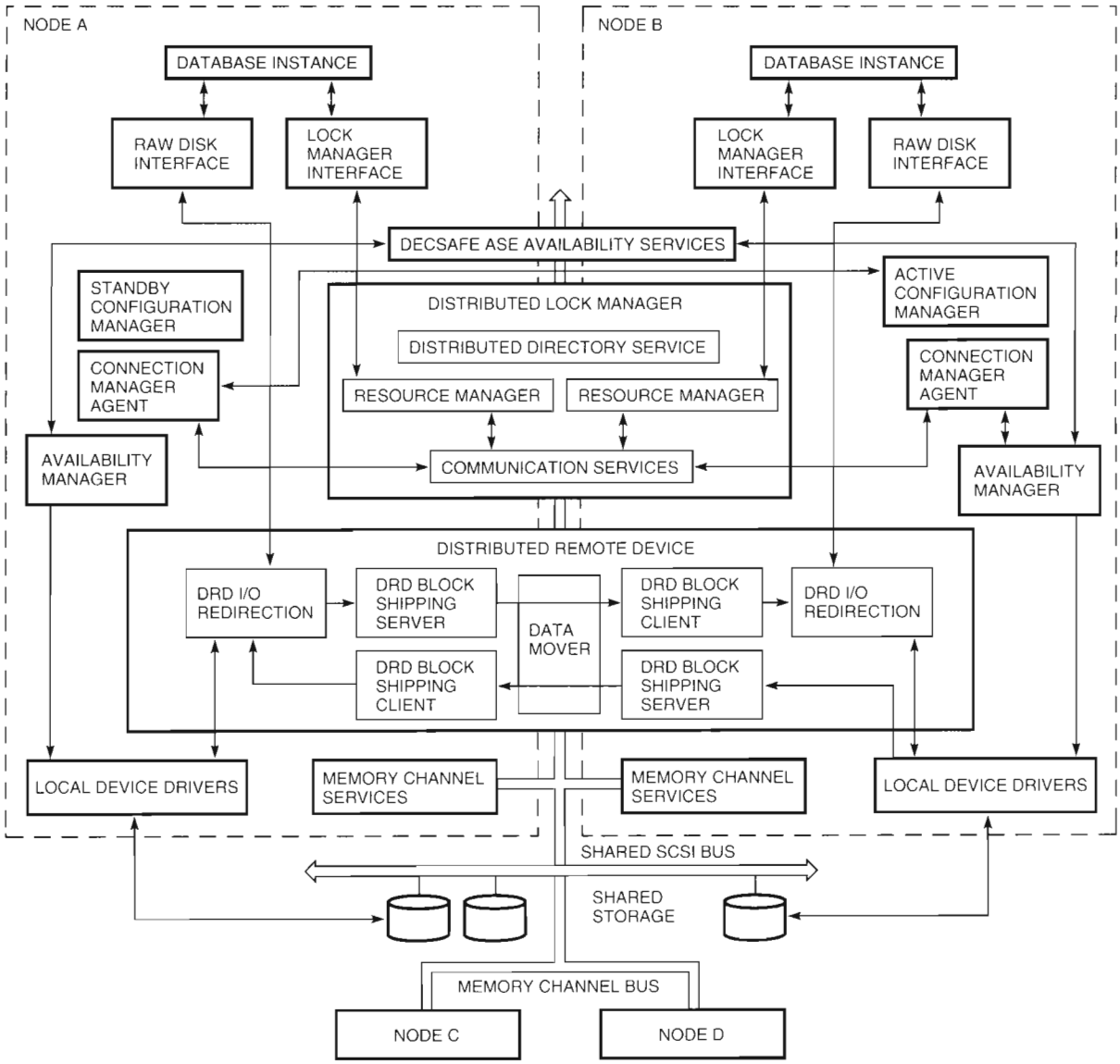


Figure 2
Software Components

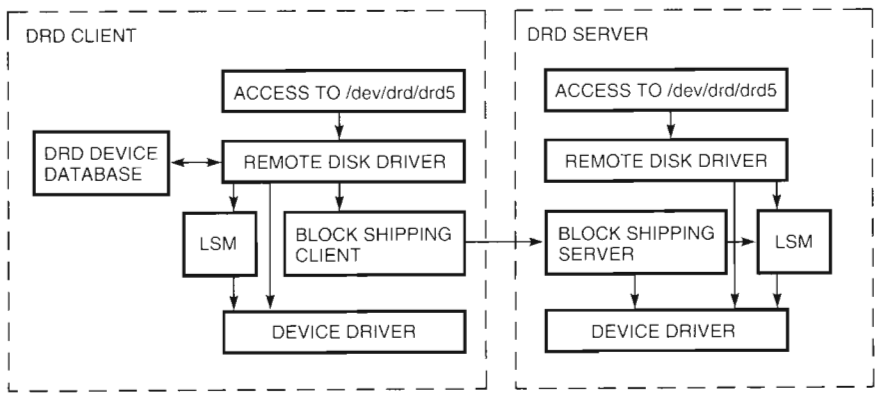


Figure 3
Distributed Remote Disk Subsystem

other hand, I/O bus contention and hardware queuing delays from fully connected, shared disk configurations can limit scalability. In addition, present standard I/O bus technologies limit configuration distances.[8] As a consequence, we selected a served disk model for the DRD implementation. With this model, software queuing alleviates the bus contention and bus queuing delays. This approach provides improved scalability and fault isolation as well as flexible storage configurations.[9,10] Full connectivity is not required, and extended machine room cluster configurations can be constructed using standard networks and I/O buses.

The DRD implementation supports clusterwide access to DRD devices using a software-based emulation of a fully connected disk configuration. Each device is assigned to a single cluster member at a time. The member registers the device into the clusterwide namespace and serves the device data to other cluster members. Failure recovery and load-balancing support are included with the DRD device implementation. The failure of a node or controller is transparently masked when another node connected to the shared bus takes over serving the disk. As an option, automatic load balancing can move service of the disk to the node generating the most requests.

In the TruCluster version 1.0 product, data is transferred between requesting and serving cluster members using the high-bandwidth, low-latency MEMORY CHANNEL interconnect, which also supports direct memory access (DMA) between the I/O adapter of the serving node and the main memory of the requesting node. The overall cluster design, however, is not dependent on the MEMORY CHANNEL interconnect, and alternative cluster interconnects will be supported in future software releases.

DRD Naming

The Digital UNIX operating system presently supports character device special file names for both physical disk devices and LSM logical volumes and maintains a separate device namespace for each. An important DRD design objective was to develop a clusterwide naming scheme integrating the physical and logical devices within the DRD namespace. We considered defining a new, single namespace to support all cluster disk devices. Our research, however, revealed plans to introduce significant changes into the physical device naming scheme in a future base system release and the complications of licensing the logical disk technology from a third party that maintains control over the logical volume namespace. These issues resulted in deferring a true clusterwide device namespace.

As an interim approach, we chose to create a separate, clusterwide DRD device namespace layered on the existing physical and logical device naming schemes. Translations from DRD device names into the underlying physical and logical devices are maintained by the DRD device mapping database on each cluster node. DRD device "services" are created by the cluster administrator using the service registration facility.[11] Each "add Service" management operation generates a unique service number that is used in constructing the DRD device special file name. This operation also creates the new DRD device special file on each cluster member. A traditional UNIX-device-naming convention results in the creation of DRD special device file names in the form of /dev/drd/drd{service number}.[12]

DRD Relocation and Failover

ASE failover (see the discussion in the section Application Failover) is used to support DRD failover and is fully integrated within the cluster product. The device relocation policy defined during the creation of a DRD device indicates whether the device may be reassigned to another cluster member as a result of a node or controller failure or a load-balancing operation. In the event of a cluster member failure, DRD devices exported by the failed member are reassigned to an alternate server attached to the same shared I/O bus. During reassignment, the DRD device databases are updated on all cluster members and DRD I/O operations are resumed. Cluster device services may also be reassigned during a planned relocation, such as for load balancing or member removal. Any DRD operation in progress during a relocation triggered by a failure will be retried based upon the registered DRD retry policy. The retry mechanism must revalidate the database translation map for the target DRD device because the server binding may have been modified. Failover is thus transparent to database applications and allows them to ignore configuration changes.

Several challenges result from the support of multiported disk configurations under various failure scenarios. One of the more difficult problems is distinguishing a failed member from a busy member or a communication fault. The ASE failover mechanism was designed to maintain data integrity during service failover, and to ensure that subsequent disk operations are not honored from a member that has been declared "down" by the remaining cluster members. This ASE mechanism, which makes use of small computer systems interface (SCSI) target mode and device reservation, was integrated into the TruCluster version 1.0 product and supports the DRD service guarantees.

Other challenges relate to preserving serialization guarantees in the case of cluster member failure. Consider a parallel application that uses locks to serialize access to shared DRD devices. Suppose the application is holding a write lock for a given data block and

issues an update for that block. Before the update operation is acknowledged, however, the local member fails. The distributed lock manager, which will have been notified of the member failure, then takes action to release the lock. A second cooperating application executing on another cluster member now acquires the write lock for that same data block and issues an update for that block. If the failure had not occurred, the second application would have had to wait to acquire a write lock for the data block until the first application released the lock, presumably after its write request had completed. This same serialization must be maintained during failure conditions. Thus, it is imperative that the write issued by the first (now failed) application partner not be applied after the write issued by the second application, even in the presence of a timing or network retransmission anomaly that delays this first write.

To avoid the reordering scenario just described, we employed a solution called a sequence barrier in which the connection manager increments a sequence number each time it completes a recovery transition that results in released locks. The sequence number is communicated to each DRD server, which uses the sequence number as a barrier to prevent applying stale writes. This is similar to the immediate command feature of the Mass Storage Control Protocol (MSCP) used by OpenVMS cluster systems to provide similar guarantees. Note that no application changes are required.

As another example, client retransmissions of DRD protocol requests that are not idempotent can cause serious consistency problems. Request transaction IDs and DRD server duplicate transaction caches are employed to avoid undesirable effects of client-generated retransmissions.[13]

Cluster member failures are mostly transparent to applications executing on client member systems. Nondistributed applications may fail, but they can be automatically restarted by ASE facilities. DRD devices exported by a serving member become unavailable for a small amount of time when the member fails. Cluster failover activities that must occur before the DRD service is again available include detecting and verifying the member failure, purging the disk device SCSI hardware reservation, assigning an alternate server, establishing the new reservation, and bringing the device back on-line. A database application serving data from the DRD device at the time of the failure may also have registered to have a restart script with a recovery phase executed prior to the restart of the database application. A possible lack of transparency may result if some client applications are not designed to accommodate this period of inaccessible DRD service. The DRD retry request policy is configurable to accommodate applications interacting directly with a DRD device.

Distributed Lock Manager

The distributed lock manager (DLM) provides synchronization services appropriate for a highly parallelized distributed database system. Databases can use locks to control access to distributed copies of data buffers (caches) or to limit concurrent access to shared disk devices such as those provided by the DRD subsystem. Locks can also be used for controlling application instance start-up and for detecting application instance failures. In addition, applications can use the locking services for their other synchronization needs.

Even though this is a completely new implementation, the lock manager borrows from the original design and concepts introduced in 1984 with the VAXcluster distributed lock manager.[14] These concepts were used in several recent lock manager implementations for UNIX by other vendors. In addition, the Oracle Parallel Server uses a locking application programming interface (API) that is conceptually similar to that offered here.

Usage of the DLM

The lock manager provides an API for requesting, releasing, and altering locks.[15,16] These locks are requested on abstract names chosen by the application. The names represent resources and may be organized in a hierarchy. When a process requests a lock on a resource, that request is either granted or denied based on examination of locks already granted on the resource. Cooperating components of an application use this service to achieve mutually exclusive resource usage. In addition, a mode associated with each lock request allows traditional levels of sharing such as multiple readers excluding all writers.

The API provides optional asynchronous request completion to allow queuing requests or overlapping multiple operations for increased performance. Queuing prevents retry delays, eliminates polling overhead, and provides a first in, first out (FIFO) fairness mechanism. In addition, asynchronous requests can be used as the basis of a signaling mechanism to detect component failures in a distributed system. One component acquires an exclusive lock on a named resource. Other components queue incompatible requests with asynchronous completion specified. If the lock holder fails or otherwise releases its lock, the waiting requests are granted. This usage is sometimes referred to as a "dead man" lock.[17]

A process can request notification when a lock it holds is blocking another request. This allows elimination of many lock calls by effectively caching locks. When resource contention is low, a lock is acquired and held until another process is blocked by that lock. Upon receiving blocking notification, the lock can be released. When resource contention is high, the lock is acquired and released immediately. In addition, this

notification mechanism can be used as the basis of a general signaling mechanism. One component of the application acquires an exclusive lock on a named resource with blocking notification specified. Other components then acquire incompatible locks on that resource, thus triggering the blocking notification. This usage is known as a "doorbell" lock.[17]

The DLM is often used to coordinate access to resources such as a distributed cache of database blocks. Multiple copies of the data are held under compatible locks to permit read but not write access. When a writer wants an incompatible lock, readers are notified to downgrade their locks and the writer is granted the lock. The writer modifies the data before downgrading its lock. The reader's lock requests are again granted, and the reader fetches the latest copy of the data. A value block can also be associated with each resource. Its value is obtained when a lock is granted and can be changed when certain locks are released. The value block can be used to communicate any useful information, including the latest version number of cached data protected by the resource.

Design Goals of the DLM

The overall design goal of the lock manager was to provide services for highly scalable database systems. Thus correctness, robustness, scaling, and speed were the overriding subgoals of the project.

Careful attention to design details, rigorous testing, internal consistency checking, and years of experience working with the VMS distributed lock manager have all contributed to ensuring the correctness of the implementation for the Digital UNIX system. Because the lock manager provides guarantees about the state of all locks when either a lock holder or the node upon which it is running fails, it can ensure the internal lock state is consistent as far as surviving lock holders are concerned. This robustness permits the design of applications that can continue operation when a cluster node fails or is removed for scheduled service. The choice of a kernel-based service and the use of a message protocol also contribute to robustness as discussed below.

In terms of performance and scaling, the lock manager is designed for minimal overhead to its users. The kernel-based service design provides high performance by eliminating the context switch overhead associated with server daemons. The lock manager uses the kernel-locking features of the Digital UNIX operating system for good scaling on SMP systems. A kernel-based service as opposed to a library also allows the lock manager to make strong guarantees about the internal consistency state of locks when a lock-holding process fails.

The message protocol contributes to cluster scaling and performance through a scaling property that maintains a constant cost as nodes are added to the cluster.[14] The message protocol also provides sufficiently loose coupling to allow the lock manager to maintain internal lock state when a node fails. The use of messages controls the amount of internal state visible to other nodes and provides natural checkpoints, which limit the damage resulting from the failure of a cluster node.

DLM Communication Services

The DLM session service is a communication layer that takes advantage of MEMORY CHANNEL features such as guaranteed ordering, low error rate, and low latency. These features allow the protocol to be very simple with an associated reduction in CPU overhead. The service provides connection establishment, delivery and order guarantees, and buffer management. The connection manager uses the communication service to establish a channel for the lock manager. The lock manager uses the communication services to communicate between nodes. Because the service hides the details of the communication mechanism, alternative interconnects can be used without changes to the lock manager's core routines.

The use of the MEMORY CHANNEL interconnect provides a very low latency communication path for small messages. This is ideal for the lock manager since lock messages tend to be very small and the users of the lock manager are sensitive to latency since they wait for the lock to be granted before proceeding. Small messages are sent by simply writing them into the receiving node's memory space. No other communication setup needs to be performed. Many network adapters and communication protocols are biased toward providing high throughput only when relatively large packets are used. This means that the performance drops off as the packet size decreases. Thus, the MEMORY CHANNEL interconnect provides a better alternative for communicating small, latency-sensitive packets.

Connection Manager

The connection manager defines an operating environment for the lock manager. The design allows generalization to other clients; but in the TruCluster version 1.0 product, the lock manager is the only consumer of the connection manager services. The environment hides the details of dynamically changing configurations. From the perspective of the lock manager, the connection manager manages the addition and removal of nodes and maintains a communication path between each node. These services allowed us to simplify the lock manager design.

The connection manager treats each node as a member of a set of cooperating distributed components. It maintains the consistency of the set by admitting and removing members under controlled conditions.

The connection manager provides configuration-related event notification and other support services to each member of a set. It provides notification when members are added and removed. It also maintains a list of current members. The connection manager also provides notification to clients when unsafe operation is possible as a result of partitioning. Partitioning exists when a member of a set is unaware of the existence of a disjoint set of similar clients.

The connection manager can be extended in client-specific ways to facilitate handling of membership change events. Extensions are integral, well-synchronized parts of the membership change mechanism. The lock manager uses an extension to distribute a globally consistent directory database and to coordinate lock database rebuilds.

The connection manager maintains a fully connected web of communication channels between members of the set. Membership in the set is contingent upon being able to communicate with all other members of that set. The use of the communication channels is entirely under the control of the lock manager or any other client that may use the connection manager in the future. When a client requests admission to a set, the connection manager establishes a communication channel between the new client and all existing clients. It monitors these connections to ensure they remain functional. A connection fails when a communication channel is unusable between a pair of clients or when a client at either end of the channel fails. The connection manager detects these conditions and reconfigures the set to contain only fully connected members.

The combination of a highly available communication channel, together with set membership and synchronized membership change responses, allows optimizations in the lock manager's message protocol. The lock manager can send a message to another node and know that either the message will be delivered or that the configuration will be altered so that it does not matter.

The use of the connection manager greatly simplifies the design and implementation of the lock manager. The connection manager allows most of the logic for handling configuration changes and communication errors to be moved away from main code paths. This increases mainline performance and simplifies the logic, allowing more emphasis on correct and efficient operation.

Memory Channel Interconnect

Cluster performance is critically dependent on the cluster interconnect. This is due both to the high-bandwidth requirements of bulk data transport for DRD and to the low latency required for DLM operations. Although the cluster architecture allows for any high-speed interconnect, the initial implementation supports only the new MEMORY CHANNEL interconnect designed specifically for the needs of cluster systems. This very reliable, high-speed interconnect is based on a previous interconnect designed by Encore Computer Corporation.[18] It has been significantly enhanced by Digital to improve data integrity and provide for higher performance in the future.

Each cluster node has a MEMORY CHANNEL interface card that connects to a hub. The hub can be thought of as a switch that provides either broadcast or point-to-point connections between nodes. It also provides ordering guarantees and does a portion of the error detection. The current implementation is an eight-node hub, but larger hubs are planned.

The MEMORY CHANNEL interconnect provides a 100-megabyte-per-second, memory-mapped connection to other cluster members. As shown in Figure 4, cluster members may map transfers from the MEMORY CHANNEL interconnect directly into their memory. The effect is of a write-only window into the memory of other cluster systems. Transfers are done with standard memory access instructions rather than special I/O instructions or device access

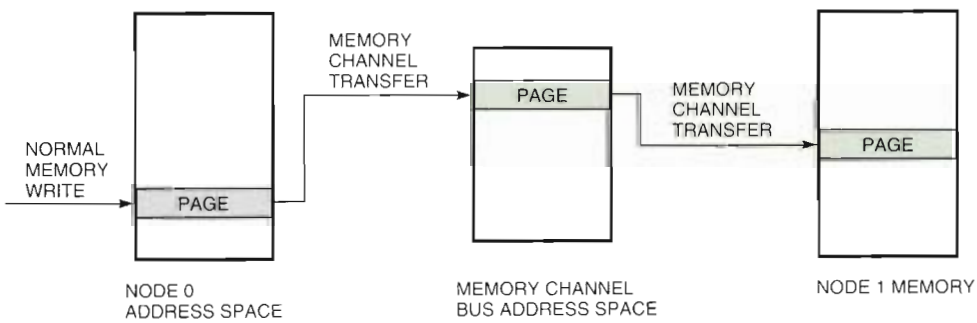


Figure 4
Transfers Performed by the MEMORY CHANNEL Interconnect

protocols to avoid the overhead usually present with these techniques. The use of memory store instructions results in extremely low latency (two microseconds) and low overhead for a transfer of any length.

The MEMORY CHANNEL interconnect guarantees essentially no undetected errors (approximately the same undetected error rate as CPUs or memory), allowing the elimination of checksums and other mechanisms that detect software errors. The detected error rate is also extremely low (on the order of one error per year per connection). Since recovery code executes very infrequently, we are assured that relatively simple, brute-force recovery from software errors is adequate. Using hardware error insertion, we have tested recovery code at error rates of many per second. Thus we are confident there are no problems at the actual rates.

Low-level MEMORY CHANNEL Software

Low-level software interfaces are provided to insulate the next layer of software (e.g., lock manager and distributed disks) from the details of the MEMORY CHANNEL implementation. We have taken the approach of providing a very thin layer to impact performance as little as possible and allow direct use of the MEMORY CHANNEL interconnect. Higher-level software then isolates its use of MEMORY CHANNEL in a transport layer that can later be modified for additional cluster interconnects.

The write-only nature of the MEMORY CHANNEL interconnect leads to some challenges in designing and implementing software. The only way to see a copy of data written to the MEMORY CHANNEL interconnect is to map MEMORY CHANNEL transfers to another region of memory on the same node. This leads to two very visible programming constraints. First, data is read and written from different addresses. This is not a natural programming style, and code must be written to treat a location as two variables, one for read and one for write. Second, the effect of a write is delayed by the transfer latency. At two microseconds, this is short but is enough time to execute hundreds of instructions. Hardware features are provided to stall until data has been looped back, but very careful design is necessary to minimize these stalls and place them correctly. We have had several subtle problems when an algorithm did not include a stall and proceeded to read stale data that was soon overwritten by data in transit. Finding these problems is especially difficult because much evidence is gone by the time the problem is observed. For example, consider a linked list that is implemented in a region of memory mapped to all cluster nodes through the MEMORY CHANNEL interconnect. If two elements are inserted on the list without inserting proper waits for the loopback delay, the effect of the first insert will not be visible when the second insert is done. This results in corrupting the list.

The difficulties just described are most obvious when dealing with distributed shared memory. Low-level software intended to support applications is instead oriented toward a message-passing model. This is especially apparent in the features provided for error detection. The primary mechanisms allow either the receiving or the sending node to check for any errors over a bounded period of time. This error check requires a special hardware transaction with each node and involves a loopback delay. If an error occurs, the sender must retransmit all messages and the receiver must not use any data received in that time. This mechanism works well with the expected error rates. However, a shared memory model makes it extremely difficult to bound the data affected by an error, unless each modification of a data element is separately checked for errors. Since this involves a loopback delay, many of the perceived efficiencies of shared memory may disappear. This is not to say that a shared memory model cannot be used. It is just that error detection and control of concurrent access must be well-integrated, and node failures require careful recovery. In addition, the write-only nature of MEMORY CHANNEL mappings is more suited to message passing than shared memory due to the extremely careful programming necessary to handle delayed loopback at a separate address.

APIs are provided primarily to manage resources, control memory mappings, and provide synchronization. MEMORY CHANNEL APIs perform the following tasks:

- Allocation and mapping
  - Allocate or deallocate the MEMORY CHANNEL address space.
  - Map the MEMORY CHANNEL interconnect for receive or transmit.
  - Unmap the MEMORY CHANNEL interconnect.
- Spinlock synchronization
  - Create and delete spinlock regions.
  - Acquire and release spinlocks.
- Other synchronization
  - Create and delete write acknowledgment regions.
  - Request write acknowledgment.
  - Create and delete software notification channels.
  - Send notification.
  - Wait for notification.
- Error detection and recovery
  - Get current error count.
  - Check for errors.
  - Register for callback on error.

Higher layers of software are responsible for transferring data, checking for errors, retrying transfers, and synchronizing their use of MEMORY CHANNEL address space after it is allocated.

Synchronization

Efficient synchronization mechanisms are essential for high-performance protocols over a cluster interconnect. MEMORY CHANNEL hardware provides two important synchronization mechanisms: first, an ordering guarantee that all writes are seen in the same order on all nodes, including the looped-back write on the originating node; second, an acknowledgment request that returns the current error state of all other nodes. Once the acknowledgment operation is complete, all previous writes are guaranteed either to have been received by other nodes or reported as a transmit or receive error on some node. We have implemented clusterwide software spinlocks based on these guarantees. Spinlocks are used for many purposes, including internode synchronization of other components and concurrency control for the clusterwide shared-memory data structures used by the low-level MEMORY CHANNEL software.

A spinlock is structured as an array with one element for each node. To acquire the spinlock, a node first bids for it by writing a value to the node's array element. A node wins by seeing its bid looped back by the MEMORY CHANNEL interconnect without seeing a bid from any other node. The ordering guarantees of the MEMORY CHANNEL ensure that no other node could have concurrently bid and believed it had won. Multiple nodes can realize they have lost, but more than one node cannot win. In case of a conflict, many different back-off techniques can be used. The winning node then changes its bid value to an own value. This last step is not necessary for correctness, but it does help with resolving contention and with various failure recovery algorithms. All higher-level synchronization is built on combinations of spinlocks, ordering guarantees, and error acknowledgments.

Error Recovery and Node Failures

Most of the difficult problems in the low-level software relate to error recovery and node failures. In spite of its reliability, errors will occur in the MEMORY CHANNEL interconnect, and they must be handled as transparently as possible. Transparency is key to simplifying the communication model seen by higher-level software. In addition, node failures from hardware or software faults are more frequent than MEMORY CHANNEL errors and must be dealt with even in the most inconvenient portions of the low-level code. The MEMORY CHANNEL interconnect is managed through a collection of distributed data structures that must be kept consistent. Software locks are used to synchronize access to these structures, but errors may leave them in an inconsistent state. Guaranteed error detection before the release of a lock allows operations to be redone in case of an error. Thus, all sequences of MEMORY CHANNEL writes must be idempotent to take advantage of this straightforward error-recovery technique.

If a node failure occurs, a surviving node must make all data structures consistent before it releases locks held by the failed node. To keep this a manageable task, we have written carefully structured algorithms to handle each inconsistent state. In general, structures are changed such that a single atomic write commits a change. If a node fails before this last write, no recovery is necessary. As an example, consider a data structure that is completely initialized before being added to a list. A single write is used to accomplish the list addition. If a node fails, the last write was either done or not and, in either case, the list is consistent. Complications arise when another node has a receive error on the last write done by a failing node. In this case, the failed node cannot retry after detecting the error, so the node with the receive error has a different view of the list than all other surviving nodes. To resolve this event, one node must propagate its view of the list to all other nodes before it releases the lock held by the failed node. Any node can do this because each has a self-consistent view of the list. If the node with the receive error propagates its view, the last element added by the failed node is lost. This situation is no different, however, from having the node fail a few instructions earlier. The challenge is to design recovery for all these cases and maintain our sanity by minimizing the number of such cases.

Another interesting problem is maintaining a consistent count of errors across all nodes. This count is key to the error protocols of both the low-level MEMORY CHANNEL software and higher layers since comparisons of a saved and a current value bound the period over which data is suspect. The count may be read on one node, transferred with a message, and compared to a current value on another node. Thus, a consistent value on all nodes is critical and must be maintained in the presence of arbitrary combinations of receive and transmit errors. (Although errors are very infrequent, they may be correlated; so algorithms must work well for error bursts.) The write acknowledgment, described earlier, guarantees that other nodes have received a write without error. It is used both to implement a lock protecting the error count and to guarantee that all nodes have seen an updated count. Updating the count is a slow operation due to multiple round-trip delays and long error time-outs, but it is performed very infrequently.

Future Enhancements to MEMORY CHANNEL Software

Fully supported MEMORY CHANNEL APIs are currently available only to other layers in the UNIX kernel for two important reasons: First, MEMORY CHANNEL is a new type of interconnect and we want to better understand its uses and advantages before committing to a fully functional API for general use. Second, many difficult issues of security and resource limits will affect the final interface. To help Digital and its customers gain the necessary experience, a limited functionality version of a user-level MEMORY CHANNEL API has been implemented in the version 1.0 product. This interface supports allocation and mapping of MEMORY CHANNEL space along with spinlock synchronization. It is oriented toward support of parallel computation in a cluster, but we also expect it will serve the needs of many commercial applications. Once we have a better understanding of how high-level applications will use the MEMORY CHANNEL interconnect, we will extend the design and provide additional APIs oriented toward both commercial applications and technical computing.

Application Failover

Digital's TruCluster multicomputer system is a logical evolution of the DECsafe Available Server Environment (ASE). An ASE system is a multinode configuration with all nodes and all highly available storage connected to shared SCSI storage buses. Figure 5 shows an ASE configuration. Software on each node monitors the status of all nodes and of shared storage. In case of a failure, the storage and associated applications are failed over to surviving systems. Planned application failover is accomplished by stopping the application on one node and restarting the application on a surviving node with access to any storage associated with the application. Application-specific scripts control failover and usually do not require application changes.

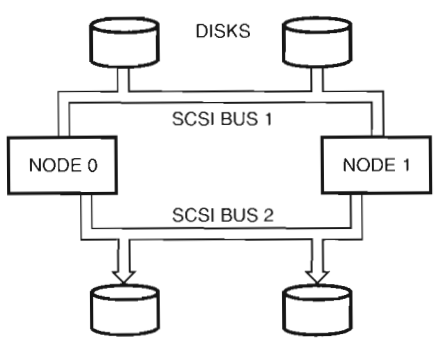


Figure 5
Typical ASE Configuration

In addition to supporting the application failover mechanisms from ASE, the TruCluster system supports parallel applications running on multiple cluster nodes. In case of a failure, the application is not stopped and restarted. Instead, it may continue to execute and transparently retain access to storage through a distributed disk server. In addition, more general hardware topologies are supported.

Hardware Configurations

The TruCluster version 1.0 product supports a maximum of four nodes connected by a high-speed MEMORY CHANNEL interconnect. The nodes may be any Digital UNIX system with a peripheral component interconnect (PCI) that supports storage and the MEMORY CHANNEL interconnect. Highly available storage is on shared SCSI buses connected to at least two nodes. Thus, a cluster looks like multiple ASE systems joined by a cluster interconnect.

Although the limitation to four nodes is temporary, we do not intend to support large numbers of nodes. Ten to twenty nodes on a high-speed interconnect is a reasonable target. A cluster is a component of a distributed system, not a replacement for one. If very large numbers of nodes are desired, a distributed system is built with cluster nodes as servers and other nodes as clients. This allows maintaining a simple model of a cluster system without having to allow for many complex topologies. Aside from simplicity, there are performance advantages from targeting algorithms for relatively small and simple cluster systems. Although the number of nodes is intended to be small, the individual nodes can be high-end multiprocessor systems. Thus, the overall computing power and the I/O bandwidth of a cluster are extremely large.

Conclusions

With the completion of the first release of Digital's TruCluster product, we believe we have met our goal of providing an environment for high-performance commercial database servers. Both the distributed lock manager and the remote disk services are meeting expectations and providing reliable, high-performance services for parallelized applications. The MEMORY CHANNEL interconnect is proving to be an excellent cluster interconnect: Its synchronization and failure detection are especially compatible with many cluster-aware components, which are enhanced by its low latencies and simplified by its elimination of complex error handling. The error rates have also proven to be as predicted. With over 100 units in use over the last year, we have observed only a very small number of errors other than those attributable to debugging new versions of the hardware.

Detailed component performance measurements are still in progress, but rough comparisons of DRD against local I/O have shown no significant penalty in latency or throughput. There is of course additional CPU cost, but it has not proven to be significant for real applications. DLM costs are comparable to VMS and thus meet our goals. Audited TPC-C results with the Oracle database also validated both our design approach and the implementation details by showing that database performance and scaling with additional cluster nodes meet our expectations.

The previous best reported TPC-C numbers were 20,918 tpmC on Tandem Computers' Himalaya K10000-112 system with the proprietary NonStop SQL/MP database software. The best reported numbers with open database software were 11,456 tpmC on the Digital AlphaServer 8400 5/350 with Oracle7 version 7.3. A four-node AlphaServer 8400 5/350 cluster with Oracle Parallel Server was recently audited at 30,390 tpmC. This represents industry-leadership performance with nonproprietary database software.

Future Developments

We will continue to evolve the TruCluster product toward a more scalable, more general computing environment. In particular, we will emphasize distributed file systems, configuration flexibility, management tools, and a single-system view for both internal and client applications. Work is under way for a cluster file system with local node semantics across the cluster system. The new cluster file system will not replace DRD but will complement it, giving applications the choice of raw access through DRD or full, local-file-system semantics. We are also lifting the four-node limitation and allowing more flexibility in cluster interconnect and storage configurations. A single network address for the cluster system is a priority. Finally, further steps in managing a multinode system as a single system will become even more important as the scale of cluster systems increases.

Further in the future is a true single-system view of cluster systems that will transparently extend all process control, communication, and synchronization mechanisms across the entire cluster. An implicit transparency requirement is performance.

Acknowledgments

In addition to the authors, the following individuals contributed directly to the cluster components described in this paper: Tim Burke, Charlie Briggs, Dave Cherkus, and Maria Vella for DRD; Joe Amato and Mitch Condylis for DLM; and Ali Rafieymehr for MEMORY CHANNEL. Hai Huang, Jane Lawler, and especially project leader Brian Stevens made many direct and indirect contributions to the project. Thanks also to Dick Buttlar for his editing assistance.

References and Notes

1. "Introduction to DCE," *OSF DCE Documentation Set* (Cambridge, Mass.: Open Software Foundation, 1991).
2. Internet RFCs 1014, 1057, and 1094 describe ONC XDR, RPC, and NFS protocols, respectively.
3. G. Pfister, *In Search of Clusters* (Upper Saddle River, N.J.: Prentice-Hall, Inc., 1995): 19–26.
4. N. Kronenberg, H. Levy, and W. Strecker, "VAXclusters: A Closely-Coupled Distributed System," *ACM Transactions on Computer Systems,* vol. 4, no. 2 (May 1986): 130–146.
5. L. Cohen and J. Williams, "Technical Description of the DECsafe Available Server Environment," *Digital Technical Journal,* vol. 7, no. 4 (1995): 89–100.
6. TPC performance numbers for UNIX systems are typically reported for databases using the character device interface.
7. The file system interfaces on the Digital UNIX operating system are being extended to support direct I/O, which results in bypassing the block buffer cache and reducing code path length for those applications that do not benefit from use of the cache.
8. A fast wide differential (FWD) SCSI bus is limited to a maximum distance of about 25 meters for example.
9. M. Devarakonda et al., "Evaluation of Design Alternatives for a Cluster File System," *USENIX Conference Proceedings,* USENIX Association, Berkeley, Calif. (January 1995).
10. J. Gray and A. Reuter, *Transaction Processing—Concepts and Techniques* (San Mateo, Calif.: Morgan Kaufman Publishers, 1993).
11. This mechanism is inherited from the DECsafe Available Server management facility, including the asemgr interface.
12. As an example, if the first DRD service created for a cluster is 1, the DRD device special file name is /dev/drd/drd1 and its minor device number is also 1.
13. C. Juszczak, "Improving the Performance and Correctness of an NFS Server," *USENIX Conference Proceedings,* USENIX Association, San Diego, Calif. (Winter 1989).
14. W. Snaman, Jr. and D. Thiel, "The VAX/VMS Distributed Lock Manager," *Digital Technical Journal,* vol. 1, no. 5 (September 1987): 29–44.
15. R. Goldenberg, L. Kenah, and D. Dumas, *VAX/VMS Internals and Data Structures* (Bedford, Mass.: Digital Press, 1991).

16. *TruCluster Application Programming Interfaces Guide* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QL8PA-TE, 1996).

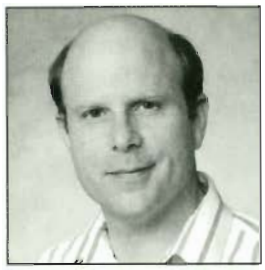
17. T. Rengarajan, P. Spiro, and W. Wright, "High Availability Mechanisms of VAX DBMS Software," *Digital Technical Journal*, vol. 1, no. 8 (February 1989): 88–98.

18. *Encore 91 Series Technical Summary* (Fort Lauderdale, Fla.: Encore Computer Corporation, 1991).

Biographies

Wayne M. Cardoza
Wayne Cardoza is a senior consulting engineer in the UNIX Engineering Group. He joined Digital in 1979 and contributed to various areas of the VMS kernel prior to joining the UNIX Group to work on the UNIX cluster product. Wayne was also one of the architects of PRISM, an early Digital RISC architecture; he holds several patents for this work. More recently, he participated in the design of the Alpha AXP architecture and the OpenVMS port to Alpha. Before coming to Digital, Wayne was employed by Bell Laboratories. He received a B.S.E.E. from Southeastern Massachusetts University and an M.S.E.E. from MIT.

Frederick S. Glover
Fred Glover is a software consulting engineer and the technical director of the Digital UNIX Base Operating System Group. Since joining the Digital UNIX Group in 1985, Fred has contributed to the development of networking services, local and remote file systems, and cluster technology. He has served as the chair of the IETF/TSIG Trusted NFS Working Group, as the chair of the OSF Distributed File System Working Group, and as Digital's representative to the IEEE POSIX 1003.8 Transparent File Access Working Group. Prior to joining Digital, Fred was employed by AT&T Bell Laboratories, where his contributions included co-development of the RMAS network communication subsystem. He received B.S. and M.S. degrees in computer science from Ohio State University and conducted his thesis research in the areas of fault-tolerant distributed computing and data flow architecture.



William E. Snaman, Jr.
Sandy Snaman joined Digital in 1980. He is currently a consulting software engineer in Digital's UNIX Software Group, where he contributed to the TruCluster architecture and design. He and members of his group designed and implemented cluster components such as the connection manager, lock manager, and various aspects of cluster communications. Previously, in the VMS Engineering Group, he was the project leader for the port of the VMScluster system to the Alpha platform and the technical supervisor and project leader for the VAXcluster executive area. Sandy also teaches MS Windows programming and C++ at Daniel Webster College. He has a B.S. in computer science and an M.S. in information systems from the University of Lowell.

Delivering Binary Object Modification Tools for Program Analysis and Optimization

Linda S. Wilson
Craig A. Neth
Michael J. Rickabaugh

Digital has developed two binary object modification tools for program analysis and optimization on the Digital UNIX version 4.0 operating system for the Alpha platform. The technology originated from research performed at Digital's Western Research Laboratory. The OM object modification tool is a transformation tool that focuses on postlink optimizations. OM can apply powerful intermodule and interlanguage optimizations, even to routines in system libraries. Atom, an analysis tool with object modification, provides a flexible framework for customizing the transformation process to analyze a program. With Atom, compilation system changes are not needed to create both simple and sophisticated tools to directly diagnose or debug application-specific performance problems. The linker and loader are enhanced to support Atom. The optimizations OM performs can be driven from performance data generated with the Atom-based *pixie* tool. Applying OM and Atom to commercial applications provided performance improvements of up to 15 percent.

Historically on UNIX systems, optimization and program analysis tools have been implemented primarily in the realm of compilers and run-time libraries. Such implementations have several drawbacks, however. For example, although the optimizations performed by compilers are effective, typically, they are limited to those that can be performed within the scope of a single source file. At best, the compiler can optimize the set of files presented during one compilation run. Even the most sophisticated systems that save intermediate representations usually cannot perform optimizations of calls to routines in system libraries or other libraries for which no source or intermediate forms are available.[1]

The traditional UNIX performance analysis tools, prof and gprof, require compiler support for inserting calls to predefined run-time library routines at the entry to each procedure. The monitor routines allow more user control over prof and gprof profiling capabilities, but their usage requires modifications to the application source code. Because these capabilities are implemented as compilation options, users of the tools must recompile or, in the case of the monitor routines, actually modify their applications. For a large application, this can be an onerous requirement. Further, if the application uses libraries for which the source is unavailable, many of the analysis capabilities are lost or severely impaired.

By comparison, object modification tools can perform arbitrary transformations on the executable program. The OM object modification tool is a transformation tool that focuses on postlink optimizations. By performing transformations after the link step, OM can apply powerful intermodule and interlanguage optimizations, even to routines in system libraries.

Object transformations also have benefits in the area of program analysis. Atom, an analysis tool with object modification, provides a flexible framework for customizing the transformation process to analyze a program. With Atom, compilation system changes are not needed to develop specialized types of debugging or performance analysis tools. Application developers can create both simple and sophisticated tools to directly diagnose or debug application-specific performance problems.

The OM and Atom technologies originated from research performed at Digital's Western Research Lab (WRL) in Palo Alto, California.[2] The software was developed into products by the Digital UNIX Development Environment (DUDE) group at Digital's UNIX engineering site in Nashua, New Hampshire. Both technologies are currently shipping as supported products on Digital UNIX version 4.0 for the Alpha platform.[3]

This paper first provides technical overviews for both OM and Atom. An example Atom tool is presented to demonstrate how to use the Atom application programming interface (API) to develop a customized program analysis tool. Because OM and Atom can be used together to enhance the effectiveness of optimizations to application programs, the paper includes an overview regarding the benefits of profiling-directed optimizations.

Subsequent sections discuss the product development and technology transfer process for OM and Atom and several design decisions that were made. The paper describes the working relationship between WRL and DUDE, the utilization of the technology on Independent Software Vendor (ISV) applications, and the factors that drove the separate development strategies for the two products. The paper concludes with a discussion about areas for further investigation and plans for future enhancements.

Technology Origins

Researchers at WRL investigating postlink optimization techniques created OM in 1992.[4] Unlike compile-time optimizers, which operate on a single file, postlink optimizers can operate on the entire executable program. For instance, OM can remove procedures that were linked into the executable but were never called, thereby reducing the text space required by the program and potentially improving its paging behavior.[5]

Using the OM technology, the researchers further discovered that the same binary code modification techniques needed for optimizations could also be applied to the area of program instrumentation. In fact, the processes of instrumenting an existing program and generating a new executable could be encapsulated and a programmable interface provided to drive the instrumentation and analysis processes. Atom evolved from this work.[6,7]

In 1993, WRL researchers Amitabh Srivastava and Alan Eustace began planning with DUDE engineers to provide OM and Atom as supported products on the Digital UNIX operating system. Different product development and technology transfer strategies were used for delivering the two technologies. The section Product Development Considerations discusses the methods used and the forces that influenced the strategies.

Technical Overview of OM

OM performs transformations in three phases. It produces an intermediate representation, performs optimizations on that representation, and produces an executable image.

Intermediate Representation

In the first phase, OM reads a specially linked input file that is produced by the linker, parses the object code, and produces an intermediate representation of the instructions in the program. The flow information and the program structure are maintained in this representation.

Optimization

In the optimization phase, OM performs various transformations on the intermediate representation created in the first phase. These transformations include

- Text size reductions
- Data size reductions
- Instruction and data reorganization to improve cache behavior
- Instruction scheduling and peephole optimization
- User-directed procedure inlining

Text Size Reductions One type of text size reduction is the elimination of unused routines. Starting at the entry point of the image, OM examines the instruction stream for transfer-of-control instructions. OM follows each transfer of control until it finds all reachable routines in the image. The remaining routines are potentially unreachable and are candidates for removal. Before removing them, OM checks all candidates for any address references. (Such references will show up in the relocation entries for the symbols.) If no references exist, OM can safely remove the routine. A second type of text size reduction is the elimination of most GP register reloading sequences.[8,9]

Data Size Reductions Because it operates on the entire program, OM performs optimizations that compilers are not able to perform. One instance is with the addressability of global data. The general instruction sequence for accessing global data requires the use of a table of address constants (the .lita section) and code necessary for maintaining the current position in the table. Each entry in the address constant table is relocated by the linker. Because OM knows the location of all global data, it can potentially remove the address entry while inserting and removing code to more efficiently reference the data directly. Removing as many of the .lita entries as possible leaves more room in the data cache (D-cache) for the application's global data.

This optimization is not possible at link time, because the linker can neither insert nor remove code.

Reorganization of the Image By default, OM reorganizes an image by reordering the routines in the image as determined by a depth-first search of the routine order, starting at the main entry point. In the absence of profiling information, this ordering is usually better than the linker's ordering.

In the presence of profiling feedback, OM performs two more instruction-stream reorderings: (1) reordering of routines based on basic block counts and (2) routine ordering based on execution frequency. OM first reorganizes routines based on the basic block information collected by a previous run of the image instrumented with the Atom-based pixie tool. OM lays the basic blocks to match the program's likely flow of control. Branches are aligned to match the hardware prediction mechanism. As a result, OM packs together the most commonly executed blocks. After basic block reorganization, OM then reorders the routines in the image based on the cumulative basic block counts for each routine. The reorganized image is ordered in a way similar to the way the prof tool displays execution statistics. The reordering performed by OM is superior to that performed by cord, because cord does not reorder basic blocks. cord is a UNIX profiling-directed optimization utility that reorders procedures in an executable program to improve cache performance. The cord(1) reference page on Digital UNIX version 4.0 describes the operation of this utility in more detail.

Elapsed-time Performance The optimizations that OM performs without profiling feedback can provide elapsed-time performance improvements of up to 5 percent. The feedback-directed optimizations can often provide an additional improvement of from 5 to 10 percent in elapsed time, for a total improvement of up to 15 percent over an image not processed by OM. Several commercial database programs have realized elapsed-time performance improvements ranging from 9 to 11 percent with feedback.

Executable Image

Finally, in the third phase, OM reassembles the transformed intermediate representation into an executable image. It performs relocation processing to reflect any changes in data layout or program organization.

Technical Overview of Atom

The Atom tool kit consists of a driver, an instrumentation engine, and an analysis run-time system. The Atom engine performs transformations on an executable program, converting it to an intermediate form. The engine then annotates the intermediate form and generates a new, instrumented program.

The Atom engine is programmable. Atom accepts as input an instrumentation file and an analysis file. The instrumentation file defines the points at which the program is to be instrumented and what analysis routine is to be called at each instrumentation point. The analysis file (defined later in this section) defines the analysis routines. Atom allows instrumentation of a program at a very fine level of granularity. It supports instrumentation before and after

- Program execution
- Shared library loading
- Procedures
- Basic blocks
- Individual instructions

Supporting instrumentation at these points allows the development of a wide variety of tools, all within the Atom framework. Examples of these tools are cache simulators, memory checking tools, and performance measurement tools. The framework supports the creation of customized tools and can decrease costs by simplifying the development of single-use tools.

The instrumentation file is a C language program that contains calls to the Atom API. The instrumentation file defines any arguments to be passed to the analysis routine. Arguments can be register values, instruction fields, symbol names, addresses, etc. The instrumentation file is compiled and then linked with the Atom instrumentation engine to create a tool to perform the instrumentation on a target program.

The analysis file contains definitions of the routines that are called from the instrumentation points in the target program. The analysis routines record events or process the arguments that are passed from the instrumentation points.

By convention, the instrumentation and analysis files are named with the suffixes inst.c and anal.c, respectively. Atom is invoked as follows to create an instrumented executable:

```
% atom program tool.inst.c tool.anal.c
```

The atom command is a driver that invokes the compiler and linker to generate the instrumented program. The five steps of this process are

1. Compile the instrumentation code.
2. Link the instrumentation code with the Atom instrumentation engine to create an instrumentation tool.
3. Compile the analysis code.
4. Link the analysis code with the Atom analysis run-time system, using the UNIX ld tool with the -r option so the object may be used as input to another link.

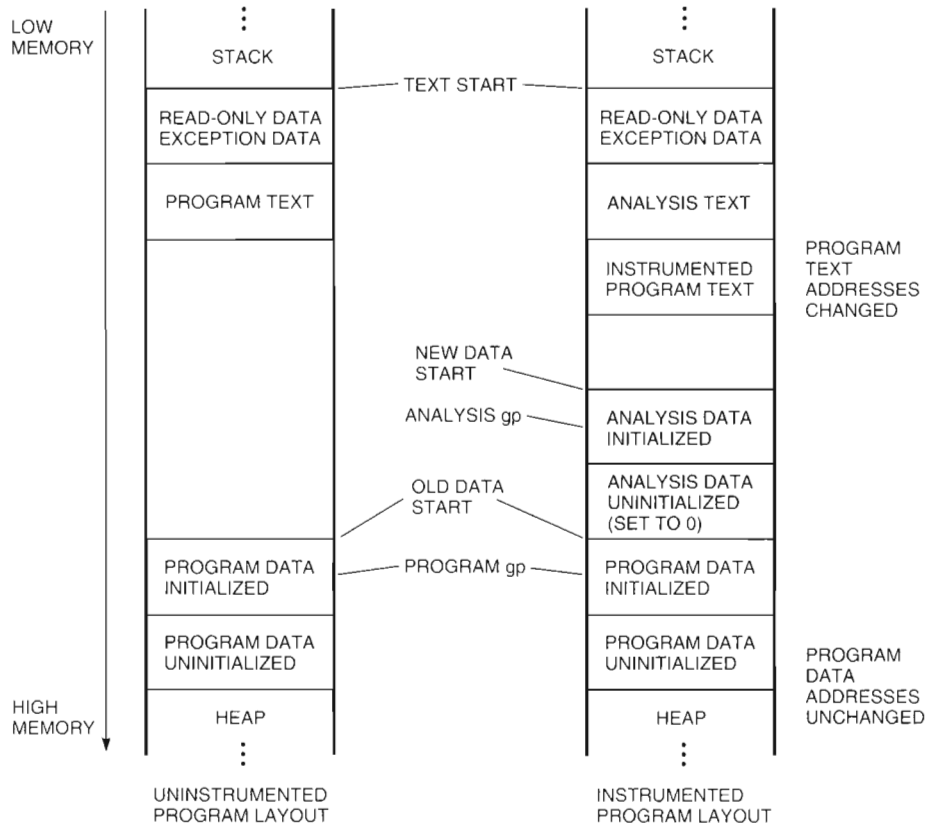
5. Execute the instrumentation tool on the target program, providing the linked analysis code as an argument.

The final step produces an instrumented program linked with the analysis code. Figure 1 shows the changes in memory layout between the original program and the instrumented program.

An Example Atom Tool for Memory Debugging

The following discussion of an example Atom tool demonstrates how to use the Atom API to develop a customized program analysis tool.

A common development problem is locating the source of a memory overwrite. Figure 2 shows a contrived example program in which the loop to initialize an array exceeds the array boundary and overwrites a



Source: A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation,* Orlando, Fla. (June 1994). This paper is also available as Digital's Western Research Laboratory (WRL) Research Report 94/2.

Figure 1
Memory Layout of Instrumented Programs

```
 1  long numbers[8] = {0};
 2  long *ptr = numbers;          /* This pointer is overwritten */
 3
 4  main()
 5  {
 6        int i;
 7
 8        for(i=0; i<25; i++)     /* by this array initialization. */
 9           numbers[i] = i;
10  }
```

Figure 2
Example Program with Memory Overwrite

pointer variable. In this case, the initialization of the *numbers* array defined in line 1 overwrites the contents of the variable *ptr* defined in line 2. This type of problem is often difficult and time-consuming to locate with traditional debugging tools.

Atom can be used to develop a simple tool to locate the source of the overwrite. The tool would instrument each store instruction in the program and pass the effective address of the store instruction and the value being stored to an analysis routine. The analysis routine would determine if the effective address is the address being traced and, if so, generate a diagnostic.

The instrumentation and analysis files for the mem_debug tool are shown in Figure 3. InstrumentInit( ) registers the analysis routines with the Atom instrumentation engine and specifies that calls to the get_args( ) and open_log( ) routines be inserted before the program begins executing. A call to the close_log( ) routine is dictated when the program terminates execution. The Atom instrumentation engine calls InstrumentInit( ) exactly once.

The Atom instrumentation engine calls the Instrument( ) routine once for each executable object in the program. The routine instruments each store instruction that is not a stack operation with a call to the analysis routine mem_store( ). Each call to the routine provides the address of the store instruction, the target address of the store instruction, the value to be stored, and the file name, procedure name, and line number.

The open_log( ) and close_log( ) analysis routines are self-explanatory. The messages could have been written to the standard output, because, in this example, they would not have interfered with the program output.

The get_args( ) routine reads the value of the *MEM_DEBUG_ARGS* environment variable to obtain the data address to be traced. The tool could have been written to accept arguments from the command line using the -toolargs switch. The instrumentation code would then pass the arguments to the analysis routine. In the case of this tool, using the environment variable to pass the arguments is beneficial because the program does not have to be reinstrumented to trace a new address.

The mem_store( ) routine is called from each store instruction site that was instrumented. If the target address of the store operation does not match the trace address, the routine simply returns. If there is a match, a diagnostic is logged that gives information about the location of the store.

To demonstrate how this tool would be used, suppose one has determined by debugging that the variable *ptr* is being overwritten. The nm command is used to determine the address of *ptr*, as follows:

```
% nm -B program | grep ptr
0x000001400000c0 G ptr
```

Instrument the program with the mem_debug tool.

```
% atom program mem_debug.inst.c
  mem_debug.anal.c
```

Set the *MEM_DEBUG_ARGS* environment variable with the address to trace.

```
% setenv MEM_DEBUG_ARGS 1400000c0
```

Run the instrumented program,

```
% program.atom
```

and view the log file.

```
% more program.mem_debug.log

Tracing address 0x1400000c0

Address 0x1400000c0 modified with\
  value 0x16:
at : 0x1200011c4  Procedure: main\
  File: program.c  Line: 9
```

Using this simple Atom tool, the location of a memory overwrite can be detected quickly. The instrumented program executes at nearly normal speed. Traditional debugging methods to detect such errors are much more time-consuming.

Other Tools

An area in which Atom capabilities have proven particularly powerful is for hardware modeling and simulation. Atom has been used as a teaching tool in university courses to train students on hardware and operating system design. Moreover, Digital hardware designers have developed sophisticated Atom tools to evaluate designs for new implementations of the Alpha chip.

The Atom tool kit contains 10 example tools that demonstrate the capabilities of this technology. The examples include a branch prediction tool, which is useful for compiler designers, a procedure tracing tool, which can be useful in following the flow of unfamiliar code, and a simple cache simulation tool.

Atom Tool Environments

Analysis of certain types of programs can require use of specially designed Atom tools. For instance, to analyze a program that uses POSIX threads, an Atom tool to handle threads must also be designed, because the analysis routines will be called from the threads in the application program. Therefore, the analysis routines need to be reentrant. They may also need to synchronize access to data that is shared among the threads or manage data for individual threads. The thread management in the analysis routines adds overhead to the execution time of the instrumented program; this overhead is not necessary for a nonthreaded program. To effectively support both threaded and nonthreaded programs, two distinct versions of the same Atom tool need to coexist. Designers developed the concept of tool environments to address the issues of providing multiple versions of an Atom tool.

```
/*
 * mem_debug_inst.c      - Instrumentation for memory debugging tool
 *
 * This tool instruments every store operation in an application and
 * reports when the application writes to a user-specified address.
 * The address should be an address in the data segment, not a
 * stack address.
 *
 * Usage: atom program mem_debug.inst.c mem_debug.anal.c
 *
 */

#include <string.h>
#include <cmplrs/atom.inst.h>

/*
 * Initializations: register analysis routines
 *                  define the analysis routines to call before and after
 *                  program execution
 *
 * get_args() - reads environment variable MEM_DEBUG_ARGS for address to trace
 * open_log() - opens the log file to record overwrites to the specified address
 * close_log() - closes the log file at program termination
 * mem_store() - checks if a store instruction writes to the specified address
 */
void InstrumentInit(int argc, char **argv)
{
    AddCallProto("get_args()");
    AddCallProto("open_log(const char *)");
    AddCallProto("close_log()");
    AddCallProto("mem_store(VALUE,REGV,long,const char *,const char *,int)");

    AddCallProgram(ProgramBefore, "get_args");
    AddCallProgram(ProgramBefore, "open_log",
                   basename((char *)GetObjName(GetFirstObj())));
    AddCallProgram(ProgramAfter, "close_log");
}

/*
 * Instrument each object.
 */
Instrument(int argc, char *argv[], Obj *obj)
{
    Proc  *proc;
    Block *block;
    Inst  *inst;
    int   base;          /* base register of memory reference */

    /*
     * Search for all of the store instructions into the data area.
     */
    for (proc = GetFirstObjProc(obj);  proc;  proc = GetNextProc(proc)) {
        for (block = GetFirstBlock(proc);  block;  block = GetNextBlock(block)){
            for (inst = GetFirstInst(block);  inst;  inst = GetNextInst(inst)) {
                /*
                 * Instrument memory references. Skip $sp based references
                 * because they reference the stack, not the data area.
                 * Memory references are instrumented with a call to the
                 * mem_store analysis routine. The arguments passed are
                 * the target address of the store instruction,
                 * the value to be stored at the target address,
                 * the PC address of the store instruction in the program,
                 * the procedure name, file name, and source line for the
                 * PC address.
                 */
```

Figure 3
Instrumentation and Analysis Code for the mem_debug Tool

```
                if (IsInstType(inst, InstTypeStore)) {
                    base = GetInstInfo(inst, InstRB);
                    if (base != REG_SP) {
                        AddCallInst(inst, InstBefore, "mem_store",
                            EffAddrValue,
                            GetInstRegEnum(inst, InstRA),
                            InstPC(inst),
                            ProcName(proc),
                            ProcFileName(proc),
                            (int)InstLineNo(inst));
                    }
                }
            }
        }
    }
}

/*
 * mem_debug.anal.c - analysis routines for memory debugging tool
 *
 * Usage: setenv MEM_DEBUG_ARGS hex_address before running
 * the program.
 * Diagnostic output is written to program.mem_debug.log
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

static FILE *log_file;                  /* Output file for diagnostics */

static caddr_t trace_addr;              /* Address to monitor */

/*
 * Create log file for diagnostics.
 */
void
open_log(const char *progname)
{
    char        name[200];

    sprintf(name, "%s.mem_debug.log", progname);
    log_file = fopen(name, "w");

    if (!log_file) {
        fprintf(stderr, "mem_debug: Can't create %s\n", name);
        fflush(stderr);
        exit(1);
    }

    fprintf(log_file, "Tracing address 0x%p\n\n", trace_addr);
    fflush(log_file);
}

/*
 * Close the log file.
 */
void
close_log(void)
{
    fclose(log_file);
}

/*
 * Get address to trace from the environment.
 */
void
get_args(void)
```

Figure 3 (continued)

```
{
    char *addr;
    if (!(addr = getenv("MEM_DEBUG_ARGS"))) {
        fprintf(stderr, "mem_debug: set MEM_DEBUG_ARGS to hex address\n");
        fflush(stderr);
        exit(1);
    }
    trace_addr = (caddr_t) strtoul(addr, 0, 16);
}

/*
 * The target address is about to be modified with the given value.
 * If this is the address being traced, report the modification.
 */
void
mem_store(
    caddr_t             target_addr,    /* Address being stored into */
    unsigned long       value,          /* Value being stored at target_addr */
    caddr_t             pc,             /* PC of this store instruction */
    const char          *proc,          /* Procedure name */
    const char          *file,          /* File name */
    unsigned            line)           /* Line number */
{
    if (target_addr == trace_addr) {
            fprintf(log_file, "Address 0x%p modified with value 0x%lx:\n",
                    target_addr, value);
            fprintf(log_file, "\tat   : 0x%p  ", pc);
            if(proc != NULL) {
               fprintf(log_file,"Procedure: %s  ", proc);
               if(file != NULL)
                 fprintf(log_file,"File: %s  Line: %d", file, line);
            }
            fprintf(log_file, "\n");
            fflush(log_file);
    }
}
```

Figure 3 (continued)

Tool environments accommodate seamless integration of specialized versions of tools into the Atom tool kit. They provide a means for extending the Atom kit. This facility allows the addition of specialized Atom tools by Digital's layered product groups or by customers, while maintaining a consistent user interface.

The versions of the Atom tools hiprof, pixie, and Third Degree that support POSIX threads are provided as a separate environment. hiprof is a performance analysis tool that collects data similar to but with more precision than gprof. pixie is a basic block profiling tool. Third Degree is a memory leak detection tool.

The following command invokes the Atom-based pixie tool for use on a nonthreaded program:

```
% atom program -tool pixie
```

The following command invokes the version of the pixie tool that supports threaded programs:

```
% atom program -tool pixie -env threads
```

Tools for other specialized environments can be provided by defining a new environment name. For example, Atom tools written to work with a language-specific run-time environment can be added to the Atom tool kit by selecting an environment name for the category of tools. Similarly, tools designed to work on the kernel could be collected into an environment.

The environment name is used in the names of the tool's instrumentation, analysis, and description files. The description file for a tool provides the names of the instrumentation and analysis files, as well as special instructions for compiling and linking the tool. For example, the pixie description file for threaded programs is named pixie.threads.desc. It identifies the threaded versions of the pixie instrumentation and analysis files. The Atom driver builds the name of the description file from the arguments to the -tool and -env switches on the command line. The contents of the description file then drive the subsequent steps of the build process.

Tool environments can be added without modification to the base Atom technology, thereby providing extensibility to the tool kit while maintaining a consistent interface.

Compact Relocations

Atom inserts code into the text of the program, thus changing the location of routines. Atom requires that relocation information be retained in the

executable image created by the linker. This allows Atom to properly perform relocations on the instrumented executable.

During the normal process of linking, the relocation entries stored in object files are eliminated once they have been resolved. Because it effectively relinks the executable, Atom must have access to the relocation information.

Consider, for example, an application that invokes a function through a statically initialized function pointer variable, as shown in the following code segment:

```
void foo(int a, int b)
{
        ...
}

void (*ptr_foo)(int, int) = foo;

void bar()
{
        ...
        (*ptr_foo)(1,2);
}
```

The address of function *foo* is stored in the memory location referenced by the *ptr_foo* variable. When Atom instruments this application, the address of *foo* will change, and Atom needs to know to update the contents of the memory location referenced by *ptr_foo*. This is possible only if there is a relocation record pointing at this memory location. Adding compact relocations to the executable file solves this problem.

Compact relocations are smaller than regular relocations for two reasons. First, the Atom system does not require all the information in the regular relocation records in order to instrument an executable. Atom changes only the layout of the text segment, so relocation records that describe the data segments are not needed. Second, the remaining relocations can often be predicted by analyzing other parts of the executable file. This property is used to store a compact form of the remaining relocation records. Since compact relocation records are represented in a different form than regular relocations, they are stored in the .comment section of the object file rather than in the normal relocation area.

Profiling-directed Optimization

OM and the Atom-based pixie tool can interoperate using profiling-directed optimization. The Atom-based pixie tool is a basic block profiler that provides execution counts for each basic block when the program is run. The execution counts are then used as input to OM for performing optimizations on the executable that are driven from actual run-time performance data.

As an example, the following steps would be performed to utilize profiling-directed optimizations with OM:

```
1. % cc -non_shared -o program *.o
2. % atom -tool pixie program
3. % program.pixie
4. % cc -non_shared -om
     -WL,-om_ireorg_feedback,program *.o
```

In step 1, a nonshared version of the program is built. In step 2, the Atom-based pixie tool instruments the program. Step 2 produces program.pixie and program.Addrs files. Step 3 results in the execution of the instrumented program to generate a program.Counts file. This file contains an execution count for each basic block in the program. The last step provides the basic block profile as input to OM. OM rearranges the text segment of the program such that the most frequently executed basic blocks and procedures are placed in proximity to each other, thus improving the instruction cache (I-cache) hit rate.

Product Development Considerations

Bringing the OM and Atom technologies from the laboratory into use on current Digital UNIX production systems required frequent communication and coordination between WRL and DUDE engineers working on opposite coasts of the U.S. The success of both projects depended upon establishing and maintaining an atmosphere of cooperation among the engineers at the two locations. Common goals and criteria for bringing the technology to product supported the teams during development and planning work.

Among the product development considerations for OM and Atom were

1. The products must address a business or customer requirement.
2. The products must meet customer expectations of features, usability, quality, and performance.
3. Engineering, quality assurance, and documentation resources must be identified to ensure that the products could be enhanced, updated to operate on new platform releases, and supported throughout their life cycles.
4. The products must be released at the appropriate times. Releasing a product too early could result in high support costs, perhaps at the expense of new development. Releasing a product too late could compromise Digital's ability to leverage the new technology most effectively.

Product Development and Technology Transfer Process for OM

As part of their research and development efforts, WRL engineers applied OM to large applications. Researchers and Digital engineers at ISV porting laboratories worked together to share information and to diagnose the performance problems of programs in

use on actual production systems, such as relational database and CAD applications. This cooperative effort helped engineers determine the types of optimizations that would benefit the broadest range of applications. In addition, the engineers were able to identify those optimizations that would be useful to specific classes of applications and make them switch-selectable through the OM interface. The performance improvements achieved on ISV applications enabled OM to meet the criteria for addressing customer needs.

Although WRL researchers also applied OM to the SPEC benchmark suite to measure performance improvements, the primary focus of the OM technology development was to provide performance improvements for applications currently in widespread use by the Digital UNIX customer base. With the focus of performance improvements on large customer applications, OM satisfied a prominent Digital business need for inclusion in the Digital UNIX development environment.

Engineers discussed the limitation that OM did not support shared libraries and the programs that used them. In this respect, the technology would not meet the expectations of all customers. Many ISV applications and other performance-sensitive programs, however, are built nonshared to improve execution times. Engineers determined that the benefits for this class of application outweighed this limitation of OM, and, therefore, the limitation did not prevent moving forward to develop the prototype into a product. Developers recognized the risks and support costs associated with shipping the prototype, yet again decided that the proven benefits to existing applications outweighed these factors.

Because of the pressing business and customer needs for this technology, DUDE and WRL engineering concurred that OM should be provided as a fully supported component in Digital UNIX version 3.0. Full product development commitments from DUDE engineering, documentation, and quality assurance could not be made for that release, however. After discussion, WRL provided technical support and extensions to OM to address this need. DUDE engineering agreed to provide consulting support to WRL researchers on object file and symbol table formats and on evaluations of text and data optimizations.

The next issue the engineers faced was how to integrate OM into the existing development environment. They evaluated three approaches.

The first approach was to make OM a separate tool directly accessible to users as /usr/bin/om. Thus, an application developer could utilize OM as a separate step during the build process. This approach offered two advantages. First, it was similar to the approach used to achieve the present internal use of OM and would require few additional modifications to the Digital UNIX development environment. The second advantage was that Atom and OM could be more easily merged into one tool since their usage would be similar. This merging would provide the potential efficiencies of a single stream of sources for the object modification technology.

A major disadvantage of this approach was that it put additional burden on the application developer. OM requires a specially linked input file produced by the linker. This intermediate input file is not a complete executable nor is it a pure OMAGIC file.[10] This approach would require customers to add and debug additional build steps to use OM on their applications. The WRL and DUDE engineers agreed that the user complexity of this approach would be a significant barrier to user acceptance of OM.

The second approach was to change the compiler driver to invoke OM for linking an executable. With this approach, a switch would be added to the compiler driver. If this switch was given, the driver would call /usr/lib/cmplrs/cc/om instead of the system linker to do the final link.

This approach had the advantage of reducing the complexity of the user interface for building an application with OM. A developer could specify one switch to the compiler driver, and the driver would automatically invoke OM. This would allow a developer to introduce feedback-directed optimizations into the program by simply relinking with the profiling information, thus making OM easier to use and less error-prone.

The disadvantage of this second approach was that the complex symbol resolution process in the linker would need to be added to OM. The process of performing symbol resolution on Digital UNIX operating systems is nontrivial. There are special rules, boundary conditions, and constraints that the linker must understand. OM was designed to modify an already resolved executable, and any problems introduced from adding linker semantics would discourage its use. Also, duplicating linker capabilities in OM would require additional overhead in maintaining both components.

The advantages and disadvantages of the second approach motivated the development of a third approach. The compiler driver could be changed to invoke OM during a postlink optimization step. As in the second approach, a switch from the developer would trigger the invocation of OM; however, OM would be run after the linker had performed symbol and library resolution.

The third approach is the one currently used. This method maintains separation between the linking and optimization phases. When directed by the -om switch, ld produces a specially linked object that will be used as input to OM. The compiler driver supplies this object as input to OM when the linking is completed.

The WRL and DUDE engineers found that this functional separation also improved the efficiency of the development efforts between WRL and DUDE. The separation permitted concurrent WRL development on OM and DUDE development on ld, with minimal interference. This approach allowed more development time to be dedicated to technical issues rather than dealing with source management and integration issues.

DUDE engineers added the OM sources into the Digital UNIX development pool and integrated updates from WRL. WRL assumed responsibility for testing OM prior to providing source updates. As previously outlined, DUDE engineers integrated support for OM into the existing development environment tools for the initial release.

Because of proven performance improvements on ISV applications, committed engineering efforts by WRL, and testing activities at both Digital sites, engineers judged the technology mature enough for release on production systems. Efficient development strategies enabled Digital to rapidly turn this leading-edge technology into a product that benefits an important segment of the Digital UNIX customer base.

WRL continued engineering support for OM through the Digital UNIX version 3.0 and 3.2 releases. Responsibility for the technology gradually shifted from WRL to DUDE over the course of these releases. Currently, DUDE fully supports and enhances OM while WRL continues to provide consultation on the technology and input for future improvements.

Product Development and Technology Transfer Process for Atom

WRL deployed early versions of the Atom tool kit at internal Digital sites, ISV porting laboratories, and universities, thus allowing developers to experiment with and evaluate the Atom API. The early availability of the tool kit promoted use of the Atom technology. User feedback and requests for features helped the engineers to more quickly and effectively develop a robust technology from the prototype.

Engineers throughout Digital recognized Atom as a unique and useful technology. Atom's API, with instrumentation and analysis capabilities down to the instruction level, increased the power and diversity of tools that could be created for software and hardware development. Hardware development teams used Atom to simulate the performance of new Alpha implementations. Software developers created and shared Atom tools for debugging and measuring program performance. The value of the Atom technology in solving application development problems provided the business justification for developing the technology into a product.

The prototype version of Atom had several limitations.

- Like OM, the prototype version of Atom worked only on nonshared applications. A production version of Atom would require support for call-shared programs and shared libraries, since, by default, programs are built as call-shared programs. A viable Atom product offering needed to support these types of programs, in addition to nonshared programs.
- Programs needed to be relinked to retain relocation information before Atom could be used. This additional build step impaired the usability of Atom.
- The Atom prototype performed poorly because it consumed a large amount of memory. Much of the data collected about an executable for optimization purposes was not needed for program analysis transformations.
- The Atom API required extensive design and development to support call-shared programs and shared libraries.

The engineers decided to allow the OM and Atom technologies to diverge so that the differing requirements for optimization and program analysis could be more effectively addressed in each component.

Because the cost of supporting a release of the Atom prototype would have been high, WRL and DUDE engineering developed a strategy for simultaneously releasing the Atom prototype while focusing engineering efforts on developing the production version. An Atom Advanced Development Kit (ADK) was released with Digital UNIX version 3.0 as the initial step of the strategy. The ADK provided customer access to the technology with limited support. Engineers viewed the lack of support for shared executable objects as an acceptable limitation for the Atom ADK but unacceptable for the final product.

In addition to allowing WRL and DUDE engineers to focus on product development, this first strategic step permitted the engineering teams more time and flexibility to incrementally add support for Atom into other production components, such as the linker and the loader. For usability purposes, minor extensions were made to the loader to allow it to automatically load instrumented shared libraries produced by Atom tools.

The second step of the strategy was to provide updated Atom kits to users as development of the software progressed. These kits included the source code for example tools, manuals, and reference pages. The update kits performed two functions; they supported users and they provided feedback to the development teams. DUDE and WRL engineers posted information internally within Digital when kits were available and developed a mailing list of Atom users. The Atom user

community grew to include universities and several external customers.

Once the Atom ADK and update strategy were established, DUDE engineering began to implement support for Atom in the linker. As mentioned earlier, Atom inserts text into a program and requires relocation information to create a correctly instrumented executable. The Atom prototype required a program to be linked to retain the relocation information, and this requirement presented a usability problem for users. Ideally, Atom would be able to instrument the executables and shared libraries produced by default by the linker.

Modifying the linker to retain all traditional relocation information by default was not acceptable since the size increase in the executable would have been prohibitive. In some cases, 40 percent of the object file consists of relocation records. Engineers did not view an increase of that magnitude as a viable solution. In addition, this solution conflicted with the goal of Digital UNIX version 3.0 to reduce object file size. As a compromise, DUDE engineering implemented compact relocation support in the linker. Compact relocations provided an acceptable solution since they required far less space than regular relocation records, typically less than 0.1 percent of the total file size.

Another side effect of using compact relocations as a solution was that it introduced a dependency between Atom and ld. All executable objects to be processed by Atom needed to have been generated with the linker that contained compact relocation support. Therefore, to support Atom, layered product libraries and third-party libraries needed to be relinked with the compact relocation support.

In Digital UNIX version 3.0, ld was modified to generate compact relocation information in executable objects. This allowed Atom to instrument the default output of ld. Engineers viewed this capability as critical to the usability and ultimate success of the Atom technology. The compact relocation support in ld was refined and extended over the course of several Digital UNIX releases as development work with Atom progressed.

Concurrently, the WRL research team expanded and began development of the Atom Third Degree and hiprof tools. WRL engineers also continued with additions and improvements to a suite of example Atom tools.

After the release of Digital UNIX version 3.0, DUDE began design and development of the production version of the core Atom technology and the API. DUDE engineers modified and extended the Atom API as tool development progressed at WRL. During peak development periods, engineers discussed design issues daily by telephone and electronic mail.

The original Atom ADK included the source code for a number of example Atom tools. Because some of these tools contained hardware implementation dependencies, they would require ongoing and long-term support to remain operational on changing implementations of the Alpha architecture. For the second shipment of the Atom ADK in Digital UNIX version 3.2, these high-maintenance tools were removed and made available through unsupported channels.

Between releases of the ADK on the Digital UNIX operating system, the engineering teams continued to deliver update kits. Engineers scheduled delivery of the update kits to coincide with key milestones in the software development process. This strategy gave them more control over the development schedule and minimized risk. The update kits provided immediate field test exposure for the evolving Atom software. The design, development, and kit process was practiced iteratively over a year to develop the original ideas into a full product. The Atom update kits were provided for Digital UNIX version 3.0 and later systems, since most users did not have access to early versions of Digital UNIX version 4.0. Providing Atom kits for use on pre–version 4.0 systems allowed the software to be exercised in the field on actual applications prior to its initial release as a supported product. Although support for earlier operating system versions added overhead and complexity to the process of providing the update kits, the engineering teams valued the abundance of user feedback that the process yielded. The benefits of user input to the software development process outweighed the overhead costs.

During Digital UNIX version 4.0 development, WRL engineers finalized the implementations of the hiprof and Third Degree tools and transferred the tool sources to DUDE for further development. The WRL developers had added support for threaded applications on pre–version 4.0 Digital UNIX systems. Because the implementation of threads changed in version 4.0, DUDE engineers needed to update the Atom tools accordingly.

DUDE engineers also developed an Atom-based pixie tool with support for threaded applications. In fact, the Atom-based pixie tool replaced the original version of pixie in Digital UNIX version 4.0. The Atom-based pixie allowed new features such as support for shared libraries and threads to be more efficiently added into the product offering. The development of an Atom-based pixie tool solved the extensibility problems that were being faced with the original version of pixie.

WRL engineers also began to use Atom for instrumenting pre–version 4.0 Digital UNIX kernels, developing special tools for collecting kernel statistics. Atom was extended by DUDE engineering as needed to support instrumentation and analysis of the kernel.

The Atom tool kit and example tools were shipped with Digital UNIX version 4.0. The pixie, hiprof, and Third Degree tools were shipped with the Software Development Environment subset of Digital UNIX version 4.0. Research regarding use of Atom for kernel instrumentation and analysis continues.

WRL continues to share ideas and consults with DUDE on the future directions for the Atom technology.

Conclusions

Developing OM into a product directly from research proved to be challenging. Problems and issues that needed to be addressed had to be handled within the schedule constraints and pressures of a committed release plan.

In contrast, the ADK method used to deliver the Atom product allowed the Atom developers to spend more time on product development issues in an environment relatively free from the pressures associated with daily schedules. The ADK mechanism, however, probably limited the exposure of Atom technology at some customer sites.

The close cooperation of engineers from both research and development was necessary to accomplish the goals of the two projects. We believe that a collaborative development paradigm was key to successfully bringing research to product.

Future Directions

This paper describes the evolution of the OM and Atom technologies through the release of the Digital UNIX version 4.0 operating system. Digital plans to investigate many new and improved capabilities, some intended for future product releases. Plans are under way to

- Provide support in OM for call-shared programs and shared libraries.
- Support the use of Atom tools on programs optimized with OM.
- Investigate providing an API to allow programmable, customized optimizations to be delivered through OM.
- Investigate reuse of instrumented shared libraries by multiple call-shared programs that have been instrumented with the same Atom tool.
- Research support for Atom tools that provide systemwide and per-process analysis of shared libraries.
- Extend Atom to improve kernel analysis.
- Simplify the use of the profiling-directed optimization facilities of Atom and OM through an improved interface.
- Extend the Atom tool kit to provide development support for thread-safe program analysis tools.

In addition to enhancements to the Atom product, original Atom-based tools are expected to become available through the development activities of students and educators at universities. Internal Digital developers will continue to develop and share tools as well.

Acknowledgments

Many people contributed to the development of the OM and Atom products. The following list gives recognition to those most actively involved. Amitabh Srivastava led the research and development work at WRL on OM and Atom and mediated many of the design discussions on the Atom design. Greg Lueck of DUDE designed and implemented the production version of Atom, compact relocations, and the Atom-based pixie tool. Alan Eustace developed Atom example tools, created the first Atom ADK, worked diligently with users, developed kernel tools, provided training and documentation on using Atom, and displayed eternal optimism. Russell Kao at WRL contributed the hiprof tool with thread support. Jeremy Dion and Louis Monier at WRL developed Third Degree and an Atom-based code coverage tool called tracker. John Williams and Chris Clark of DUDE completed the process of turning the hiprof, pixie and Third Degree tools into products. Dick Buttlar provided documentation on every component. Last but not least, the authors wish to extend a final thanks to all the users who contributed feedback to the OM and Atom development teams.

References

1. F. Chow, M. Himelstein, E. Killian, and L. Weber, "Engineering a RISC Compiler System," *Proceedings of COMPCON*, San Francisco, Calif. (March 1986): 132–137.
2. Western Research Laboratory, located on the Web at http://www.research.digital.com/wrl.
3. R. Sites and R. Witek, *Alpha AXP Architecture Reference Manual*, 2d ed. (Newton, Mass.: Digital Press, 1995).
4. A. Srivastava and D. Wall, "A Practical System for Intermodule Code Optimization at Link-time," *Journal of Programming Languages*, vol. 1 (1993): 1–18. Also available as WRL Research Report 92/6 (December 1992).
5. A. Srivastava, "Unreachable Procedures in Object-oriented Programming," *ACM LOPLAS*, vol. 1, no. 4 (December 1992): 355–364. Also available as WRL Research Report 93/4 (August 1993).

6. A. Eustace and A. Srivastava, "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools," *Proceedings of the Winter 1995 USENIX Conference*, New Orleans, La. (January 1995). Also available as WRL Technical Note TN-44 (July 1994).

7. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, Fla. (June 1994). Also available as WRL Research Report 94/2 (March 1994).

8. A. Srivastava and D. Wall, "Link-Time Optimization of Address Calculation on a 64-bit Architecture," *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, Fla. (June 1994). Also available as WRL Research Report 94/1 (February 1994).

9. *Digital UNIX Calling Standard for Alpha Systems*, Order No. AA-PY8AC-TE, Digital UNIX version 4.0 or higher (Maynard, Mass.: Digital Equipment Corporation, 1996).

10. *Digital UNIX Assembly Language Programmer's Guide*, Order No. AA-PS31C-TE, Digital UNIX version 4.0 or higher (Maynard, Mass.: Digital Equipment Corporation, 1996).

General Reference

J. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," *SIGPLAN Conference on Programming Language Design and Implementation* (June 1995).

Biographies

Linda S. Wilson
As a principal software engineer in the Digital UNIX Development Environment group, Linda Wilson leads the development of program analysis tools for the Digital UNIX operating system. In prior positions, she was responsible for the delivery of other development environment components, including DEC FUSE, the dbx debugger, and run-time libraries on the ULTRIX and Digital UNIX operating systems. Linda received a B.S. in computer science from the University of Nebraska-Lincoln. Before joining Digital in 1989, Linda held software engineering positions at Masscomp in Westford, Massachusetts, and Texas Instruments in Austin, Texas.

Craig A. Neth
Craig Neth is a principal software engineer in the Digital UNIX Development Environment group, where he is the technical leader of link-time tools. In prior positions at Digital, Craig has worked on the OM object modification tool and the VAX and DEC COBOL compilers, and led the development of DEC COBOL versions 1 and 2. Craig received a B.S. in computer science from Purdue University in 1984 and an M.S. in computer science from the University of Illinois in 1986.

Michael J. Rickabaugh
Michael Rickabaugh is a principal software engineer in the Digital UNIX Development Environment group. He started his Digital career in 1986 in the SEG/CAD Engineering group as a software engineer on the DECSIM logic simulation project. In 1991, Michael transitioned to the DEC OSF/1 AXP project and was a member of the original team responsible for delivering the UNIX development environment on the DEC OSF/1 Alpha platform. He has since been a technical contributor to all aspects of the Digital UNIX link-time technology as well as the creator of the ASAXP assembler for the Windows NT operating system. Michael received a B.S. in electrical and computer engineering from Carnegie Mellon University in 1986.

Design of eXcursion Version 2 for Windows, Windows NT, and Windows 95

John T. Freitas
James G. Peterson
Scot A. Aurenz
Charles P. Guldenschuh
Paul J. Ranauro

Version 2 of the eXcursion product is a complete rewrite of the successful Windows-based X server software package. Based on release 6 of the X Window System version 11 protocol, the new product runs on Microsoft's Windows, Windows NT, and Windows 95 operating systems. The X server is one of several components that compose this package. The other components are X Image Extension, the control panel (which constitutes the user interface for product configuration), the error logger, the application launcher, and the setup program. An interprocess communication facility enables the eXcursion components to communicate in a uniform fashion under all three operating systems. A unique server design using object-oriented programming techniques integrates the X graphics context with the Windows device context into a combined state management facility. The resulting implementation maximized graphics performance while conserving Windows resources, which are in limited supply under the 16-bit version of the Windows operating system. The control panel was coded completely in the C++ programming language, thus making full use of the Microsoft Foundation Class library to minimize development time and to ensure consistency with the Windows user interface paradigm.

Digital developed the eXcursion family of display server products to provide interoperability between desktop personal computers (PCs) running the Microsoft Windows operating system and remote hosts running the X Window System operating system under the UNIX or OpenVMS operating systems. The first version of the eXcursion X server was a 16-bit application written specifically for Microsoft Windows versions 3.0 and 3.1. As the popularity of Windows increased and desktop systems were connected to corporate networks, the market for X interoperability grew quickly. The 16-bit eXcursion code, much of which had been ported from 32-bit UNIX code, was again ported—this time to Microsoft's Win32 application programming interface (API) to support the Windows NT operating system. When release 6 of the X Window System version 11 protocol (X11R6) appeared and a new sample implementation source kit became available from the X Consortium, the eXcursion team decided that it was time for a complete rewrite of the eXcursion software. Microsoft had established the Win32 API as a uniform coding interface for all its Windows-based operating systems. Since development tools such as 32-bit compilers and debuggers of sufficient quality and robustness had become available, it was now possible to implement a high-quality, 32-bit product. This product would support the entire range of Windows-based platforms, from notebook PCs running the Windows operating system to high-end Alpha systems running the Windows NT operating system.

Terminology

This paper incorporates certain conventions to clarify the distinction between the two window systems under consideration. *X window* refers to the collection of data structures, concepts, and operations that constitute a window, as defined in the X Window System environment. *Win32 window* refers to a window as defined in Microsoft's Win32 API.

When referring to a window system as opposed to a particular window instance, *X Window System* is sometimes abbreviated to *X*. *Windows* denotes the Microsoft Windows operating system.

Note that the word *bitmap* has more than one meaning. In the X environment, a bitmap is a two-dimensional array of bits, and a *pixmap* is a two-dimensional array of pixels, where each pixel may consist of one or more bits. Under the Win32 API, the term bitmap is used exclusively; that is, no distinction is made between an array of depth 1 and an array of depth *n*. In this paper, the term pixmap is used in its general sense to refer to X pixel arrays, and the term bitmap refers to the Win32 concept.

Another common point of confusion when discussing the X Window System environment is the use of the terms *server* and *client*. To one familiar with file and print servers, the meanings of these two terms in the X environment may seem to be reversed. In the X environment, the server is a display server, and the clients are the applications requesting display services. The X server and the X client applications may reside on the same PC, but the power of the eXcursion software is in its ability to bridge the gap between the Windows desktop and the traditional X11 UNIX and OpenVMS workstations.

eXcursion Version 2 Product Goals

The design of eXcursion version 2 was driven primarily by the following product goals:

- Support X Window System version 11, release 6.
- Support the Microsoft Windows, Windows NT, and Windows 95 operating systems.
- Code the single source pool to Microsoft's Win32 API.
- Exceed graphics performance of eXcursion version 1 as measured with the standard benchmark tests x11perf and Xbench.
- Preserve maintainability by using modular coding and limiting changes of the sample implementation from the X Consortium.
- Maximize reliability by performing extended error checking and resource management.
- Correct known protocol conformance deficiencies in version 1. For example, in version 1, plane mask support was implemented for only a few graphics operations. Version 2 would provide plane mask support for all graphics operations.

Components of eXcursion Version 2

In eXcursion version 1, most of the functions provided by the product were combined in a single executable. To conserve resources and to partition the code for easier maintenance, version 2 is divided into several separate components or modules. Some of these run as individual processes, and some are built as dynamic link libraries (DLLs). A DLL is a shared memory library module that is linked to the calling program at run time.

eXcursion version 2 is partitioned into the following major components:

- X server. The X server is the primary component of eXcursion version 2. The X server process is responsible for displaying windows and graphics on the Windows desktop and for sending keyboard, mouse, and other events to the client application.
- X Image Extension. X extensions are additions to the server that support functionality not addressed by the core X11 protocol, such as displaying shaped (nonrectangular) windows, handling large requests, testing/recording, and imaging. All extensions except the X Image Extension (XIE) are implemented internally in the X server. Because of its size, XIE is implemented as a pair of DLLs, one for XIE version 3 and one for XIE version 5.
- Control panel. As the primary user interface, the control panel provides the user with access to the many configuration settings. It is an independent Win32 application implemented using Microsoft Visual C++ and the Microsoft Foundation Class (MFC) library.
- Interprocess communication library. The interprocess communication (IPC) library is an operating system–independent library used by cooperating processes or tasks to communicate configuration and status information.
- Error logger. The error logger is a simple Win32 application that records error and status information from other eXcursion components in a window, a file, or the Windows NT event log.
- Application launcher. The application launcher is a Win32 application that starts X client applications at the request of the X server or the control panel. The application launcher is invisible to the user.
- Registry interface. The registry interface is an operating system–independent interface to the eXcursion configuration profile. The registry interface is implemented as a Win32 DLL.

X Server

The core of the eXcursion product is the X server, a Win32 application that accepts X requests from client applications and transforms them into graphics on the Windows desktop. The device-independent portion of the server code is ported from the sample implementation provided by the X Consortium. The device-dependent portion treats the Win32 API as the device interface through which client requests are materialized on the screen. The eXcursion X server is illustrated in Figure 1.

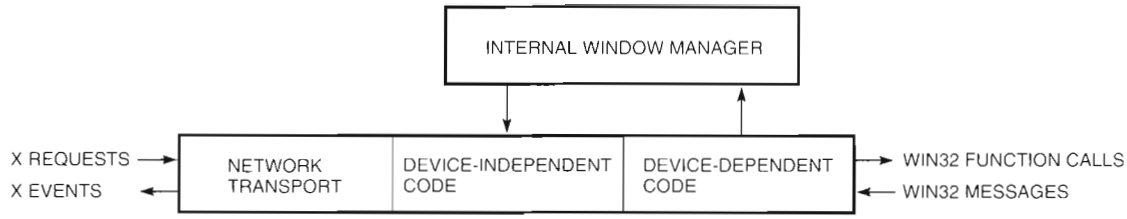


Figure 1
The eXcursion X Server

The server can operate in one of two modes: single-window mode or multiwindow mode. In single-window mode, the server creates one Win32 window, which represents the X root window. All descendant windows and their contents are drawn into the root window using Win32 function calls. In multiwindow mode, the root window is a virtual window; that is, it is never drawn on the screen. Each top-level child of the root window has a corresponding Win32 window, which is created when the X window is mapped. All descendants of a top-level window are drawn inside the Win32 window with Win32 calls. Multiwindow mode thereby creates a desktop environment in which X applications are peers of other Win32 applications.

Single-window mode is useful for emulating a complete workstation environment including the window manager and the session or desktop manager. In multiwindow mode, drawing to and getting input from the root window is restricted by the X server to prevent conflicts with the Microsoft Windows system's use of the desktop window. Despite this restriction, the multiwindow mode, when used with the native window manager, provides the cleanest integration of the X and Windows environments.

Resource Management and Performance

Both the X and Win32 systems have built-in notions of graphics state and resource allocation. The semantics and usage of the concept, however, are quite different in the two window systems.

In X, graphics state is maintained in a data structure known as a graphics context (GC). A GC has an independent existence and may be created, destroyed, updated, queried, and copied at will by the X application. During graphics operations, a GC is associated with the X "drawable" (window or pixmap) being drawn into, and information in the GC is used to fully define the operation. For example, the GC may specify foreground or background colors, line styles, or font information.

The Win32 API has a concept called a device context (DC), which also contains state information but whose purpose is more closely related to providing device independence. Consequently, two different types of DCs are required under the Win32 API, depending on whether the graphics operation is drawing to a window or to a bitmap. Furthermore, a window DC may be allocated either permanently or from a cache, depending on its expected lifetime. Any drawing operation therefore requires that both the GC used in the X graphics request and the DC used in the ultimate Win32 call be properly set up and synchronized. The manner in which this is done has a significant effect on the graphics performance of the server.

Before an X graphics operation can be started, the GC must be validated. Validation is a process of preparing the output device to render the graphics properly. In the case of the eXcursion server, the output device is a Win32 DC. For every graphics command, the GC must be checked for changes and the appropriate Win32 objects and state values must be selected into the DC. This process can be very time-consuming. The key to maximizing performance is to recognize that most operations are repetitive. A typical stream of X requests tends to contain many commands directed at the same window with the same GC. Therefore, the way to reduce GC/DC validation time is to cache the most recent GC/DC pair so that subsequent commands that use the same combination need not trigger a validation step. In some cases, graphics operations will toggle between two or more GCs. (For example, the CopyArea operation takes a source and a destination.) The performance in these cases can be improved by simply caching more than one recent GC/DC pair. Tuning experiments on the server revealed that a cache size between 2 and 4 was sufficient to maximize performance. Under the Windows and Windows 95 operating systems, where resources are limited, a cache size of 2 is used. Under the Windows NT operating system, the cache size is 4.

In the eXcursion server, the notion of a cached GC/DC pair is encapsulated in a C++ class called a WXDC. The WXDC remembers the Win32 objects that have been selected into the DC and the last GC that was used with it. As long as these elements do not change from one graphics operation to the next, no validation is necessary. If the client application changes the contents of the GC, any affected objects in the DC are tagged and the next graphics operation on that WXDC will require new objects to be selected into the DC.

Events in the window system can also cause WXDC elements to become invalid. For example, if the window is moved on the screen by the window manager, its clip list may have changed. This causes the WXDC to invalidate the clip region in its DC. (Clip list and region are defined in the following section.) The next graphics operation on that window will require the clip region to be recalculated and reloaded.

Clipping in Single-window Mode

In the X Window System environment, all descendants of the root window have a clip list, which is a list of rectangles that defines the visible area of the window. The clip list is equal to the area of the child window minus any areas that are occluded by other X windows. Before drawing into a descendant window, the server must convert the clip list into a Win32 region. In the Win32 API, a region is a polygonal area, not necessarily rectangular, that can be selected into a DC for clipping. Before initiating a graphics output operation, the target WXDC checks to see if the current region for the window is valid. If it is not, the X clip list is converted to a Win32 region and combined with the client-supplied clip list in the GC, if any. The result is selected into the output DC.

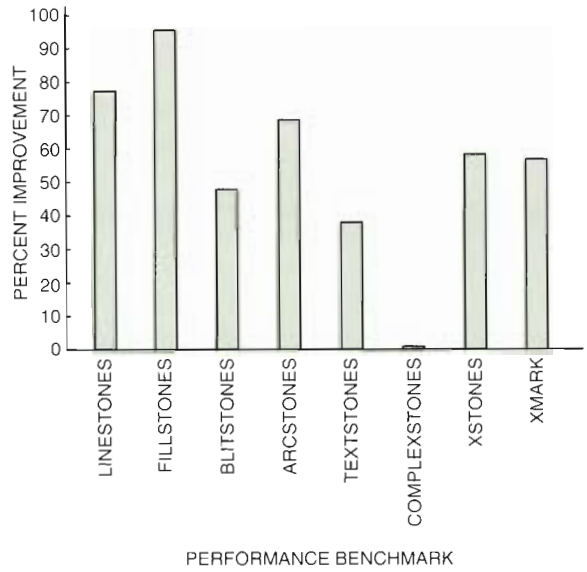
Clipping in Multiwindow Mode

In multiwindow mode, the root window is invisible. Each top-level X window (first-generation child of the root) corresponds to a Win32 window on the desktop. No clipping is necessary for these windows, because Win32 does this automatically. For windows below the first generation, clipping is accomplished in a manner similar to that used in single-window mode, except that the offset of the clip region must be adjusted to be relative to the top-level window instead of relative to the root window.

Graphics Rendering

Graphics rendering is at the heart of the X server. Two of the core goals for the eXcursion version 2 project were to significantly improve server performance over that of the eXcursion version 1 server and to improve server compliance to the X protocol specification. Figure 2 compares the performance of the eXcursion version 2 server with that of the version 1 server. The standard benchmark tests X11perf and Xbench were run over a local area network to eXcursion running on a 66-megahertz Pentium processor with an S3 video card.

The sample X server upon which the eXcursion X server is based provides a machine-independent layer that is capable of rendering all X graphics through a small set of device-dependent functions. In the eXcursion X server, the Win32 functions provide the virtual hardware interface. For maximum performance, X graphics requests are passed to the Win32



Performance Benchmark	eXcursion Version 1	eXcursion Version 2	Improvement
XBench			
lineStones	135,735	239,740	76.6%
fillStones	38,083	74,331	95.2%
blitStones	59,743	88,320	47.8%
arcStones	2,172,720	3,662,770	68.6%
textStones	156,190	214,762	37.5%
complexStones	71,633	71,699	0.1%
XStones	80,057	126,408	57.9%
X11perf			
Xmark	**1.6495**	**2.5805**	**56.4%**

Notes:
The test machine was a DECpc XL 566.

Since eXcursion version 1 did not support 16-bit fonts, the version 2 numbers were substituted to obtain the Xmark number.

Figure 2
Comparison of eXcursion Version 1 and Version 2 Performance

API as early as possible without compromising the requested rendering. Many X graphics requests map neatly into Win32 calls with little or no data manipulation. Some complex graphics requests, however, cannot be practically mapped into high-level Win32 calls and achieve proper pixelization. In such cases, the machine-independent functions are called as helper functions to break the request down into simpler graphics requests.

GDI Context Switching To reduce context switching, Windows batches graphics device interface (GDI) calls. The default GDI batch size is 20, but this limit can be adjusted per thread. Testing with a mix of all X requests showed that an overall performance increase of about 9 percent could be achieved by increasing the GDI batch limit to 30. At this level, there is no measurable latency, and, furthermore, increasing the batch size beyond this point had no measurable benefit.

Some competing X server products set the batch size very high (100) at the beginning of every request and flush the queue at the end. This approach has no measurable benefit over our simpler method, probably because the Windows operating system already performs timer-based flushing to prevent drawing latency.

Similarly, whenever possible, Win32 graphics calls are combined to reduce the overhead of context switching. For example, an X PolyLine request could be rendered with a series of Win32 LineTo calls, but it is much more efficient to render the PolyLine request with a single Win32 PolyLine call. Similarly, a PolyRectangle X request is best rendered with a single PolyPolyLine call.

Solid Fills Many different Win32 resources such as pens, brushes, fonts, and clip regions may be required for any given graphics request. The resources needed are determined by the graphics operation itself and the state of the X GC. As noted earlier, these resources are created as needed and managed by the WXDC objects, removing significant complexity and nearly redundant code from the actual graphics drawing routines.

Windows Pen structures provide color and dash pattern when drawing line objects. For drawing lines, segments, and arcs, the X server creates and uses Pens that correspond to the GC state. In some cases, however, exact pixelization cannot be achieved when using Windows Pens. Examples of this are drawing wide lines with raster operations other than GXcopy or with long, dash patterns. In these cases, machine-independent functions are used to reduce the request to a set of spans (single-width horizontal lines) to be filled. The use of Pens is also abandoned in special cases when the highly optimized GDI pattern block transfer (PatBlt) function can be used. PatBlt fills rectangular regions with specified colors or patterns. It is faster, for example, to use the PatBlt function to draw vertical or horizontal lines than to use the Windows traditional line-drawing functions.

Windows Brush structures provide color and pattern when drawing filled rectangles, filled polygons, and filled arcs. Again, for performance reasons, the PatBlt function is often used even when there is a higher-level function that seems to be a closer match. For example, PatBlt can perform the X PolyPoint request about 10 percent faster than SetPixelV, the Windows standard call for setting single pixel values. Similarly, PatBlt can perform the X PolyFillRect request about 14 percent faster than the Windows FillRectangle call.

Tile and Stipple Fills An X pixmap can be specified as a pattern to be used when performing fill operations. When the pixmap is created, it is realized as a Win32 bitmap. When the pixmap has a depth greater than 1, it is used as a color tile that will be used for the fill. If the pixmap has a depth of 1, it can be used as either a transparent or an opaque stipple. An opaque stipple draws both the GC's foreground and background colors, where the stipple is 1 and 0 respectively. A transparent stipple is similar except that it leaves the destination untouched where the stipple is 0.

When the tile or opaque stipple is 8 by 8 or smaller, a Win32 color brush is created and cached for the drawing. On the Windows NT system, brushes larger than 8 by 8 can be created, but our experience has shown it to be slower to draw with them than it is to perform a series of bit block transfer (BitBlt) operations from the tile/stipple bitmap to the destination.

Transparent Stipple Fills There is a Win32 function, MaskBlt, that seems ideally suited for performing transparent stipple fills. This function, however, was not fully implemented on all platforms at the time we designed the eXcursion version 2 software product. Without this function, there is no easy way in the Win32 environment to perform the transparent stipple operations. When the foreground color is either 0 or 0xFFFF, the raster operation can be remapped to get the proper effect. General rectangular fills that do not meet the requirements of the special case previously mentioned must be accomplished by first converting the stipple bitmap to the depth of the destination and then remapping the raster operation. In general cases that are not rectangular fills, machine-independent functions are called to break down the request into spans.

Image Requests The GetImage and PutImage requests are other examples of X graphics requests that do not map well into the Win32 API. The only way in the Win32 environment to put image data on the screen is to first create a Win32 bitmap and initialize it with the image data, and then call the BitBlt function to copy the bitmap to the screen. X image data always lists the top scan lines first, whereas the bottom scan lines are listed first in Windows bitmap data. Therefore, before the bitmap is initialized, the X image data must be scan-line flipped. Similarly, the X GetImage request requires the use of an intermediate bitmap and also requires the scan-line flip.

Plane Mask Support Any graphics operation in X can be modified by setting a plane mask in the GC. The plane mask specifies which bits of the destination pixel are allowed to be changed. Without a plane mask, an X graphics operation may be defined as

$$dst \leftarrow src \otimes dst,$$

where $\otimes$ is one of the 16 binary raster operations (e.g., OR, AND, and XOR). When a plane mask is given, the following assignment defines the destination pixel:

$$dst \leftarrow ((src \otimes dst) \,\&\, pm) \mid (dst \,\&\, \sim pm)$$

Most video hardware devices support plane masking, and those that do not support it generally provide fast access to video random-access memory (RAM). The Win32 API, however, provides neither plane masking nor direct video RAM access. To understand why, you must realize that Windows has virtualized the color handling in an attempt to mediate conflicts between applications that would otherwise want to modify the colormap (the pixel–to–color mapping table). In this virtual color environment, the concept of plane masks has no meaning because Win32 applications need not know the pixel value that corresponds to a particular color. See the section Color Resource Management for an explanation of how the eXcursion software manages to assign specific pixel values to colors.

In the general plane mask case, it is necessary for the X server to first save the contents of the destination in a bitmap. The graphics can then be temporarily drawn without regard to the plane mask. Those bits in the destination that are specified by the plane mask as being unaffected can then be restored from the saved bitmap. This process will work in every case but is inefficient since it involves several graphics operations before achieving the final result. Many special cases can be reduced to one or two simple steps by modifying the source color and raster operation. Table 1 shows how the source color and raster operation can be set to achieve the plane mask effect. The eXcursion X server uses these optimizations for many graphics operations when the source fill is a solid color.

Internal Window Manager

In the absence of a window manager, the eXcursion server creates all windows as pop-up windows. All windows, including top-level windows in multiwindow mode, are undecorated. They have no Win32 borders, title bars, or system menus. To move, size, minimize, maximize, or close windows, the user must run a window manager.

An eXcursion user always has the option of using one of the many X-based window managers available, such as the Motif Window Manager. However, many users will want a window manager paradigm that is consistent with Windows so that all windows on the desktop have the same user interface. To accomplish this, a built-in window manager is provided as part of the eXcursion server. This internal window manager is operative only in multiwindow mode.

The internal window manager, although linked with the server, is functionally isolated from the rest of the code so that it can easily be disabled. This allows external window managers to be used and also facilitates debugging by allowing problems to be isolated. The window manager creates a "hook" into the server's window procedure, so that all Win32 messages are first examined by the window manager. This gives the window manager the opportunity to act on window management–related messages such as those that indicate a change in the window's configuration or state. If the window manager decides to handle a message, it is removed from the queue, and the server never sees it. If the window manager is not interested, the message is passed on to the normal window procedure.

The purpose of the internal window manager is to give X windows the same appearance and behavior as Win32 windows that are created by typical desktop applications, such as word processors and spreadsheets. When an X window is mapped for the first time, the internal window manager receives a Win32 WM_CREATE message. Before the window becomes visible on the screen, the window manager alters the style of the Win32 window to WS_OVERLAPPEDWINDOW. Win32 windows with this style are automatically managed by Windows, which handles moving, resizing, iconifying, maximizing, and closing the windows. Each of these actions causes a corresponding message to be sent to the server's window procedure. The internal window manager intercepts the messages and dispatches them to the appropriate internal function.

The role of the internal window manager complements the role of the server. The server processes client requests on X windows and translates them into operations on Win32 windows. The internal window manager handles Windows messages that indicate changes to a Win32 window and translates them into corresponding changes to the underlying X window. For example, the most important message that the window manager handles is WM_WINDOWPOSCHANGING. This message is sent just before any change in the window's position, size, stacking order, or visibility. If this message indicates that the window size changed, the window manager changes the size of the corresponding X window and sends a ConfigureNotify event to the client. Similarly, the window manager translates other user-directed events such as focus change, window stacking, and iconification into changes to the underlying X data structures. In most cases, the window manager does this by calling into the device-independent layer, thus simulating an X request that would occur from an external window manager.

Mouse, Keyboard, and Input Focus

Mouse actions and keystrokes are received by the eXcursion server as Win32 messages. Each message contains information about the window that received the input and the time of the input. For mouse moves and clicks, the server uses the window information to locate the corresponding X window and forwards an X event to that window. Keyboard input is forwarded to the window that currently has X focus.

Table 1
Plane Mask Optimizations

Requested X Raster Operation	src 0 dst 0	0 1	1 0	1 1	Notes	Modified Source Color and Raster Operations
GXclear	0	0	0	0	4	src ← ~pm, rop ← and
GXand	0	0	0	1	1	src ← src | ~pm
GXandReverse	0	0	1	0	6	src ← src | ~pm src ← ~pm, rop ← xor
GXcopy	0	0	1	1	8	src ← ~pm, rop ← and src ← src & pm, rop ← or
GXcopy (src & pm) = pm	0	0	1	1	8	src ← pm, rop ← or
GXcopy (src & pm) = 0	0	0	1	1	8	src ← src | ~pm, rop ← and
GXandInverted	0	1	0	0	2	src ← src & pm
GXnoop	0	1	0	1	10	—
GXxor	0	1	1	0	2	src ← src & pm
GXor	0	1	1	1	2	src ← src & pm
GXnor	1	0	0	0	7	src ← src & pm src ← ~pm, rop ← xor
GXequiv	1	0	0	1	1	src ← src | ~pm
GXinvert	1	0	1	0	5	src ← pm, rop ← xor
GXorReverse	1	0	1	1	7	src ← src & pm src ← ~pm, rop ← xor
GXcopyInverted	1	1	0	0	9	src ← ~pm, rop ← and src ← ~src & pm, rop ← or
GXorInverted	1	1	0	1	1	src ← src | ~pm
GXnand	1	1	1	0	6	src ← src | ~pm src ← ~pm, rop ← xor
GXset	1	1	1	1	3	src ← pm, rop ← or

Notes:

1. dst is unchanged when src equals 1 for these raster operations. Therefore, to preserve the value of dst when pm equals 0, set src equal to 1.
2. dst is unchanged when src equals 0 for these raster operations. Therefore, to preserve the value of dst when pm equals 0, set src equal to 0.
3. This operation sets all dst bits to 1 except where the plane mask equals 0. This can be done simply by ORing pm into dst.
4. This operation clears all dst bits except where the plane mask equals 0. This can be done simply by ANDing pm into dst.
5. XORing with 1 has the effect of inverting. To invert only where pm equals 1, XOR pm with dst.
6. These operations are performed in two steps. Note that dst is inverted when src equals 1. First perform the operation with src set to 1 where pm equals 0. dst is now correct except that it is inverted where pm equals 0. The second operation of XORing with the invert of pm corrects this.
7. These operations are performed in two steps. Note that dst is inverted when src equals 0. First perform the operation with src set to 0 where pm equals 0. dst is now correct except that it is inverted where pm equals 0. The second operation of XORing with the invert of pm corrects this.
8. This operation is performed in two steps. First dst is set to 0 whenever pm equals 1. Then dst is set to 1 whenever both pm and src equal 1. The two special cases can be reduced to operations that use GXset and GXclear.
9. This operation is performed in two steps. First dst is set to 0 whenever pm equals 1. Then dst is set to 1 whenever pm equals 1 and src equals 0.
10. dst is unchanged; therefore, no operation is required.

The X server is a single application in the Win32 environment that "owns" all the X windows it creates. From the user's perspective, though, there may appear to be more than one X application running, each with its own collection of windows. The user expects to be able to shift the keyboard focus from one window to another in the same fashion that focus is shifted between other applications. When an external window manager is in use, focus control is straightforward. The window manager, using whatever semantic it was designed for, monitors mouse events and shifts focus accordingly. However, the semantic model for this may or may not be consistent with the Win32 model. In either case, the window decorations, e.g., borders, title bars, and menus, are almost guaranteed to be different. A user who wants a consistent user interface model across all applications must employ the internal window manager.

At any given time, one window on the screen has Win32 focus and one X window has X focus. The two windows are not necessarily the same. Since the X server creates and owns all the X windows in use, the server receives keyboard input when any one of its windows has Win32 focus. The keystrokes are not necessarily sent to the underlying X window, however. They are sent to the window that has X focus. The internal window manager assigns X focus to the X window that receives Win32 focus. The client receives notification of this event and may decide to assign X focus to some other window, perhaps a child window.

The server must therefore keep track of both the X window that currently has focus and the state of Win32 focus. When the server loses Win32 focus, the X focus is assigned to the root window. When the server receives Win32 focus, X focus is assigned to the X window that previously had it. Whenever X focus is changed by an application or by the window manager, the current X focus state is cached so that it can be restored later, if necessary.

Font Management

Fonts and text functionality make up a significant portion of any graphics architecture. Both the X and the Win32 systems define a rich set of text-rendering operations and can process several font formats.

X and Win32 Fonts The X font management library is a modular architecture that defines an API for reading and writing individual font formats. The module that implements the API for a given font format is called a renderer. This approach allows X to support several font formats: the library's renderer modules convert external formats to a single, internal bitmap format, which is used for all drawing operations. The term *X font* refers to font data in this internal format.

The font management library supports both bitmap and scalable outline fonts. Bitmap font glyphs are simply reformatted and used. Scalable formats, such as Adobe Type1, are rasterized on demand into the X font format.

For maximum performance, the server draws text with native Win32 fonts using the Win32 API. Win32 fonts are bitmap fonts in the FON format. Win32 functionality covers the great majority of text-drawing operations, but there are a few cases in which it is either not possible or not efficient to use Win32 fonts.

The server can also draw directly with the X fonts to provide full X font support and complete text-drawing functionality. This method uses Win32 BitBlt( ) operations to copy the character glyphs to the display as bitmaps. Drawing speed with this method is acceptable but not maximum.

Therefore, both X and Win32 fonts are used. The Win32 fonts may be thought of as optional accelerators: the server uses them whenever possible and falls back to the X fonts when necessary. The decision to fall back can be made on a variety of conditions. This technique has also proved useful in working around problems such as text-drawing bugs in individual video drivers.

Since scalable font outlines are rasterized into bitmaps at run time, they are generally drawn directly with the internal X font format. The extra work of compiling a companion Win32 font at run time generally outweighs its value as an accelerator.

X bitmap fonts are most commonly distributed in the Bitmap Distribution Format (BDF), an ASCII text source file. The eXcursion team wrote a font compiler tool that generates native Win32 (FON format) fonts from the BDF sources. The fonts created can be used by any Win32 application.

The compiler can generate either the commonly used version 2 format or the extended version 3 format, which is necessary for large fonts that require more than 64 kilobytes (KB) of glyph storage. Figure 3 illustrates the process of generating equivalent X and Win32 fonts from a common source.

The X font format contains extra information (e.g., metrics and properties) that cannot be derived from

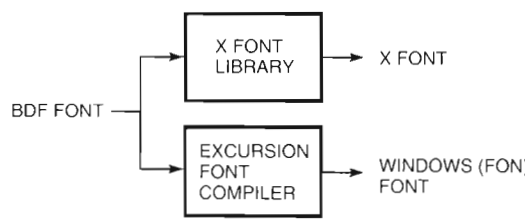


Figure 3
Font Conversion

the Win32 font. Therefore, the X and Win32 fonts are used together; the X information comes from the X font and the Win32 font is used by the Win32 API.

Realizing Win32 and X Fonts When the X server first opens a font, it invokes the function RealizeFont( ). This function gives the server an opportunity to initialize data structures and perform any format-specific operations necessary to make the font available.

To make a Win32 font available for drawing, the server retrieves the filename of the font from the server's look-up table and registers it with the Win32 API using the function AddFontResource( ). A handle to the font is obtained from CreateFontIndirect( ), and thereafter the handle is selected into the desired DC for drawing operations. If the Win32 realization fails for any reason, the code simply realizes the X font instead. Failing to realize a Win32 font does not necessarily imply an error condition. Such failure happens in any case in which the server decides that it is best to use the X font directly.

The internal X font format is a set of data structures. The glyphs are stored in conventional arrays in user memory. To improve performance, the server realizes an X font by writing all glyphs to a Win32 bitmap in off-screen memory. CreateBitmap( ) returns a handle for later reference, and the glyphs in the bitmap are indexed for use in drawing operations.

Drawing with Win32 and X Fonts The glyphs in X text strings are often kerned, that is, overlapped for best typographic appearance. To draw with Win32 fonts, the server emulates the way X draws text by using ExtTextOut( ), which uses an intercharacter spacing vector to place the individual glyphs. The font's X metrics are used directly to calculate this vector.

Glyphs from X fonts are drawn by performing BitBlts from the Win32 bitmap to the target window or bitmap. The server places the glyphs using the font's X metrics as described in the previous paragraph.

Color Resource Management

Although some X Window System concepts and structures map fairly closely to those in the Win32 system, color resource management is handled very differently. The difference is most evident when dealing with pseudocolor video systems. Consequently, this paper describes only this case.

The X Window System environment shares 256 colormap cells among all applications that use the default colormap (i.e., those that do not have a private colormap). Applications can allocate cells in the default colormap to protect them from modification by other applications. In contrast, the Win32 system allows each application complete access to the system palette while the application has focus and maps the palettes of the windows without focus as best it can.

In the X Window System environment, when an application reserves a colormap cell, it references the cell with a pixel value. This value is an index into the colormap and is used to look up the value that will actually be stored in screen memory when that pixel value is used in a drawing operation.

In the Win32 system, color management is handled by the palette manager through a palette structure. Each application has a logical palette, and a single system palette contains the colors currently mapped to the hardware colormap. Applications reference colors relative to their logical palette, and the palette manager handles the mapping between the logical palette and the system palette. When an application is given focus, the palette manager maps all the colors from the logical palette into the system palette. If the system palette does not have enough empty cells, the palette manager frees cells allocated to other applications. If this occurs, the palette manager will attempt to remap the other applications' colors into any remaining free cells in the system colormap. If not enough cells are free, any remaining unmapped colors are mapped to the system palette colors that most closely match.

Because of this way of handling color resource management, an application does not know what value is being stored in screen memory for any particular color and the value stored for any color can change over the lifetime of the application. This situation presents significant difficulties for X operations that require exact knowledge of the pixel values in screen memory, such as the GetImage operation and operations involving plane masks. The server works around the difficulties by creating two Win32 logical palettes.

The first palette, i.e., the working palette, corresponds exactly to the X default colormap and does not allow sharing of the palette by Win32 applications. Whenever an X window has focus, the working palette is in use. This causes the Win32 palette manager to set up the system palette such that it directly corresponds to the X colormap, and operations that are pixel based work properly.

The other palette, i.e., the identity palette, is set up to correspond exactly to the system palette. The identity palette is used whenever no X window has focus. Because of the correspondence, no translation is involved between the identity palette and the system palette, which allows the X server to know what pixel value is stored in screen memory.

The X Window System environment allows for private colormaps, which are created and used by a single application. The server creates a working palette for every colormap created. When the colormap is installed (normally by the window manager when the X application is given focus), the eXcursion software installs the working palette associated with the private colormap.

The eXcursion X server currently supports the PseudoColor visual class and the StaticGray depth 1

visual class, which is mainly used for bitmaps. eXcursion version 1 also supported a StaticColor visual class for 16-color video graphics array (VGA) displays. eXcursion version 2 treats VGA devices identically to PseudoColor devices and allows the Windows palette manager to generate dithering patterns for the unavailable colors.

Network Interface

With the release of X11R6, the X Consortium combined all transport-specific code into a single place in the source tree, the X transport interface. The eXcursion team extended the X transport interface to include Network Computing Device's (NCD's) Xremote serial line transport. Combined with the transmission control protocol/internet protocol (TCP/IP) and DECnet transports, the eXcursion product can now execute X sessions over any of these transports simultaneously. The eXcursion product supports any TCP/IP stack that complies with the Winsock version 1.1 implementation, PATHWORKS DECnet protocol, and NCD's Xremote protocol for serial line.

The X transport interface provides functions that are common to all transports, such as parsing an address into a host and port number. The interface does not provide an abstraction for the select() call, because it assumes that this call is transport independent. Unfortunately, the Xremote protocol requires an independent select() mechanism, and, thus, it was necessary to implement a select() abstraction to combine the transport-independent select() with the Xremote select(). Although somewhat compromised by this addition, performance was a problem only when the Xremote protocol was used in combination with either the TCP/IP or the DECnet protocol.

X Image Extension

eXcursion version 2 provides versions 3 and 5 of the X Image Extension to support a wide range of imaging applications. Because it is a large body of code, XIE is implemented as a pair of Win32 DLLs to conserve memory on systems that will not be running applications that use XIE.

Normally, access to a DLL is one-way. Applications can load and make function calls into a DLL, but because it is linked dynamically at run time, the DLL code cannot make function calls back into the calling application. XIE, however, must call into the device-dependent layer of the server to perform any required drawing after processing its imaging requests. To permit this, an addition to the interface was designed. When the XIE DLL is initialized, the caller supplies a list of pointers to the functions needed by the XIE. The DLL fills an array with these pointers and then calls back indirectly through the array. On the Windows operating system, this design could create a problem because under Win32 APIs, global data in a DLL is not instanced; that is, the code is not reentrant. The approach works in this case because there is only one copy of the DLL loaded. If another application was sharing the DLL, the pointers would be overwritten by the second initialization.

Control Panel

The eXcursion control panel is the primary interface through which the user configures and controls the product. Some other components create simple windows or icons, but these functions are limited. The control panel constitutes 90 percent of the user interface for the eXcursion application. This fact makes the control panel an ideal candidate for the rapid application development features of the Microsoft Visual C++ environment. The control panel is a Win32 application coded almost entirely in C++ and linked with the Microsoft Foundation Class library.

The main purpose of the control panel is to present a manageable interface through which the user can view and modify the eXcursion configuration profile. To do this in a manner consistent with the new Windows 95 shell, the Property Sheet MFC object was chosen. Property Sheets are tabbed dialog boxes that have the advantage of organizing large amounts of data settings in a compact space. They are used extensively by the Windows 95 operating system and by the most recent versions of Microsoft applications.

The Property Sheet object is a subclass of the Windows object and is essentially a container for the tabbed pages. Each tab, when clicked by the user, displays a dialog box that is subclassed from the MFC Property Page object. The individual pages can be visually configured and revised using the class wizard feature of Microsoft Visual C++. The designer simply selects dialog box controls such as buttons, drop lists, or edit fields and positions them on the dialog box. The code to handle user actions is then filled in.

The eXcursion control panel is shown in Figure 4. We constructed an initial prototype of the control panel application with about 60 percent of the final functionality in less than one month.

Interprocess Communication Library

eXcursion version 2 consists of several cooperating processes that must communicate and synchronize with one another. When a remote X application is started by the server or the control panel, the application launcher signals when the operation is complete.

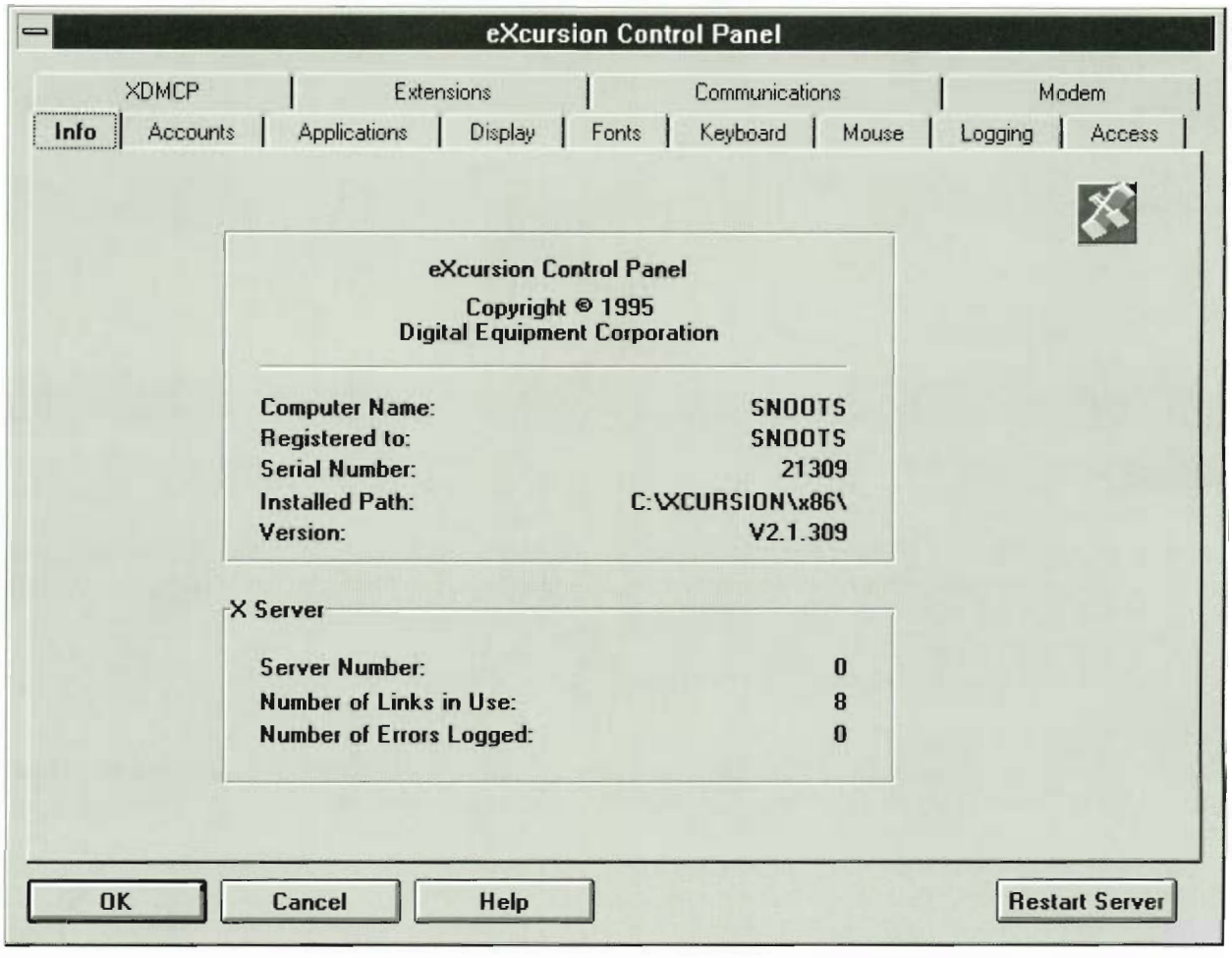


Figure 4
The eXcursion Control Panel

Error and status information is sent to the error logger by the other components. When the user changes a configuration setting through the control panel, the change must be communicated to the X server, if it is running. In some cases, the change can take effect immediately; in other cases, the server cannot implement the change without restarting. The control panel and the server must engage in a dialog so that the user can be informed as to what action must be taken, if any. The IPC library is an operating system–independent API that permits eXcursion components to determine which other components are present and to exchange commands and configuration information.

The Windows NT operating system provides several built-in IPC mechanisms, but most are not available on the Windows or Windows 95 systems. The only mechanism that is universal to the three operating systems is the message-passing interface in the Win32 API. This mechanism, while not the most efficient, is relatively straightforward to implement. Since the performance demands on the IPC library were determined to be very light, this mechanism was chosen.

The disadvantage of the Win32 message-passing interface is that it is window based, not process based. Messages are received by a callback procedure that must be associated with a window before any communication can take place. If an application has not yet created a window, or never creates a window, as is the case with the application launcher, no communication is possible. To remedy this, the IPC library creates its own window when the calling process initializes. The IPC window is never mapped to the screen, so it is not visible to the user. All interprocess communication passes through the IPC window.

The IPC library consists of a collection of unique messages and an API. The messages are registered with the Win32 function RegisterWindowMessage. This ensures that the messages used by the eXcursion application do not conflict with system messages or messages used by other applications. The eXcursion IPC messages are

- ipcComponentStartedMsg, which the IPC posts to all components when a component initializes.

- ipcRestartServerMsg, which the IPC sends to the server to tell it to restart.
- ipcRestartServerStatusMsg, which the IPC posts with the status of the restart request.
- ipcInquireMsg, which the IPC sends to retrieve a data item from a component.
- ipcProfileChangedMsg, which the control panel sends when the registry profile changes.
- ipcLaunchOneCompleteMsg, which the application launcher sends to notify the server of launch completion.
- ipcLaunchAllCompleteMsg, which the application launcher sends to notify the server of launch completion.
- ipcHideAllWindowsMsg, which the server sends to all components to tell them to hide all their windows. The eXcursion application uses this message to execute the pause/resume feature.
- ipcShowAllWindowsMsg, which the server sends to all components to tell them to show all their windows. The eXcursion application uses this message to execute the pause/resume feature.

In addition to sending and receiving messages, eXcursion processes can use the IPC library to determine which other components are running. The IPC initialization procedure creates a window with a unique name that identifies the calling component. To determine whether a specific component is present in the system, the IPC searches all windows on the system until it finds one with the correct name.

Error Logger

The error logger is a Win32 application that receives error and informational messages from other components and either displays them in a window or logs them to a file. On the Windows NT operating system, information that may help system managers or users diagnose problems may additionally be recorded in the Windows NT event log.

Application Launcher

The application launcher is a Win32 application that handles requests from the control panel or server to start X client applications. The client may reside on a remote host or on the same machine.

When the user requests the server or control panel to start an X client application, it starts the eXcursion application launcher in a separate process. The application command, host name, account information, network transport, and command shell are passed to the launcher in its command line arguments. The launcher makes the connection to the remote system, initiates the command using the selected protocol (rexec, rsh, DECnet object, or local command), and sends an IPC message to the server indicating that a new application is starting.

Registry Interface

The Windows NT operating system introduced a new concept called the registry. This is a protected database maintained by the operating system, wherein Win32 applications may store configuration and state information. The registry has a well-defined API and a maintenance utility program that is shipped with the Windows NT operating system. Under the Windows operating system, configuration information is kept in simple text files, which are vulnerable to accidental or malicious tampering. At the time the design of eXcursion version 2 was under way, it was unknown which, if either, of these two methods would be available under the Windows 95 operating system. Nevertheless, all three of these operating systems had to be supported.

We designed an API for accessing the configuration information in a manner independent of the operating system. Knowledge of the operating system and its registry access method is encapsulated in the library. Since several independent processes must access the information, the library is built as a DLL to conserve memory. The interface basically resembles that of the Windows NT registry API but eliminates some of the complexity.

If the eXcursion software has not been configured when the registry interface first accesses the profile, default values for all settings are selected to allow the software to function normally.

Summary

With computer systems based on the Microsoft Windows operating system increasing in power and decreasing in price, Windows-based systems are appearing on desktops that once held workstations running the UNIX or OpenVMS operating systems. Windows systems must be able to access applications on remote file and compute servers running in the X Window System environment. Version 2 of the eXcursion product provides desktop integration of X client applications with native Win32 applications. Modular coding techniques, object-oriented programming, and selective use of the Microsoft Foundation Class library helped reduce development time, and improve performance, maintainability, and reliability.

General References

D. Giokas and A. Leskowitz, "eXcursion for Windows: Integrating Two Windowing Systems," *Digital Technical Journal*, vol. 4, no. 1 (Winter 1992): 56–67.

X Window System

S. Angebranndt et al., *Definition of the Porting Layer for the X v11 Sample Server* (Cambridge, Mass.: X Consortium, Inc., 1994).

J. Fulton, *The X Font Service Protocol, Version 2.0, X Version 11, Release 6* (Cambridge, Mass.: X Consortium, Inc., 1994).

E. Israel and E. Fortune, *The X Window System Server, X Version 11, Release 5* (Woburn, Mass.: Digital Press, 1993).

O. Jones, *Introduction to the X Window System* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1989).

K. Packard and D. Lemke, *The X Font Library* (Cambridge, Mass.: X Consortium, Inc., 1995).

D. Rosenthal, *Inter-Client Communication Conventions Manual, Version 2.0* (Cambridge, Mass.: X Consortium, Inc., 1994).

R. Scheifler, *X Window System Protocol, X Version 11, Release 6* (Cambridge, Mass.: X Consortium, Inc., 1994).

R. Scheifler and J. Gettys, *X Window System* (Bedford, Mass.: Digital Press, 1992).

Networks

M. Hall et al., "Windows Sockets: An Open Interface for Network Programming under Microsoft Windows, Version 1.1" (1993).

K. Packard, *X Display Manager Control Panel, Version 1.0, X Version 11, Release 5* (Cambridge, Mass.: MIT X Consortium, 1989).

W. Stevens, *UNIX Network Programming* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1990).

X Transport Interface (Dayton, Ohio: NCR Corporation, 1993).

Windows Operating Systems

R. Blake, *Optimizing Windows NT, Windows NT Resource Kit*, vol. 3 (Redmond, Wash.: Microsoft Press, 1993).

H. Custer, *Inside Windows NT* (Redmond, Wash.: Microsoft Press, 1993).

A. King, *Inside Windows 95* (Redmond, Wash.: Microsoft Press, 1994).

Win32 Programmer's Reference, vols. 1–5 (Redmond, Wash.: Microsoft Press, 1995).

Windows Programming

K. Christian, *The Microsoft Guide to C++ Programming* (Redmond, Wash.: Microsoft Press, 1992).

P. DiLascia, *Windows++: Writing Reusable Windows Code in C++* (Reading, Mass.: Addison-Wesley Publishing Company, 1992).

The GUI Guide, International Terminology for the Windows Interface (Redmond, Wash.: Microsoft Press, 1993).

S. McConnell, *Code Complete: A Practical Handbook of Software Construction* (Redmond, Wash.: Microsoft Press, 1993).

C. Petzold, *Programming Windows*, 2d ed. (Redmond, Wash.: Microsoft Press, 1990).

B. Stroustrup, *The C++ Programming Language* (Reading, Mass.: Addison-Wesley Publishing Company, 1986).

Biographies

John T. Freitas

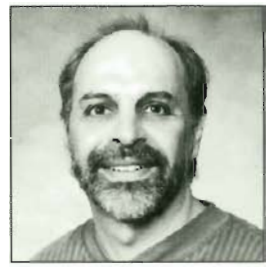
Presently a software engineer at Atria Software, John Freitas worked at Digital for 15 years. For the last few years, he was associated with Digital's eXcursion product as an individual contributor, an architect, and a designer. Previously, he was in the Workstation group. John received a B.S.E.E. from Northeastern University in 1967. While in college, he worked as a co-op student on the Apollo Project at MIT's Draper Laboratory. During the 1970s, he worked for Harvard University developing and maintaining medical computing facilities at Massachusetts General Hospital.

James G. Peterson

James Peterson is currently a software engineer at DeLorme Mapping. As a member of Digital's Windows NT group, James led the releases of the eXcursion software from version 1.1 through version 2.1. In addition, he worked as architect and individual contributor on the eXcursion project, concentrating on graphics and performance. Earlier, he worked in the PATHWORKS and Rainbow groups. James was employed by Compion Corporation before joining Digital in 1984. He received a B.A. (1979) in mathematics from Indiana University and an M.S. (1981) in mathematics and an M.S. (1984) in computer science, both from the University of Illinois.

Scot A. Aurenz
Scot Aurenz is a principal software engineer in the Windows NT group where he works on the development of the eXcursion PC X server. Scot has contributed to many projects at Digital, including the Language Sensitive Editor (DECset LSE) and the SUVAX workstation. Scot came to Digital in 1979 as a Purdue University co-op student and became a full-time employee after receiving his B.S.E.E. in 1982. He received an M.S.E.E. from the University of Illinois in 1986.

Charles P. Guldenschuh
Charles Guldenschuh is a principal software engineer in Digital's Windows NT group. He is responsible for color support and software installation of the eXcursion product. Previously, he worked in the Real-Time Software, Professional 300 Software Engineering, and RT-11 Engineering groups. Charles joined Digital after receiving his B.S. in information and computer science from the Georgia Institute of Technology in 1976.



Paul J. Ranauro
Paul Ranauro joined Digital in 1987 and is a principal software engineer in the Windows NT group. He is responsible for application failover for the Digital Clusters for Windows NT product. In earlier work, he participated in the development of the eXcursion software and the ACMSxp transaction processing monitor, specifically, in the implementation of the RTI protocol. He also participated in the implementation of the Manufacturing Messaging Service OSI application layer protocol for the DEComni product and a network performance analyzer. Prior to coming to Digital, he was a consultant at Index Systems and a senior software engineer at Micom-Interlan. Paul holds a B.A. in history from the University of Massachusetts at Boston.

Integrating Multiple Directory Services

Margaret Olson
Laura E. Holly
Colin Strutt

The Integrated Directory Services (IDS) infrastructure implements a directory-service-independent interface. The IDS infrastructure is used by applications that store and retrieve information about resources in environments with either multiple directory services or one of several directory services. The IDS interface isolates users and application writers from the unique requirements of different directory services by providing a view of a single, logical directory service through a simple federation mechanism. To retrieve resources from the logical directory, IDS determines its physical location and converts the resource from a directory-specific to a canonical format. Extensible schema tables represent the canonical format for each resource and allow IDS to represent resources created using both the IDS interfaces and the directory-specific interfaces.

Digital has developed the Integrated Directory Services (IDS) technology to provide a mechanism for integrating multiple directory services into a single system. In this paper, we examine the development of the IDS infrastructure. We begin by discussing the problems faced by network directory applications. Next we describe our design goals, the IDS infrastructure, and our initial implementation on the PATHWORKS product. We conclude with a brief discussion of plans for future development.

Directory Support in Multiple Environments

Although directory services are a powerful mechanism for distributing and accessing certain kinds of information, relatively few applications choose to use them. Digital's PATHWORKS application was in need of a directory for printers and file shares. PATHWORKS is a network operating system (NOS) integration product that gives users access to both Microsoft's LAN Manager and Novell's NetWare file and print shares. As we studied how to incorporate directory support into PATHWORKS, we came to a better understanding of the problems faced by directory applications in general.

Networks are growing rapidly, as are the amount and kind of information that can be accessed through the network. We were certain that future network application products would have an even greater need for a directory, and therefore a general solution was needed. We then set out to design a system that would remove the barriers to directory service application usage and deployment. We resolved the tension between the product deadline and the time required to implement the general solution by designing a complete solution and implementing what was necessary to prove the design and to meet the immediate needs of the PATHWORKS product.

Existing Directory Services

There are a number of general-purpose directory services. Some of the more familiar include X.500, Novell's NetWare Directory Service (NDS), the Cell Directory Service (CDS), and Banyan Systems'

StreetTalk.[1–4] In the past, directory services were in relatively limited use because most directory services were tied to either an operating system or a transport or both. In addition, directory services were connected to a multitude of application programming interfaces (APIs) that were incompatible and difficult to use. More recently, directory services have been tied to network operating systems or applications, rather than to host operating systems or transports. If anything, the number of "standard" APIs has grown.

In large networks, this complexity has resulted in the proliferation of directories, often containing overlapping information. This makes the network manager's job difficult, which in turn creates resistance to directory applications. At the same time, network and NOS technology has developed to a point where an ever-increasing amount of information is being shared on different machines. To give a simple example, almost every server at Digital's Littleton site has a connection to the high-volume printer in the copy center, with a different name on every server. A directory would simplify users' access to this single physical resource by presenting a single name for the printer, if only the application writer could figure out which directory service to use and how to use it.

Other Approaches

As discussed later in the Design of the IDS Framework and Service Providers section, IDS defines both an API and a service provider interface. Support for any directory service can be provided by writing a service provider module. Microsoft's OLE Directory Services (OLE DS) takes a similar approach to IDS, with a more limited initial implementation.[5] Although the current IDS implementation runs under Microsoft Windows, it was designed to port to other systems. OLE DS depends on features of the Windows operating systems.

The X/Open Federated Naming (XFN) specification was not complete at the time we were designing IDS, and it did not include either a service provider interface or a reference implementation.[6] We did examine the XFN draft and designed the IDS interface to be compatible with XFN, with a view toward supporting the XFN API in the future. Supporting the XFN interfaces on top of IDS would be a relatively straightforward task, and we have considered doing this.

The PATHWORKS Application

In the NOS environment, each NOS has its own directory or pseudo-directory. NetWare version 3 implements the Bindery; NetWare 4 implements NDS.[7] The various implementations of Microsoft's LAN Manager protocols provide a virtual directory based on information maintained by its domain controllers. In a multiple NOS environment, the user is presented with multiple information sources from the multiple directories. Even worse, the user may be faced with multiple information sources even in a single NOS environment, since there may be multiple NetWare Binderies or LAN Manager domains.

Multiple NOS environments do not, in and of themselves, cause complexity and confusion. Problems arise when people within a single environment want to share resources across multiple environments. For example, consider a common local area network (LAN) configuration where NetWare is installed on the clients and servers for one department and Microsoft's LAN Manager (contained within products such as Microsoft's Windows for Workgroups, Windows 95, and Windows NT operating systems, or the LAN Server product from International Business Machines Corporation) is installed on the clients and servers for another department. If each department's resources, users, and administration personnel are kept distinct, there is no problem. However, any desire to allow users to share resources between departments, or to have common administration over the departments introduces administrative and user problems. If a printer is to be shared by the two departments, it must be administered twice: once in the NetWare environment and once in the LAN Manager environment. Users in the two departments use different names for the same printer. Later NOS implementations, such as Digital's PATHWORKS version 5.0 or the networking software built into Microsoft's Windows 95 that provides support for multiple NOS protocols, do nothing to manage the multiplicity of names for the same network resource.

As we were contemplating the set of capabilities we needed to design for the next generation of PATHWORKS client products, we realized that solving the connectivity problem implied in a multiple NOS environment was not enough. User access and administrator control of NOS resources needed to be considerably simpler.

As we looked at the problems in larger networks, we saw the need for the ability to provide more sophisticated means to locate NOS resources. Typically, NOS client software provides the means to browse the network to locate a resource. However, browsing requires the user to know the location of the resource, specifically the name of the server, and to be able to choose the resource on the server by recognizing something about the resource name or a resource description provided by the administrator. What was needed was a design that allows a user to search, as well as browse, for a resource based on various attributes describing the resource.

Finally, existing NOS environments have a fairly limited view of the set of resources that can be referenced.

Both NetWare and various LAN Manager implementations provide support for printers and file shares. We wanted to be able to extend the types of resources that could be referenced and managed from the new directory capability that we were designing.

Thus we embarked on a design for the facility we initially called IDS, for Integrated Directory Services. The PATHWORKS version 6.0 implementation was eventually called Directory Assistant. We refer to this technology as IDS throughout this paper.

Design Goals

As we looked at the requirements of the PATHWORKS product, we found that many of those requirements could technically be met with any directory service that was integrated into the PATHWORKS applications and tool sets. PATHWORKS required the ability to

- Give a single name to resources that can be accessed by means of multiple servers or protocols
- Insulate end users from changes in the way resources are allocated among the servers
- Manage resources in an NOS-independent manner

We could not simply pick a directory service and integrate it into PATHWORKS, because we could not require that all customers deploy a particular directory service at their site. The PATHWORKS product is both NOS- and transport-independent; introducing such a dependence was unacceptable. We quickly realized that these were the requirements that kept many other applications from using directory services.

Our assumption was that many network applications would use directory services if they could, but that few of them could assume or require a particular directory service. Working from that assumption, we selected the following design requirements for IDS:

- Directory service independence
- Ability to access existing data
- Ability to join disparate namespaces into a single, logical namespace
- Removal of barriers to successful deployment of a wide area network (WAN) directory
- Ability to hide directory name syntax
- Support of search
- Support of application-specific directory entries

Directory Service Independence

Customers must be able to choose the directory service in which they store resource information. Some customers have a preferred directory service, which they want to continue to use. Other customers, who are not using a particular directory service, prefer that Digital provides the directory service. In a few cases, a customer might wish or even need to store information about different resources in different directory services.

Ability to Access Existing Data

A great deal of information currently exists in application-specific directory services and in NOS-specific directory services. A relatively large number of applications also use the native interfaces to store information in the NOS directories. Allowing users to access this information directly through IDS was critical. We expressly wanted to avoid the need to duplicate directory information in separate, incompatible systems.

Ability to Join Disparate Namespaces into a Single, Logical Namespace

Many directory services are aimed at a specific application or a set of applications. For example, current X.500 deployments contain mostly people information such as names, phone numbers, and electronic mail addresses. (Note: X.500 is an extremely flexible directory service that can be used to store almost any kind of information, but for historical reasons most deployments contain people information.) NOS directories contain information about NOS resources such as printers. Consequently, many user environments have multiple directory services, each of which contains critical business information. To access this existing data and present it to the user in a meaningful way, these multiple directory namespaces must be joined into a single, logical namespace.

Removal of Barriers to Successful Deployment of a WAN Directory

Hierarchical directory services generally require that the naming hierarchy be designed before the directory is deployed. Since the hierarchy consists of names, and names are sensitive and political entities, this can be an extremely difficult task. Organizations also change over time, further complicating the problem of designing a name hierarchy.[8]

Organizations that successfully deploy directory services do so from the bottom up. The NOS directories are deployed precisely because they avoid the problems inherent in a name hierarchy. An administrator can set up a Novell 3.x Bindery for a local organization without worrying about how the name of one group relates to the names of all the other groups. The downside to the NOS directories is that they have a limited ability to scale beyond a LAN. With IDS, we wanted to provide a framework that would grow with the user's environment. A user could start with a local directory but incorporate that directory into an enterprise or global directory when the time was appropriate, without affecting the end users or the applications.

Ability to Hide Directory Name Syntax

The syntax of the names in hierarchical directory services varies not only from one directory service to another, but in some cases from one implementation of a single directory service to another. The syntax for Domain Name System names is ordered the same as a postal mail address, that is, from the most-specific component.[9,10] For example, a machine at Digital might be bigAlpha.digital.com. The X.500 name order is usually (depending on the implementation) the reverse. The corresponding X.500 name might be: c=us;o=Digital;cn=bigAlpha. Particularly in the X.500 case, different systems and applications also accept different separator characters.

Together, the IDS designers have much experience with a number of directory services and their name syntaxes. Users and applications developers alike have been quick to point out the problems with directory names. These names are cumbersome, confusing, or just plain inconvenient to type. The separator characters within a directory name may have special meanings on some operating systems.

Because of these limitations, we decided that a name syntax specific to IDS would detract from the value of the solution. An application using IDS may choose to present its own syntax, one that is suitable to its particular environment and preferences. The API takes the object name and the context, as described in the Contexts section. The service provider module uses these to construct the name in the native name syntax.

Support of Search

Users need to locate resources in a number of ways. The most familiar method is to locate resources by knowing their name; this is often referred to as a white pages lookup, named after the printed U.S. telephone directory of alphabetically ordered names. Searching for resources based upon information about the resources is referred to as a yellow pages lookup, named after the printed U.S. telephone directory organized by business category. To support yellow pages lookup, resources must be retrievable from the directory service based on their attributes. For a printer, this might include the type of printer, the location of the printer, whether it supports color or not, who is responsible for maintaining the printer, and other information. IDS needed to support both yellow pages and white pages lookups.

Support of Application-specific Directory Entries

We saw a need to support two kinds of extensibility: the ability for an application to create new kinds of directory entries, and the ability for a customer to add attributes or other descriptive information to the directory entries created by PATHWORKS or other applications. By providing applications with the capability to create new kinds of directory entries, the IDS designers allowed IDS to be used by any application, regardless of its requirements. By allowing the addition of attributes to existing directory entries, we allowed customers to easily add information that is specific to their organization to application objects. For example, a customer might add a specific code, such as an asset identification tag, to all printer directory entries.

Design of the IDS Framework and Service Providers

IDS is an object-based system that consists of a framework and a set of service providers. For clarity, we further divided the framework into an API and a service provider interface (SPI). The API consists of a subset of the framework's objects and their public virtual methods. The SPI is a generalized, directory-service-independent interface (described in detail later in this section). The SPI objects define the abstract interface to the directory service. We use the term *service provider* to refer to any directory service that provides IDS storage. The service providers interact with the framework through the SPI.

Framework

The framework performs three major functions:

- It specifies the IDS directory-independent operations.
- It dispatches operations to directory-specific modules for execution.
- It verifies that all IDS objects and operations do not violate the IDS schema.

Figure 1 illustrates the structure of IDS. When an application makes an API call, the framework examines the name information and calls the appropriate service provider. The service provider then makes the call to the appropriate native directory service client. When the directory client returns the results, the service provider converts the results into the IDS canonical form. The design supports junctions from one directory service to another, in that the result returned to the framework by the service provider may be only a reference to an object in another directory service.

The abstract interface to the directory service ensures that IDS provides applications with a consistent level of functionality without regard to which directory service a customer has in his or her environment.

Because the words "object" and "object class" are overloaded and overused in the industry, we define the words "resource" and "resource class" to denote objects represented in IDS. A *resource* is a directory

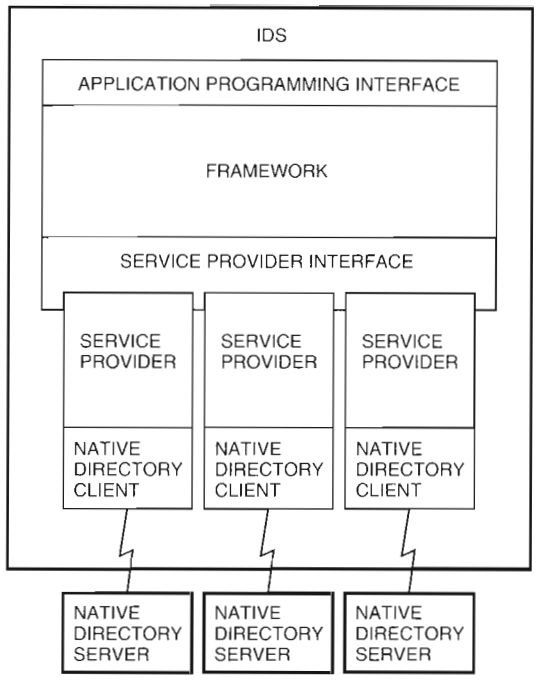


Figure 1
Structure of the Integrated Directory Services

entry; it is a directory service object that represents some network object. A *resource class* is the definition of that type of directory entry. For example, the directory entry that describes a specific printer is an IDS resource, and the IDS class that describes every printer entry is a resource class.

The framework provides extensibility by defining C++ object classes that allow for the creation and manipulation of resources, attributes, and attribute values in a type-independent manner. The type independence allows both applications and the framework itself to manipulate IDS attributes and attribute values without knowing their types. As long as the new types are built on top of existing IDS system types, application writers may define new IDS types without modifying the service providers.

The framework dispatches directory operations to the appropriate service provider and maintains overall system state and integrity. It maintains a list of the service providers that are currently available and shows the errors encountered in any failed loads. This allows the system to continue to operate, albeit in a degraded state, even though one of the service providers may be malfunctioning.

Before we discuss the design of the SPI, we describe the framework's objects.

IDS Entry The fundamental IDS object is the canonical representation of a directory entry, the IDS entry. The IDS entry is an abstract object. To create a resource class, applications define a resource type and derive it from the IDS entry. IDS entry objects are created and manipulated through the API and translated into the appropriate native directory format by the service providers. Derivatives of the IDS entry may define additional methods, but they may not override the IDS entry methods. The IDS entry methods are part of the framework.

The IDS entry methods fall into one of two categories: those which manipulate the attributes and values contained in the IDS entry in a type-independent manner, and those which perform operations on the directory. Each IDS entry, each attribute, and each attribute value contains a type. For convenience, derivatives of the IDS entry may define additional methods that manipulate certain attributes or values directly. For example, a derivation that defines a printer might define a method to set the description attribute. The implementation of this method would call the general IDS entry attribute and value manipulation method to set the value of the appropriate attribute.

As shown in Figure 2, the IDS entry contains identifying information and the attributes and attribute values that describe the resource. The context identifies the service provider that performs directory operations on this entry and the location within that directory service in which this entry is stored. The resource type defines the kind of resource that this entry represents. The resource name is the name by which applications and users refer to the entry.

The attributes of the entry are contained in a set. Each attribute in turn contains the value or list of values associated with the attribute.

Contexts The context is an object that uniquely identifies a particular location in a particular namespace. The IDS context is very similar in concept to the XFN context.[6] All contexts contain the type identifier for the directory service and an internal name. The type identifier is used by the IDS framework to dispatch operations to the appropriate service provider. The internal name is the location within the directory service described by this context. The internal name is represented in the native syntax of the underlying directory service. The service provider is responsible for setting and maintaining this internal name. (See Figure 2.)

Attributes and Attribute Values The type of an attribute defines the data type of its value or values. The attribute value object is a canonical representation of an actual attribute value. The attribute value object defines a set of methods for accessing and manipulating values. For each data type supported in IDS, there is a corresponding attribute value derivation in the

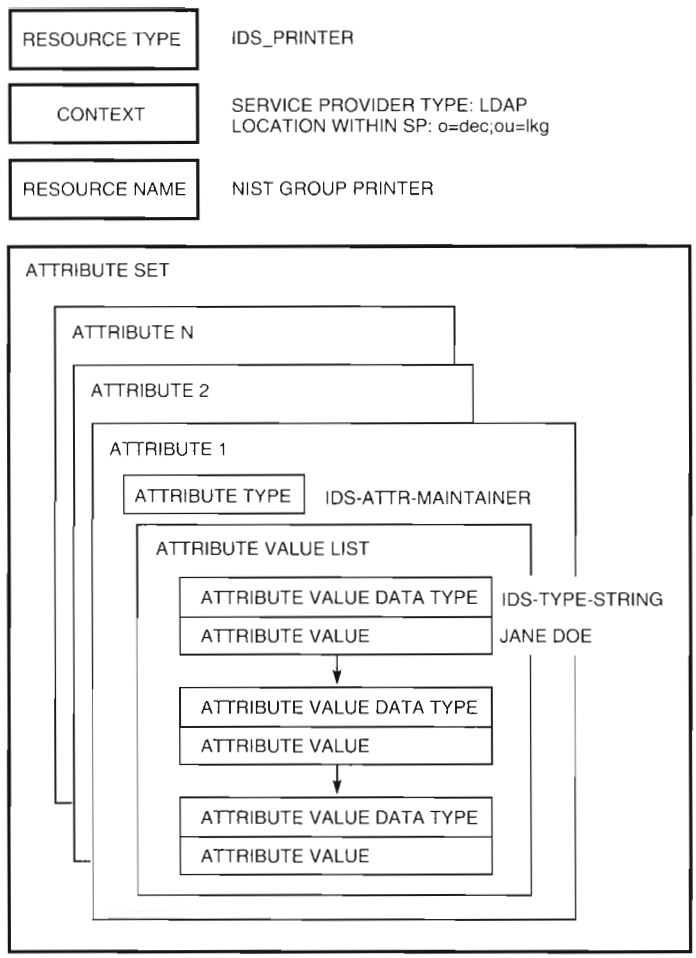


Figure 2
IDS Entry

IDS framework. This allows applications, and the IDS framework itself, to manipulate attribute values without knowing their types. The service providers, on the other hand, use the type information to translate from the IDS data formats to their native data formats.

Types To allow customers and third parties to identify their own IDS resources, the IDS type mechanism must uniquely identify objects. The two identifiers we considered using were universal unique identifiers (UUIDs) as defined by the Open Software Foundation Distributed Computing Environment (OSF DCE) and object identifiers (OIDs) as defined by the open systems interconnection (OSI) standards.[11,12] Some directory services identify attributes with OIDs, while others use UUIDs. For applications defining new resources, we wanted to avoid the necessity to obtain both an OID and a UUID. It is possible to encode a UUID in an OID, but the reverse is not true.

We could encode a UUID in an OID by registering an OID prefix. The prefix would indicate that the sequence after the prefix was a UUID. UUIDs are fixed-length structures generated from time stamps and Ethernet addresses, and therefore arbitrary information such as an OID cannot be encoded in them. UUIDs are also easier for application writers to generate because numerous systems ship with tools to generate them.

Certain directory services, for example X.500, have external type definitions for the directory entries. It is possible to define a generic entry and then map arbitrary values into that entry, but IDS entries would not be meaningful when viewed with the native directory management tools. We felt that this was unacceptable, because it would make the management of IDS entries in the namespace much more difficult. Some systems use UUIDs to represent the type information. We chose to use UUIDs since they are both easy to generate and can be used in both UUID and OID class definition systems. The use of OIDs would require UUIDs to be generated for UUID-based systems and mappings to be maintained.

Communities An IDS community is both an administrative grouping mechanism and a logical location for IDS resources. When people interact with the IDS system, they see a community as the organizing principle. The administrator controls the boundaries and membership of an IDS community. Typically, a community represents either a particular location such as a building or a functional grouping such as a work group.

Initially, we considered a supercontext to join multiple directories into a single logical directory. This supercontext would have contained multiple contexts, one for each type of resource supported by IDS. We eventually subsumed the supercontext into a community and called it a resource context list. An IDS community is stored as a special object in the directory. Each community's resource context list describes the directories that make up the community. The resource context list is the federation mechanism by which IDS determines where resources of each type are stored. Each entry in the resource context list is a pair of resource type and context. As users and applications operate on entries in a community, the IDS framework (through IDS entry and community methods) inspects the resource type and the community to determine the context. Figure 3 illustrates an IDS community.

One of the problems we anticipated was that large organizations would naturally tend to have many IDS communities: How would the user identify these? We considered an additional hierarchy in which communities would be members of other communities. Our usability consultants emphasized that users should not have to browse a hierarchy to access resources. In response, we developed the concepts of the local and the home community. The local community is associated with the machine a user is currently using—it represents a physical location. The home community is the one with which the user is associated or belongs. We envisioned that the home community would be the same as the local community at the user's normal place of work, but there is no requirement inherent in the design that things be organized this way. For example, if a user is associated with the community at her work site and the machine she uses is also located at that work site, both her local community and

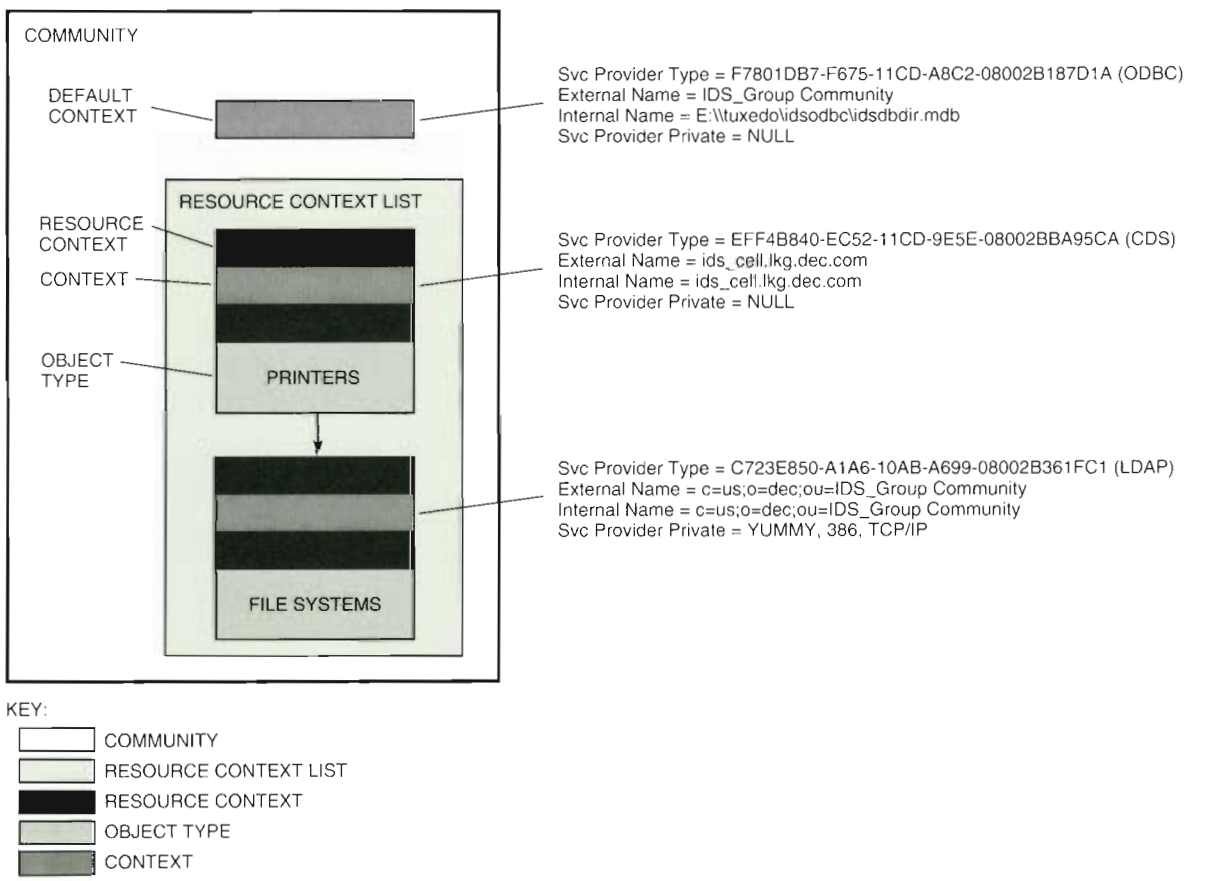


Figure 3
IDS Community

her home community represent this work site. If this user works at another work site and uses a different machine, her home community remains the same, but her local community reflects the community where the new machine resides. The concepts of local and home communities do not reduce the number of communities, but they do provide a direct method by which users can access the communities that contain the resources they most frequently use. The local and home communities are a convenience; users and applications are in no way restricted to those communities.

Search Support Searching is handled by the search object. The search object contains a community (or list of communities), a resource type, and an attribute filter. The attribute filter supports both equality and comparison matching of attribute values and allows callers to construct complex requests by concatenating comparisons together in a series of Boolean operations. For example, a caller could construct a filter that returned all printer objects that (((are located on Floor2) OR (are located on Floor3)) AND (support color printing)). Combined with the local and home community support, filters allow applications and users to express ideas such as "print this at the closest printer that supports color, two-sided printing, and then transmit it to any facsimile machine in my home community."

The search object's default filter returns all objects of the resource type in the local community. The search object resolves the community to a context and passes it to the service provider. The service provider constructs a list of matching IDS entry objects to return to the user. In IDS, the search object supports browsing.

The search object has methods that display a dialog and construct filters based on user input. When designing the system, we debated whether it was better for the search object to contain both the filter and the search dialogs or whether the filter construction belonged in the IDS entry. We chose to keep the search dialogs separate from the IDS entry. Experience with implementing resources derived from the IDS entry has shown this to be an error. Currently it is necessary to derive from two objects, IDS entry and the search object, to implement a resource that has a resource-specific search dialog. We will be modifying the search and IDS entry objects so that the construction of the filters and the dialog that constructs the filters are IDS entry methods.

Schema The service providers translate between the native directory object and the IDS entry. In general, directory service entries are not self-describing. In existing directory services, either a schema or the application is expected to know the directory-specific format of the data. The latter is more common than the former, and in any case the schema methodologies are unique to each directory service.

From the point of view of the native directory service, IDS is the application. To properly convert the data, the service providers must know what it is. The service providers use the schema to determine the correct attribute and value types to use when constructing the IDS entry of a particular type.

The schema describes resource types, attribute types, and attribute value data types. Logically, the schema is a set of tables, one for each service provider, which maps the native name or type to the IDS name or type. These tables are read by the IDS schema component when IDS is initialized. Because these tables are external to the system, they can be modified by users or applications.

There is one limitation on the extension of the schema: New attribute and resource types can be defined, but they must be composed from the predefined IDS attribute value types that the service providers can support. The service providers would have to be modified to support additional attribute value data types. This limitation is not as severe as it at first appears. A rich set of data types is defined in the existing directory services, and a relatively small set is in common usage. By defining the IDS data types to encompass the set of data types defined by existing directory services, we have reduced this limitation to a theoretical rather than a practical problem.

As a consequence of the use of schema, applications must specify the resource type for any IDS operation. This is a limitation that in principle does not exist in other directory systems. After some consideration, we concluded that few useful operations can be performed on an object whose type is unknown. To perform an operation on objects of all types, the schema can be interrogated for the list of all supported IDS object types, and the operation is then iterated over each type.

The System Object The system object loads and initializes the service providers. On initialization, the system object constructs a list of the available service providers from those defined in a local configuration file.

The system object constructs and maintains the list of known communities. The system object obtains this list using the following mechanisms:

- Inspect a well-known location (if one exists) to see if it contains a cache of known communities.
- For each service provider, call the discover method to ask the service provider for its list of known communities.
- If the system object is initializing for the first time, prompt the user to create a community.

Application Programming Interface

As mentioned previously, we divided the framework into an API and a service provider interface (SPI). The API consists of the search object methods, the IDS entry methods, the attribute object and value object methods, and the system object methods necessary to access communities.

Service Provider Interface

The SPI specifies the interface between the IDS framework and the native directory services. It defines the semantics for all operations that may be performed on IDS information regardless of which directory service stores the information. The SPI effectively insulates both the IDS framework and the IDS applications from the unique syntax and requirements of different directory services.

A directory-specific module, called a *service provider library,* provides a directory-service-specific implementation of all SPI operations and translates resource information back and forth between the IDS entry and the service-provider-specific format. A service provider library must be implemented for each directory service to be supported by IDS. Any directory service or information repository system that can provide the IDS SPI semantics may be an IDS service provider.

SPI Semantics The IDS SPI defines the following main operations: create, read, search, modify, discover, and delete. All SPI operations specify the name of the IDS community upon which to operate. Each IDS community maintains a list of contexts that specify in which service provider IDS resources of a particular type are stored and in what location within the service provider. The SPI uses this community name to retrieve the context information that directs the operation to the correct service provider library. With the exception of the delete operation, which requires an explicitly set context (to be sure that an explicitly located object is selected for deletion), if the caller does not set the community name, the local community is assumed.

The create, delete, modify, and read functions all operate on a single IDS resource at a time. Each, therefore, provides an IDS entry object to identify and/or describe the resource.

The create operation creates a new IDS resource in the directory. The create operation specifies the type of IDS resource to be created, the resource's name, and the IDS attributes and values associated with the resource. On a successful create operation, the service provider constructs a unique directory-specific name for the new IDS resource and stores this name in the object's IDS entry. The service provider subsequently may use this name to find the object more quickly rather than constructing it from the name, resource type, and context information contained in the IDS entry.

Before constructing the resource in the directory, the operation validates the IDS entry against the schema to ensure that it does not violate the schema. For example, attempting to create a resource without a required attribute value pair violates the schema and is flagged as an error. Conversely, the delete operation removes the IDS resource from the directory.

The modify operation updates the attribute and values associated with the resource in the directory. The modify operation supports the following update directives:

- Add a new attribute and value.
- Add a new value to an existing attribute.
- Replace a value of an existing attribute.
- Delete an attribute and its associated values.
- Delete a value from an existing attribute.

Each modify directive is verified against the schema before being applied to the directory.

A read operation retrieves a uniquely specified IDS resource from the directory, translates it into IDS entry format, and returns the IDS entry to the caller. The read function is typically used to compare the directory format of an IDS resource to one maintained in memory by an application, or to process IDS resources returned from a search operation one at a time.

The search function identifies and returns IDS resources that match the characteristics specified by the caller. To bound the scope of the search, the caller specifies the following search characteristics: resource type, community name or names to be searched, and a filter containing attributes and associated values or value ranges.

The discover operation is called by the IDS system object to find all communities known to a given service provider. Service providers for directory services that support a server solicitation and advertisement network protocol implement a discover function. In these directories, servers advertise their presence in response to network solicitation requests. The discover method uses the directory's native solicitation and advertisement protocol to discover local directory servers and then issues the appropriate operations to the server to determine if it has defined any IDS communities. Service providers that do not have a solicitation and advertisement protocol can implement an alternative discovery mechanism such as retrieving the community information from a file or provide no discovery mechanism.

Construction of the System: Directory, Session, and IDS Entry Objects The SPI is constructed of three framework objects: the directory object, the session object, and the directory operation methods of the IDS entry object. The directory object is responsible

for service provider initialization and termination, maintenance of session objects, and community discovery. Each service provider exports one directory object to the IDS framework. The session object implements all the directory operations on a service provider. Session objects are obtained from the service provider by means of the directory object. The IDS entry directory operation methods determine the context if it has not been set, obtain a session object from the proper directory, and dispatch the operation to the associated service provider through the session object. For efficiency, session objects may be cached by the service providers.

Implementation Considerations

Once we had established our basic approach, we turned our attention to implementation decisions.

Client versus Server

Our first consideration was whether to implement this technology as software executing on a server system or as software executing on a client system. The server solution had a number of attractive qualities: it would not be necessary to have all the native directory clients on all the desktops, and potentially complex processing would occur on an appropriate platform. However, we identified two problems with the server solution. The first concerned security. To access the directory service on behalf of a particular user, we would have to impersonate that client user on the server machine. Although this can be done without exposing security holes, doing so adds another layer of complexity to the problem. The second problem with the server solution was that it required the customer to find a machine for and deploy a server prior to getting started with the system. One of the design goals was to remove barriers to directory deployment, and we were concerned that a server solution would add a barrier. We saw a need for both client- and server-based solutions, and since the client solution was simpler to implement, we chose to start there.

Security

The IDS interfaces leave security to the underlying directory services; we did not attempt to abstract a general-purpose, access control or authentication interface. The primary reason for this was a conviction that the vast majority of current directory information is world read, and therefore a complex access control interface was not necessary. An access control and authentication layer that was directory-service-independent would have added significantly to the complexity of the project, and we chose to postpone this problem. IDS does pass requests directly to the native directory-service client; IDS does not alter or impersonate the user's identity. In that sense, it perfectly preserves the security inherent in the underlying directory services.

Filter Implementation

The implementation of the IDS attribute filter is based on the string filter as defined in RFC 1777.[13] The Lightweight Directory Access Protocol (LDAP) string filter provided a convenient internal representation, and we would be able to reuse the LDAP parsing and processing code that we had developed as part of an earlier product. We considered using SQL to construct IDS attribute filters, but chose not to do this for implementation convenience.

Service Provider Considerations

Initially, we thought that developing a directory-service-independent interface would not be difficult. Most of the required operations such as read and write are straightforward and obvious. The implementation of such an interface, however, proved to be difficult because the underlying directory services have, in some cases, very different native capabilities and semantics. We chose to implement service provider libraries for the following three types of service providers:

- Open Database Connect (ODBC)-compliant database
- X.500-based directory using the LDAP
- DCE CDS

These service providers are representative of the types of directories that exist today. Table 1 highlights some of the differences among the three directories. As this table illustrates, not all directories can natively support the semantics described by the IDS SPI. In these situations, we have followed three alternatives: (1) the service provider library implements the functionality, (2) the IDS framework implements the functionality, or (3) in a small number of cases, the service provider cannot implement the functionality and remains less functional.

Some operations cannot be supported natively by only one or a small handful of directory services. For these operations, we require the service provider developers to implement (or emulate as best they can) the functionality in the specific service provider library for that directory. For functions that a number of service providers cannot support or that are sufficiently difficult to implement, we provide a common implementation or emulation in the IDS framework that service provider libraries can call. For example, CDS does not natively support an attribute-based search mechanism. Rather than attempt to implement a CDS search capability, we chose to provide an IDS framework "prune" function that applies an IDS filter to a list of IDS entries and returns only those entries that satisfy all conditions of the filter. Service providers such as CDS can then

Table 1
Differences among the ODBC, X.500, and CDS Directories

Functionality	ODBC	X.500	CDS
Distributed directory service	No	Yes	Yes
Hierarchical organization of directory information	No	Yes	Yes
Attribute-based search	Yes	Yes	No
Attribute value-based search	Yes	Yes	No
Native schema support	Yes	Yes	No
User can extend IDS schema	No	Yes	No
Transactional semantics	Yes	No	No
Tolerant of intermittent connectivity	No	Yes	Yes
Provides security mechanism on connections	No	Yes	Yes

emulate the IDS search function by enumerating all resources of a particular type and then call the prune function to pare down the list of resources.

The IDS schema implementation is another example of a common capability we have provided for all service providers to use. Not all service providers support object, schema and, of those that do, fewer still can support user extension of the schema. We chose to allow user extensibility and implemented a service-provider-independent schema interface and mechanism.

In a few instances, we determined that it would be too expensive in terms of implementation time to provide a service-provider-specific or an IDS-framework implementation of an SPI-mandated function. In these cases, we allowed the service provider to remain noncompliant. For example, a call to initiate a session to a service provider specifies user name and password arguments. For those directories that support user name and password security mechanisms, we preserve that functionality. For directories such as the ODBC service provider that do not support these security mechanisms, however, we provide no additional security measures. The cost to implement and deploy such a security mechanism outweighs the gain of having the additional features.

In addition, we found that not all directories provide the same semantics for a particular operation. For example, when updating a resource, service providers handle existence checking of resource attributes differently. If requested to add an attribute value to an attribute that does not yet exist, one service provider returns an error, while another implicitly creates the attribute. We worked around problems of this type by carefully specifying the semantics and error conditions of all SPI operations. Service providers that do not natively support these SPI semantics must implement whatever additional functionality is required to do so. For example, the CDS service provider required additional functions that determined and flagged whether or not a particular attribute existed.

In addition to all errors that are specific to service providers, we return an error that is independent of any IDS framework service provider. This adds another level of consistency across our service-provider implementations.

Current Applications

As with any foundation technology, the proof of its viability lies with the applications that employ it. In the PATHWORKS product, we currently have three applications that use IDS:

- Network Connect
- IDS Administration
- Resource Synchronizer

The Network Connect application finds and connects users' printers and file shares. It provides a user interface that allows users to browse or search for file shares or printers. Through Network Connect, users can refer to resources by their logical name or their attributes. A single physical printer, with queues on several machines or several NOS systems, is presented to users as a single printer. Network Connect uses the IDS API to access the IDS search capabilities and to translate a printer or file share's IDS name to its network-specific name to connect to the resource. Network Connect may be accessed through the Windows version 3.1 Print Manager and File Manager utilities and through the PATHWORKS Network Connect utility.

The IDS Administration utility (IDS Admin) allows a network administrator to manage IDS resources and communities. IDS Admin is integrated into the Digital ManageWORKS Workgroup Administrator for Windows software product.[14] Admin creates, modifies, and deletes resources and communities. It also allows users to browse IDS resources and communities in the ManageWORKS hierarchy and to search for IDS resources.

An administrator can manage IDS resources manually through the ManageWORKS user interface or can rely on information provided through the semiautomatic resource collection utilities called the Resource Gatherer and Resource Synchronizer. The Resource Gatherer periodically collects information about network LAN Manager and NetWare printers and file shares. The Resource Synchronizer utility processes the gathered information, updating the directory. It also eliminates duplicate entries and discards information the administrator wishes to ignore. The gatherer and synchronizer allow the directory to be kept up-to-date, even if resources are added or removed through the native NOS interfaces.

Future Work

In the future, we plan to improve the IDS extensibility mechanisms. Currently, a local copy of the schema exists on every client. Propagating the changes to each client will become a problem as users and applications extend the schema. We are considering storing either the schema or a pointer to the schema in the directory.

The current IDS implementation runs on both the Windows version 3.1 and version 3.11 operating systems. We are currently porting it to Windows 95 and investigating ports to other operating systems, such as UNIX.

The implementation does not support the entire IDS design: Although resource context lists are implemented, there is no reasonable way for a user or administrator to create them. The user interface work for these features in the IDS Admin application has not yet been completed.

Summary

IDS provides a mechanism for integrating multiple directory services into a single system. It is predicated on the ability to define a common set of directory operations and on the type information. The implementation of three very different service providers—CDS, X.500, and ODBC—indicates that we succeeded in defining the directory operations. The use of IDS in the PATHWORKS product shows that it does address the practical aspects of the problem of integrating multiple directories into a single, logical directory service.

Acknowledgments

We would like to thank the many past and present members of the IDS team who contributed to the design and implementation of the product. Special thanks to Konstantinos Baryiames, Anthony Hinxman, David Magid, Tracy Teng, and Tamar Wexler. We would also like to thank the members of the Directory Task Force, Dah Ming Chiu, Dennis Giokas, and William Nichols.

References

1. *CCITT Recommendation X.501* (1992) and *Information Technology—Open Systems Interconnection—The Directory: Models.* ISO/IEC 9594-2: 1992 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1992).
2. "Naming Concepts" in *Using NetWare Services for NLMs* (Provo, Utah: Novell, Inc., 1993).
3. *AES/Distributed Computing—Directory Services* (Cambridge, Mass.: Open Software Foundation, 1993).
4. "StreetTalk Naming Service" in *ENS Administrator's Planning Guide* (Westborough, Mass.: Banyan Systems, Inc., 1992).
5. "Microsoft Directory Services Strategy," a white paper from the Business Systems Technology Series (Redmond, Wash.: Microsoft Corporation, 1995).
6. *X/Open CAE Specification. Federated Naming: The XFN Specification* (Reading, U.K.: X/Open Company Ltd., 1995).
7. "Bindery Services" in *NetWare System Interface: Technical Overview* (Provo, Utah: Novell, Inc., 1990).
8. S. Radicati, "Implementing the DIT" in *X.500 Directory Services: Technology and Deployment* (New York: Van Nostrand Reinhold, 1994).
9. P. Mockapetris, "Domain Names—Concepts and Facilities," Internet Engineering Task Force, RFC 1034 (November 1987).
10. P. Mockapetris, "Domain Names—Implementation and Specification," Internet Engineering Task Force, RFC 1035 (November 1987).
11. *AES/Distributed Computing—Remote Procedure Call. Appendix A* (Cambridge, Mass.: Open Software Foundation, 1993).
12. *CCITT Recommendation 208* (1992) and *Information Technology—Open Systems Interconnection—Abstract Syntax Notation One (ASN.1)* ISO/IEC 8824-2:1992 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1992).
13. W. Yeong, T. Howes, and S. Hardcastle-Kille, "X.500 Lightweight Directory Access Protocol," Internet Engineering Task Force, RFC 1777 (March 1995).
14. D. Giokas and J. Rokicki, "The Design of ManageWORKS: A User Interface Framework," *Digital Technical Journal*, vol. 6, no. 4 (Fall 1994): 63–74.

Biographies

Margaret Olson
Margaret Olson is a consulting software engineer in the Network Software Group. She was the project and technical leader for the IDS development project. For the last six years, she has had technical leadership roles in Digital's Directory Services Group. Before joining Digital in 1989, she worked in the networking and distributed computing areas at Apollo Computer. She received a B.A. (Sigma Xi) from Wellesley College in 1981. She published a paper on network licensing in 1988.

Laura E. Holly
Laura Holly is a principal engineer with the Network Software Group. She was a key technical contributor to the IDS development effort. Laura has previously contributed to the areas of DCE, distributed system, and knowledge-based system development. She joined Digital in 1985 after receiving an A.B. (high honors) from Smith College. Laura holds a patent and has published several papers in the area of knowledge-based systems.

Colin Strutt
Colin Strutt is a consulting software engineer and technical director for Teaming Software in the Network Software Group, where he is helping to define new PC-based software products. Previously, he has held technical leadership roles in directories, network management, and terminal server development, and before that led product developments in Ethernet servers and DECnet. He joined Digital in 1980 from British Airways in the U.K. He received a B.A. (honours) in 1972 and a Ph.D. in 1978, both in computer science from the University of Essex, U.K. He is a member of BCS and ACM. He has two patents issued and several patents pending and has published extensively, particularly on management technology.

Richard L. Rosenbaum
Stanley I. Goldfarb

Design of the Common Directory Interface for DECnet/OSI

Digital has developed the Common Directory Interface (CDI) as the means by which DECnet/OSI can now access and manage node name and address information in multiple directory services. CDI comprises libraries for node name-to-address translation and a tool set for managing and migrating node information among different directory services. The Common Directory Registration API is layered on top of a set of directory service wrapper routines to provide an extensible mechanism for adding new directory services. CDI gives customers greater flexibility in choosing a directory service and supports the new multiprotocol capabilities in DECnet/OSI, which support the open systems interconnection (OSI) standards.

The Common Directory Interface (CDI) provides the ability to store and retrieve DECnet node information from a variety of directory services. It consists of the CDI library, which enables multiple directory access, and the CDI registration tool set, which creates and maintains node/addressing information in multiple directory services. CDI was developed for the DECnet/OSI for OpenVMS operating system version 6.0 and for the DECnet/OSI for Digital UNIX operating system version 3.0.

This paper begins by presenting the product goals and the background of the CDI design. It then discusses the structure of the CDI components, the CDI library, and the CDI registration tool set.

Design Goals

As the interface to DECnet node information from multiple directory services, CDI was designed to meet the following goals:

- Give DECnet network administrators and users a choice of directory services.
- Provide system administrators with an easy-to-use node registration tool.
- Enable easy and flexible configuration of directory choices.
- Provide developers of the DECnet protocol software with a simple internal interface that hides the complexities and differences between the various directory services.
- Provide a common design for both DECnet/OSI platforms: the OpenVMS and the Digital UNIX operating systems.
- Interoperate with older, non-CDI systems.

Background

In 1991, Digital updated its DECnet networking products to include the use of the DECdns distributed directory service.[1] DECdns provided a highly scalable, distributed information source for translating node names to addresses and addresses to node names.

Initially, customer acceptance of this name service was low for a number of reasons:

- Adoption of this new technology required a significant learning curve.
- Significant planning was required before the DECdns service could be deployed.
- Users of small networks did not need the features of a distributed naming service—the costs outweighed the benefits. These customers requested a naming service based on local files similar to the Phase IV DECnet product.
- Customers were deploying a number of other directory services—in particular the Domain Name System—for storing host information for transmission control protocol/internet protocol (TCP/IP) networks.[2]
- A new comprehensive service, X.500, had the advantage of being an international standard.[3]

These reasons, together with the need to directly support TCP/IP host names and addresses, prompted Digital to incorporate new directory service choices in a new release of DECnet/OSI software.

CDI: Basic Design

Supporting multiple name services required decisions to be made concerning naming syntax, multiple address formats, and local file support. These decisions affected the design of both the CDI library and the CDI registration tool set.

Client-based versus Server-based Design

The earliest and most fundamental design decision was choosing between a client-based or a server-based solution. With a client-based design, support for the various directory services would be accomplished through a variety of client-based programming libraries. With a server-based design, a single client library would communicate with a new "multiheaded" server that would fan out to the directory servers.

Since clients outnumber servers, a client-based approach affects more systems during the upgrade process. In spite of this drawback, we chose a client-based solution for the following reasons:

- Implementation of the client-based design would be less complex than the server design.
- A client-based design did not have the syntax and protocol translation issues of a server-based design.
- With a server-based solution, client changes would still be required to support new native naming syntaxes.
- For small installations, no server would be needed if node information was stored in a local file: local file support was not possible with a server-only approach.

Naming Syntax

One of the most visible complications when supporting multiple naming services is the need to recognize different name syntaxes. Table 1 gives the different syntaxes for three widely used directory services.

A further complication of supporting different name syntaxes was the use of an internal DECdns name format by network management. One of the goals of the CDI design was to allow management requests to be exchanged with older, non-CDI systems.

For the initial implementation, CDI continues to support the internal DECdns format, rather than use a newer, non-DECdns specific format alongside the existing one. As a result, CDI is required to map non-DECdns names onto the DECdns format. For example, the name *hq.xyz.com* from the Domain Name System maps onto the DECdns name *DOMAIN:.hq.xyz.com* (actually onto the internal DECdns form of this name).

Multiple Address Support

Along with the introduction of CDI, a major innovation in this release of DECnet/OSI was direct support for TCP/IP transports in addition to the existing

Table 1
Naming Syntax

Directory Service	Example Name
DECdns	XYZ:.hq.sales.system1
Domain Name System	system1.sales.hq.xyz.com
X.500	/c=US/O=XYZ/ou=hq/ou=sales/ap=system1/ae=DECnet

Notes:
The X.500 service is not supported by the first release of CDI.
The syntax shown for X.500 is commonly used but is not part of a standard.

support for DECnet Phase IV and OSI. To simplify the initial implementation, IP addresses are retrieved only from the Domain Name System (not from DECdns). However, the design of CDI allows the retrieval of both kinds of addresses from any supported directory; for example, OSI addresses can be obtained from the Domain Name System.[4,5]

Support of multiple protocols created another naming issue. Many customers already have a Domain Name System in place in their networks. Often DECnet systems are also running TCP/IP protocols and are registered in the Domain Name System, yet these systems are not running DECnet software over TCP/IP. For example, a system registered as hq.xyz.com may be directly reachable with TCP/IP but not with DECnet over TCP/IP. In this case, it is possible that CDI may retrieve a valid IP address for a remote system that is unreachable by the DECnet protocol.

For these reasons, when CDI determines that both the Domain Name System and the DECdns naming service (or a local file) are specified in the search path, it does not stop processing the search path until both the IP address and the OSI address have been obtained, or until the end of the list has been reached. In this way, if the desired remote system is not running DECnet over TCP/IP, an attempt to connect will be made through the DECnet protocol, using a connectionless network service (CLNS) OSI address.

Local File Support

Early versions of the DECnet networking product offered only a local file for node-to-address information. The first release of DECnet/OSI replaced the local file with the DECdns naming service. Unfortunately, administrators of small- and medium-sized networks found that the benefits of DECdns (scaling and centralized management) were outweighed by its additional complexity.

A subsequent version of DECnet/OSI introduced the Local Naming Option. This allowed approximately 150 nodes to be stored in a local file, but many customers found this number to be too small.

CDI supports a very large local file: the supported limit is 100,000 nodes, but there is no fixed internal limit. In addition, through the use of the search path, customers can configure the local file either as a backup to a distributed service, or as a way to provide greater performance. Note that both of these qualities are also provided in a more automated way by the CDI cache (see the CDI Library Cache section for more information).

Security Considerations

CDI relies upon the security provided by the underlying directory services (or in the case of the local file, the file system). Security of its remote management features depends on the network management security system.

CDI Libraries: Basic Design

CDI is implemented as shared libraries on both the Digital UNIX and the OpenVMS operating systems. At the highest level, the design is identical on both systems, as shown in Figure 1. Name-to-address translation requests from the session control layer are passed through a single entry point in each CDI library.

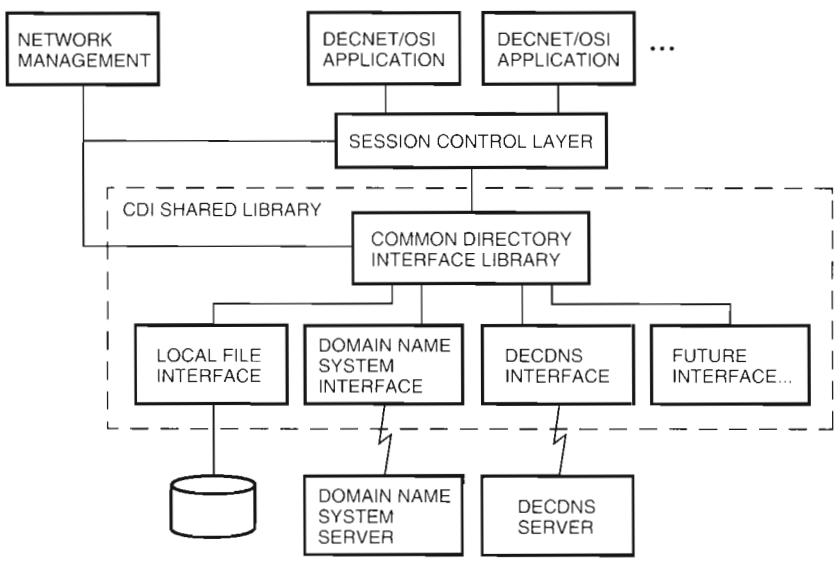


Figure 1
Block Diagram of the CDI Library

Depending upon the search path (described below), the CDI libraries translate and forward the request to one or more directory services (or they look up the information in a local file).

The CDI implementation was considerably more complex on the OpenVMS operating system than on the Digital UNIX operating system due to the differing design of DECnet/OSI on each system. On the Digital UNIX operating system, the DECnet/OSI session control layer consists of a shared library that is linked with each network application. Name resolution requests are processed synchronously. On the OpenVMS operating system, session control is a component of the NET$ACP process. Since all name resolution requests are channeled through this single process, operations must be asynchronous (requests must block concurrent operations). In addition, since multiple requests may be simultaneously outstanding, the library is multithreaded. Asynchronous, multithreaded operations on the OpenVMS operating system are implemented using the asynchronous system trap (AST) mechanism. For these reasons, the CDI implementation on OpenVMS was much larger and more complex.

CDI Search Path

Another goal was to permit flexibility in determining a configuration of directory services. The CDI design achieves this goal in two ways. First, it allows administrators to select their service(s) of choice and to use them in any order. The search path is normally created during network configuration and can be subsequently managed either locally or remotely. Second, it gives network users the ability to use short, abbreviated names instead of potentially cumbersome full names. For example, they can use "system1" instead of "system1.sales.hq.xyz.com."

A single mechanism in the CDI library—the CDI search path—provides these two capabilities. The search path consists of a series of directory service/name template pairs, as shown in Figure 2a. When the CDI library is given a name to process, it scans the search path, replacing the "*" in the name template with the supplied name. For example, if the library was searching for the name *frodo*, it would use the directory services identified from the names generated shown in Figure 2b.

During network configuration, a default search path is automatically configured based upon the local node name and the administrator-specified directory services. This search path behavior is similar to a number of existing TCP/IP host name/address lookup implementations.

CDI Library Cache

Occasionally, name service lookups can take a long time to complete (for example, if requests are traversing a slow network link, a lookup could take several seconds). To improve performance, the CDI library incorporates a single cache that accumulates node information from all the directory services. Usually, the cache is consulted before sending a request to a remote service. However, if session control determines that cached information is stale—for example, if connection to a node at a cached address reaches a node with a different name—it will reissue the call, requesting that the cache be bypassed.

Each entry in the cache has a creation time stored with it. The cache itself has a "time-to-live" value that can be modified by the administrator. If a cache lookup finds an entry whose lifetime (time since it was created) is greater than the time-to-live value, the cache entry is purged.

To prevent a period of low performance immediately after system start-up, the cache is preserved across system reboots by periodically checkpointing it to a disk file. The checkpoint interval is adjustable by the administrator.

DECdns	*
DECdns	XYZ:.hq.sales.*
DECdns	XYZ:.DNA_Node_synonym.*
Domain	*
Domain	*.sales.hq.xyz.com

(a) Directory Service/Name Template Pairs

frodo	(DECdns)
XYZ:.hq.sales.frodo	(DECdns)
XYZ:.DNA_Node_synonym.frodo	(DECdns)
frodo	(Domain)
frodo.sales.hq.xyz.com	(Domain)

(b) Address Lookup for Name *frodo*

Figure 2
Using the CDI Search Path

CDI Registration Tool: Basic Design

The CDI registration tool provides functions to create, modify, rename, display, and delete node name and address information in any of the supported directory services. It runs on the major DECnet/OSI platforms, the OpenVMS and the Digital UNIX operating systems.

The basic requirements for the CDI registration tool were the same as those for the CDI library. These three requirements were the need to:

- Support different directory services for storing node information
- Access each directory service using the appropriate application programming interfaces (APIs)

- Store data in each directory service using the appropriate data types

In addition, the following requirements were specific to the CDI registration tool:

- Both a forms and console user interface had to be provided. These had to work identically on all DECnet/OSI operating system platforms.
- Functions to transfer node information between the various directory services had to be provided.
- Other applications such as the DECnet/OSI network control language (NCL) utility and other namespace management tools had to be able to access node name management functions.

The directory services supported by the CDI registration tool are slightly different from those supported by the CDI library. The CDI registration tool supports the DECdns, the local file, and the DECnet Phase IV database services.

The DECnet Phase IV database is supported by the CDI registration tool to allow administrators to use old Phase IV node information when populating the node names and addresses for DECnet/OSI. The Phase IV database is not supported for node name-to-address lookup by the CDI library.

Due to its lack of a remote update capability, the Domain Name System is not supported by the CDI registration tool. Node name-to-address information in the Domain Name System is managed using its native tools. Dynamic updating of the Domain Name System servers is currently under study by the Internet Engineering Task Force (IETF) Domain Name System Working Group.

Application Design

The design of the CDI registration tool uses a client-based, multilayer approach. It is layered on top of a specialized API, called the Common Directory Registration (CDR) API. The CDR API differs from the API provided by the CDI library in that it presents a full set of management operations, rather than just the lookup operations required by DECnet/OSI.

In this design, the CDI registration tool provides forms and console user interfaces for node information management. It also provides functions beyond the basic ones provided by the CDR API, such as exporting from and importing to a directory service. The function of the CDR API is to perform all underlying node name management operations in a standardized manner. This layered approach was adopted to make node name management functions available to applications other than the CDI registration tool.

The CDR API defines a node definition object. This contains all the information that is exchanged between the CDR API and the application and is a canonical, directory-service-independent data representation of all information needed by the CDR API to manage node names and addresses.

To provide an extensible mechanism for adding new directory services, the CDR API is layered on top of a set of directory service wrapper routines, one per supported directory service. Access to these wrapper routines is provided by a set of entry point tables that can be extended to support new directory services. The CDR API is responsible for accepting application requests and dispatching them to the correct directory service by means of the appropriate wrapper routine. The CDR API wrapper routines are described later in this section.

Figure 3 shows the design of the CDI registration tool and the CDR API.

CDI Registration Tool User Interface

The forms and the console user interfaces had to present exactly the same characteristics on both the OpenVMS and the Digital UNIX operating systems. Because no high-level software packages at the time could provide this level of user interface portability, we developed them for this application.

The console user interface parses commands and dispatches them to the appropriate user request processing routine, using a portable command parser.

The forms user interface obtains input from task-specific forms and dispatches the function or functions associated with the form to the appropriate user request processing routine. The forms processor was written specifically for this application because no existing libraries could provide the required level of portability.

CDI Registration Tool User Request Processing

Each user request maps into a specific request processing function as follows:

- Register. Create a new node name entry in the directory service.
- Add address. Add address information to a node name entry.
- Remove address. Remove address information from a node name entry.
- Modify address. Replace the address information in a node name entry.
- Update address. Replace the address information in one or more node name entries, using information obtained from the nodes themselves (if possible).
- Modify synonym. Replace the node synonym in a node name entry.
- Rename. Change the name of a node name entry.
- Show. Display the information contained in one or more node name entries.

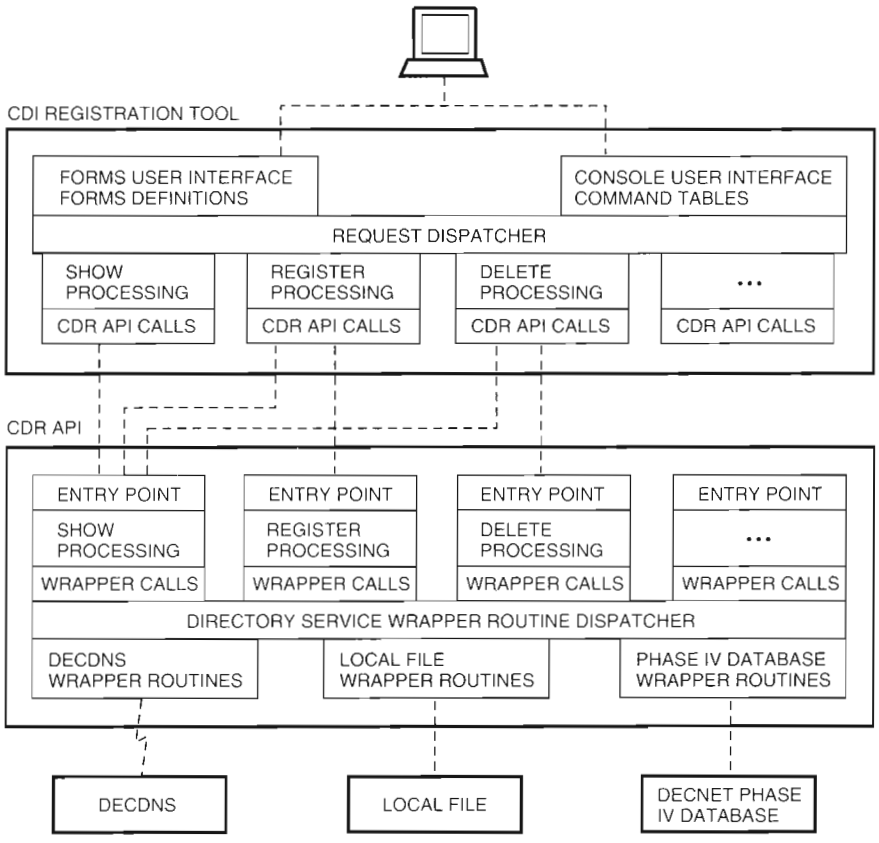


Figure 3
Block Diagram of the CDI Registration Tool and the CDR API

- Deregister. Delete one or more node name entries by name, synonym, or address.
- Repair. Fix any detected problems or inconsistencies in the directory service for one or more node name entries.
- Export. Copy the information for one or more node name entries from the directory service into a text file that can be copied between systems, edited if necessary, and imported into any other directory service.
- Import. Use an export text file to register, modify, or deregister node name entries in a directory service.

The request processing routines perform any required validation of the user request and translate those requests to calls into the CDR API. Each request may map into one or more CDR API calls, depending on the complexity of the request. For example, register and deregister requests both map into single CDR API calls, and export and import requests map into several calls.

Most requests are straightforward in their processing requirements. For example, a register request simply calls the CDR API register entry point. The CDR API takes care of any complications in processing the request.

Some requests can operate over multiple node name entries. For example, the show request enumerates the node name entries, retrieves the information contained in each node name entry, and displays the information to the user.

An export request is similar to a show request, except that the resulting information is written to a text file in a standard format instead of being displayed to the user. The import request, however, is more complicated. This request must enumerate and show the contents of the directory service, and then compare the results with the contents of the text file. Based on the specific form of the import request, it may then register new node name entries, update the information in existing node name entries, or deregister listed node name entries.

The export and import requests make use of a text file to provide maximum flexibility. The use of a text file allows the information to be copied between dissimilar platforms such as the OpenVMS and the Digital UNIX operating systems, and allows the information to be manipulated using standard tools such as batch files, grep, awk, and text editors. This is particularly useful when applying a change to all node entries.

For example, the contents of a directory service could be exported to a text file, the addresses in the text file changed to reflect a new routing area, and the results imported back into the directory to update the existing information.

The repair function performs a show operation on all specified node names to determine if any consistency errors are found. This type of error can occur in directory services that keep multiple physical records for each logical node name entry. DECdns is one example of this kind of directory service, because it uses soft links to map node synonyms and addresses back to their respective node name entries. If this type of error is found, the repair function re-registers the node synonym and address information to correct these inconsistencies.

The most complicated request is the update request. This performs a show request for the specified node names and attempts to use the current addressing information contained in the node name entry to make a network management connection to the node itself. For each node name entry, it steps through the complete set of registered addresses and tries each address in turn, using both a DECnet Phase IV connect and a DECnet/OSI connect. If a connect attempt is successful, it uses the appropriate network management requests to read the true addressing data. It then compares this addressing data to what it found in the directory service and makes any necessary corrections to the node name entry. The update operation does not operate on IP addresses due to the lack of dynamic update capabilities in the Domain Name System servers.

Before making the CDR API calls, all request processing routines convert the user request data into a node definition object, which is discussed in the next section.

CDR API Node Definition Object

The node definition object is the only input data provided to any of the CDR API entry points. It stores the necessary data for any directory service operation, using a canonical representation. The node definition object contains the following:

1. Type of directory service to access
2. Name of the node entry to access (depending on the operation being performed, it may allow a fully qualified name, a synonym, an address, or wildcards)
3. Synonym name (for DECnet Phase IV access)
4. DECnet Phase IV network service access point (NSAP) prefix (for use when a Phase IV address is specified)
5. Address information
6. Directory names used for reverse mapping of synonym names and addresses back to the fully qualified node name

The CDR API controls all access to elements within the node definition object, which further isolates the calling application from the lower-level data structures.

CDR API Entry Points

Each CDR API entry point provides one logical function to the calling application. Each user request can translate into one or more CDR API functions. The functions are

- Register. Create a new node name entry in the directory service.
- Add address. Add address information to a node name entry.
- Remove address. Remove address information from a node name entry.
- Modify address. Replace the address information in a node name entry.
- Modify synonym. Replace the node synonym in a node name entry.
- Rename. Change the name of a node name entry.
- Show. Return the information contained in one or more node name entries.
- Deregister. Delete one or more node name entries by name, synonym, or address.
- Enumerate. Return a series of node name entries, one at a time, based on a wildcard specification.

All node information passed to and from the CDR API is in the form of a node definition object, as described previously. The CDR API functions validate the canonical information contained in the node definition object and dispatch a directory-service-specific function to handle the request.

CDR API Directory Service Wrapper Routines

Each directory service supported by the CDR API has an associated set of directory service management wrapper routines. These routines provide entry points that are functionally identical to those provided by the CDR API. The CDR API does the initial input argument validation, and the directory service wrapper routines perform the data manipulation in the underlying directory service.

The CDR API dispatches the appropriate directory service wrapper routine using a set of entry point tables. This provides a means to easily extend the CDR API to include additional directory services in future versions.

CDR API Wrapper Routines for DECdns

In the DECdns name service, each node name entry contains all the information required to translate a node name to a synonym or a set of node addresses. However, no search mechanism exists to allow a

lookup of the node name entry based on the synonym or on an address. For this reason, all functions that create, modify, and delete node name entries (register, modify addresses, modify synonym, rename, and deregister) must also create, modify, and delete reverse mapping entries.

Reverse mapping entries are based on a node's synonym and addresses; they contain pointers to the true node name entry. These entries are used by the CDI library lookup functions and by the CDR API display functions (show and enumerate) to access the node name entry when given a synonym or address.

The use of reverse mapping entries requires that multiple directory service entries be created for each registered node. These must be synchronized by properly ordering the creation and deletion of the various entries when registering, modifying, or deregistering a node name. For example, when registering, the node name entry is created and its synonym and address values are set before the reverse mapping entries are created and set. Similarly, when deregistering, the reverse mapping entries are deleted before the node name entry is deleted. This prevents orphaned reverse mapping entries from being created, because they can always be found by starting from the information contained in the node name entry.

The repair function is provided in case a register or deregister operation fails before completion. The repair function corrects the reverse mapping entries by re-registering all node name entries that show errors. The CDI registration tool (not the CDR API) provides this higher-level function.

CDR API Wrapper Routines for the Local Node File

Under the OpenVMS operating system, the local node name file is implemented using a record management system (RMS)-indexed file. Under the Digital UNIX operating system, a DBM-indexed file is used. On both systems, the file content is essentially the same.

The local node name file contains a series of logical records, one for each node name entry in the directory service. Together, these records define each node's fully qualified name, its synonym, and its addresses. This logical record may be looked up using the full name, the synonym, or any of the node's addresses.

Each logical record consists of (1) a node definition physical record, which contains all information related to the node, and (2) zero or more reverse mapping physical records, which contain alternate keys for looking up the node definition. Each reverse mapping record contains only the node name key in its record data. All the data used to describe the node is contained in the node definition record.

Because multiple records compose a node name entry, operations that fail to complete can result in inconsistencies in the local node file. Fortunately, these inconsistencies can be resolved using the same synchronization algorithms as used for DECdns.

CDR API Wrapper Routines for the DECnet Phase IV Node Database

Access to the DECnet Phase IV node database is provided primarily to help users migrate their Phase IV node name data to DECnet/OSI. No access is provided to this database by the CDI library for DECnet/OSI applications. Because this database consists of a simple file, with one record per node name entry, none of the multiple record synchronization problems exist.

Conclusion

The Common Directory Interface, consisting of the CDI registration tool set and the CDI library, provides flexible and extensible directory service access for DECnet/OSI. Initial customer acceptance of these new capabilities has been high and future enhancements are being studied.

Acknowledgments

The design and development of the Common Directory Interface involved the contributions of the entire directory services and DECnet engineering teams. We extend our thanks to all the team members, as well as to product and engineering management for supporting this project.

References

1. S. Martin, J. McCann, and D. Oran, "Development of the VAX Distributed Name Service," *Digital Technical Journal*, vol. 1, no. 9 (June 1989): 9–15.

2. P. Mockapetris, "Domain Names—Implementation and Specification," RFC 1035, Internet Document (November 1987).

3. CCITT Sixth Plenary Assembly, "The Directory—Overview of Concepts, Models and Services," *Recommendation X.500 and ISO 9594-1. Data Communications Networks Directory: Recommendations X.500 to X.521, CCITT Blue Book,* vol. xiii.8 (Geneva: International Telecommunications Union, 1989).

4. R. Rosenbaum, "Using the Domain Name System to Store Arbitrary String Attributes," RFC 1464, Internet Document (May 1993).

5. B. Manning and R. Colella, "The Domain Name System NSAP Resource Records," RFC 1706, Internet Document (October 1994).

Biographies

Richard L. Rosenbaum
Rich Rosenbaum is a software engineering consultant in the Internet Software Business Unit, where he is focusing on the application of indexing and collaboration technologies to the World Wide Web. In his 17 years with Digital, he has worked on networking products operating on Digital's 16-, 32-, 36-, and 64-bit platforms. He is the co-author of several patents on network software. Rich obtained a B.S. from the State University of New York at Stony Brook.

Stanley I. Goldfarb
Stan Goldfarb is a principal software engineer with the Internet Software Business Unit. Since joining Digital in 1976, he has contributed to several network and network management projects, including DECnet/RSX, DECnet-PRO, DECnet-DOS, DECmcc, DECnet/OSI, and PATHWORKS, and he has co-authored several patents on network management software. He is currently working on a Workgroup Web Forum application to provide electronic mail subscription and distribution services. Stan holds B.S. and M.S. degrees in computer science from Worcester Polytechnic Institute and an M.S. in management from Lesley College.

Recent Digital U.S. Patents

The following patents were recently issued to Digital Equipment Corporation. Titles and names supplied to us by the U.S. Patent and Trademark Office are reproduced exactly as they appear on the original published patent.

D336,081	S. K. Morgan and M. L. Hetfield	Electronic Device Module
D336,082	S. K. Morgan and M. L. Hetfield	Electronic Device Module
D336,290	S. K. Morgan and M. L. Hetfield	Enclosure for Electronic Module
D336,636	R. Veno, K. Palumbo, P. Roach, P. Barron, and M. Freeman	Power Supply Door
D337,761	S. K. Morgan and M. L. Hetfield	Electronic Device Module
D338,001	M. Falkner, M. Good, and M. Wiesenhahn	Positioning Device
D338,653	S. K. Morgan and M. L. Hetfield	Power Supply Module
D339,325	L. Spencer and C. Detsikas	Face Plate
D340,035	R. Faranda	Central Processing Unit Enclosure
D342,523	S. K. Morgan and M. L. Hetfield	Cover for Wall-mounted Electronic Equipment
D344,710	S. K. Morgan and M. L. Hetfield	Electronic Device Module
D346,370	M. L. Hetfield and S. K. Morgan	Network Multiplexor for an Office Environment
D347,624	W. McCarthy, R. Hellweg, R. Masters, M. Freeman, C. Williams, C. Brench, K. Palumbo, D. Snow, and P. Barron	Card Cage Enclosure
D348,448	C. E. Vaillant, J. D. Read, and G. J. Norquay	Removable Rigid Disk Drive
D348,672	M. J. Falkner and M. W. Kleeman	Desktop Audio Enclosure
D350,341	C. Landry	Display Monitor
5,209,389	K. Sullivan and P. Caine	Solder Pump Bushing Seal
5,210,795	S. Lipner, M. Gasser, and B. W. Lampson	Secure User Authentication from Personal Computer
5,212,776	M. Kindervater and F. Zandveld	Computer System Comprising a Main Bus and an Additional Communication Means Directly Connected between Processor and Main Memory
5,214,963	D. Widder	Method and Apparatus for Testing Inner Lead Bonds
5,215,608	R. Stroud and K. Vonbrandt	Composition and Method for Bonding Electrical Components
5,216,655	P. Hearn, A. Prentakis, W. Lewis, and F. Zayas	Method and Apparatus for Surface Reallocation for Improved Manufacturing Process Margin
5,216,672	P. M. Goodwin, D. W. Smelser, and D. A. Tatosian	Parallel Diagnostic Mode for Testing Computer Memory
5,218,513	D. Brown	Plenum for Air-impingement Cooling of Electronic Components
5,220,271	J. Palczynski	Cross Regulator for a Multiple Output Power Supply
5,223,710	R. Pavlak	Optical Angular Position Sensing System for Use with a Galvanometer
5,223,806	R. Curtis and D. Skendzic	Method and Apparatus for Reducing Electromagnetic Interference and Emission Associated with Computer Network Interfaces
5,223,996	C. E. Vaillant and J. D. Read	Combined Shock Mount Frame and Seal for a Rigid Disk Drive
5,224,235	P. Lison and W. Baines	Electronic Component Cleaning Apparatus

5,224,263	W. Hamburgen	Gentle Package Extraction Tool and Method
5,225,790	R. Noguchi, J. Rinaldis, and P. Esling	Tunable Wideband Active Filter
5,226,092	K. Chen	Method and Apparatus for Learning in a Neural Network
5,227,041	B. Brogden, L. Brown, and S. Husain	Dry Contact Electroplating Apparatus
5,227,582	D. J. Velasco, J. P. Copeland, D. C. Robinson, and R. L. Fernandez	Video Amplifier Assembly Mount
5,227,604	G. Freedman	Atmospheric Pressure Gaseous-flux-assisted Laser Reflow Soldering
5,228,066	C. Devane	System and Method for Measuring Computer System Time Intervals
5,229,901	M. Mallary	Side-by-Side Read/Write Heads with Rotary Positioner
5,229,914	D. Bailey	Cooling Device that Creates Longitudinal Vortices
5,229,926	D. D. Donaldson and D. Wissell	Power Supply Interlock for Distributed Power Systems
5,231,246	J. Benson, D. Alessandrini, and W. Rett	Apparatus for Securing Shielding or the Like
5,232,570	W. Haines, R. Raymond, C. Byun, E. Johns, D. Ravipati, Q. Ng, and G. Rauch	Nitrogen-containing Materials for Wear Protection and Friction Reduction
5,235,617	W. Mallard	Transmission Media Driving System
5,235,642	E. Wobber, M. Abadi, A. Birrell, and B. W. Lampson	Access Control Subsystem and Method for Distributed Computer System Using Locally Cached Authentication Credentials
5,239,260	D. Widder and D. Ringleb	Semiconductor Probe and Alignment System
5,239,274	K. Chi	Voltage-controlled Ring Oscillator Using Complementary Differential Buffers for Generating Multiple Phase Signals
5,240,549	W. Hamburgen and J. Fitch	Fixture and Method for Attaching Components
5,241,632	T. Creedon, D. Smith, and A. O'Connell	Programmable Priority Arbiter
5,241,639	F. Feldbrugge	Method for Updating Modified Data from a Cache Address Location to Main Memory and Maintaining the Cache Address in Registration Memory
5,243,308	R. Curtis and B. Shusterman	Combined Differential-mode and Common-mode Noise Filter
5,243,495	C. E. Vaillant, J. D. Read, and G. Norquay	Removable Enclosure Housing a Rigid Disk Drive
5,243,756	W. Hamburgen and J. Fitch	Integrated Circuit Protection by Liquid Encapsulation
5,247,426	W. Hamburgen and J. Fitch	Semiconductor Heat Removal Apparatus with Non-uniform Conductance
5,248,253	A. Philipossian and E. Culley	Thermal Processing Furnace with Improved Plug Flow
5,251,316	P. Anick and R. Flynn	Method and Apparatus for Integrating a Dynamic Lexicon into a Full-text Information Retrieval System
5,254,930	J. A. Daly	Fault Detector for a Plurality of Batteries in Battery Backup Systems
5,255,287	D. C. Davies, D. G. Vonada, and R. A. Curtis	Transceiver Apparatus and Methods
5,255,375	N. Crook, P. Bruce, and R. Galuszka	High Performance Interface between an Asynchronous Bus and One or More Processors or the Like
5,256,060	A. Philipossian and E. Culley	Reducing Gas Recirculation in Thermal Processing Furnace
5,256,975	R. Mellitz and E. Stearns	Manually Operated Continuity/Shorts Test Probe for Bare Interconnection Packages
5,260,864	J. Simonelli and Z. Arbanas	Configurable Inverter for 120 VAC or 240 VAC Output
5,260,928	A. Jain, N. Lee, and E. Keppeler	Apparatus and Method for Fabricating a Lens/Mirror Tower
5,260,945	T. L. Rodeheffer	Intermittent Component Failure Manager and Method for Minimizing Disruption of Distributed Computer System
5,260,999	R. Wyman	Filters in License Management System
5,261,002	R.J. Perlman and C. W. Kaufman	Method of Issuance and Revocation of Certificates of Authenticity Used in Public Key Networks and Other Systems
5,263,030	P. S. Rotker, and E. W. Ertel	Method and Apparatus for Encoding Data for Storage on Magnetic Tape
5,263,032	B. Porter, C. A. Mega, and R. L. Myers	Computer System Operation with Corrected Read Data Function
5,265,212	B. E. William	Sharing of Bus Access among Multiple State Machines with Minimal Wait Time and Prioritization of Like Cycle Types
5,265,216	C. P. Murphy, T. Creedon, and C. D. Cremin	High-performance Asynchronous Bus Interface
5,266,156	A. Nasr	Methods of Forming a Local Interconnect and a High Resistor Polysilicon Load by Reacting Cobalt with Polysilicon

5,267,112	S. Batra, S. Ramaswamy, and M. Mallary	Thin Film Read/Write Head for Minimizing Erase Fringing and Method of Making the Same
5,267,199	R. J. Galuszka, A. J. Walton, and C. Choi	Apparatus for Simultaneous Write Access to a Single Bit Memory
5,267,235	C. P. Thacker	Method and Apparatus for Resource Arbitration
5,267,237	A. T. Townley	Collision Detection and Signaling Circuit
5,267,867	F. Aghadel and C. W. Ho	Package for Multiple Removable Integrated Circuits
5,268,837	M. Saylor	Robotics Workstation
5,268,962	M. Abadi, M. Burrows, and B. W. Lampson	Computer Network with Modified Host-to-Host Encryption Keys
5,269,013	K. D. Abramson, H. B. Butts, and D. A. Orbits	Adaptive Memory Management Method for Coupled Memory Multiprocessor Systems
5,272,390	H. A. Collins, R. B. Watson, and R. Iknaian	Method and Apparatus for Clock Skew Reduction through Absolute Delay Regulation
5,272,445	S. G. Lloyd and H. Partovi	Resistance Tester Utilizing Regulator Circuits
5,273,455	L. MacLellan	Torsion Bar Connector
5,274,210	G. Freedman, P. Elmgren, and M. Brodeur	Laser Bonding Highly Reflective Surfaces
5,274,509	B. D. Buch	On-the-fly Splitting of Disk Data Blocks Using Timed Sampling of a Data Position Indicator
5,274,628	N. D. Godiwala and K. M. Thaller	Multisignal Synchronizer with Shared Last Stage
5,276,569	W. F. Even	Spindle Controller with Startup Correction of Disk Position
5,276,872	D. B. Lomet and B. J. Salzberg	Concurrency and Recovery for Index Trees with Nodal Updates Using Multiple Atomic Actions by Which the Trees Integrity is Preserved during Undesired System Interruptions
5,278,703	B. Rub, J. E. Deroo, S. B. Skraly, A. Solli, and R. Frame	Embedded Servo Banded Format for Magnetic Disks for Use with a Data Processing System
5,278,783	J. Edmondson	Fast Area-Efficient Multi-bit Binary Adder with Low Fan-out Signals
5,279, 865	R. P. Chebi and S. Mittal	High Throughput Interlevel Dielectric Gap Filling Process
5,280,437	D. A. Corliss	Structure and Method for Direct Calibration of Registration Measurement Systems to Actual Semiconductor Wafer Process Topography
5,280,608	A. J. Beverson, T. E. Hunt, and G. P. Lidington	Programmable Stall Cycles
5,281,869	J. R. Lundberg	Reduced Voltage NMOS Output Driver
5,283,560	J. F. Bartlett	Computer System and Method for Displaying Images with Superimposed Partially Transparent Menus
5,285,007	A. E. Deluca, J. M. Lewis, C. L. Leo, T. J. Orr, D. T. Symmes, and R. A. Barker	System for Reducing the Emission of High Frequency Electromagnetic Waves from Computer Systems
5,286,919	J. W. Benson and D. T. Staffiere	Computer Cable Management System
5,287,263	M. Shilo	Inrush Current Control Circuit
5,287,359	W. Engelse	Synchronous Decoder for Self-clocking Signals
5,287,500	P. Stoppani	System for Allocating Storage Spaces Based upon Required and Optional Service Attributes Having Assigned Priorities
5,287,501	D. B. Lomet	Multilevel Transaction Recovery in a Database System Which Loss Parent Transaction Undo Operation upon Commit of Child Transaction
5,287,517	B. A. Maskas, J. A. Metzger, and G. J. Harris	Self-compensating Voltage Level Shifting Circuit
5,289,046	J. A. Daly, J. M. Gregorich, and G. J. Brand	Power Converter with Controller for Switching between Primary and Battery Power Sources
5,289,328	G. Saliba	Method and Apparatus for Variable Density Read-after-writing on Magnetic Tape
5,289,347	W. F. McCarthy, D. M. Snow, and C. E. Brench	Enclosure for Electronic Modules
5,291,529	N. A. Crook, P. L. Bruce, and R. J. Galuszka	Synchronization Scheme
5,293,486	M. A. Jordan and D. J. Donnelly	Deterministic Method for Allocation of a Shared Resource
5,293,487	A. P. Russo, S. L. Rege, M. F. Kempf, and E. T. Sullivan	Network Adapter with High Throughput Data Transfer Circuit to Optimize Network Data Transfers, with Host Receive Ring Resource Monitoring and Reporting
5,294,842	R. Iknaian and R. B. Watson	PVT Update Synchronizer

5,294,994	D. C. Robinson, J. P. Copeland, D. J. Velasco, R. L. Fernandez, and S. D. Venditti	Integrated Computer Assembly
5,297,107	J. A. Metzger and P. J. Graffam	Interconnect Arrangement for Electronic Components Disposed on a Circuit Board
5,297,291	D. L. Murphy	System for Linking Program Units by Binding Symbol Vector Index in the Symbol Table into Calling Image to Obtain Current Value of the Target Image
5,297,992	D. Bailey, P. Martino, and B. Arsenault	Method and Apparatus for Liquid Spill Containment
5,299,206	A. Beaverson and C. J. Devane	A General Process for Finding Patterns in Large Logic Traces (or Other Large Binary Arrays) Using Multiple Concurrent Finite Automata with Cross-communication
5,301,186	R. Galuszka, A. Walton, and S. Bryant	High Speed Transmission Line Interface
5,301,283	C. P. Thacker and D. Hartwell	Dynamic Arbitration for System Bus Control in Multiprocessor Data Processing System
5,301,320	P. J. Cerqua, S. M. Kennedy, J. D. McAtee, and P.J. Piccolomini	Workflow Management and Control System
5,301,325	T. R. Benson	Use of Stack Depth to Identify Architecture and Calling Standard Dependencies in Machine Code
5,302,960	P. Boers	Multi-element Susceptibility Room
5,303,302	M. Burrows	Network Packet Receiver with Buffer Logic for Reassembling Interleaved Data Packets
5,303,347	S. L. Rege and D. A. Gagne	Attribute Based Multiple Data Structures in Host for Network Received Traffic
5,304,939	D. C. Davies	Tracking Peak Detector
5,305,185	V. Samarov, J. DeCarolis, R. Patel, G. Piche, G. Skutt, and S. Norris	Coplanar Heatsink and Electronics Assembly
5,305,354	N. D. Godiwala and K. M. Thaller	Aborting Synchronizer
5,305,389	M. L. Palmer	Predictive Cache for Improved Performance in Retrieving Cached Data
5,306,994	L. Supino	Automatic Phase Margin Compensation Control Circuit and Method for Disk Drives
5,307,217	G. A. Saliba	Magnetic Head for Very High Track Density Magnetic Recording
5,307,256	R. Silverstein	Trickle Charge Circuit for an Off-line Switching Power Supply
5,307,336	N. K. Lee, A. Jain, E. Keppeler, and M. Bouchard	Multi-disk Optical Storage System
5,307,345	S.-T. Ben-Michael and P. Lozowick	Method and Apparatus for Cut-through Data Packet Transfer in a Bridge Device
5,307,492	T. R. Benson	Mapping Assembly Language Argument List References in Translating Code for Different Machine Architectures
5,308,429	S. J. Bradley	System for Bonding a Heatsink to a Semiconductor Chip Package
5,309,035	H. Collins, R. Watson, and R. Iknaian	Method and Apparatus for Clock Skew Reduction through Absolute Delay Regulation
5,309,294	D. Cahalan	Method and Circuitry to Provide True Voltage Bias to a Magnetoresistive Head
5,309,451	E. S. Noya, M. N. Rosich, and R. M. Arnott	Data and Parity Prefetching for Redundant Arrays of Disk Drives
5,309,569	N. A. Warchol	Self-configuring Bus Termination Component
5,311,081	D. D. Donaldson, R. A. Dame, and R. E. Nikel	Data Bus Using Open Drain Drivers and Differential Receivers Together with Distributed Termination Impedances
5,313,369	M. Lewis, L. Treseder, R. Martinez, and R. Tusler	Reduced Tolerance Interconnect System
5,313,501	C. P. Thacker	Method and Apparatus for Deskewing Digital Data
5,313,577	K. Meinerth, C. Case, R. Gamache, B. Fanning, and C. Franklin	Translation of Virtual Addressing in a Computer Graphics System
5,313,595	M. Lewis and R. Ravey	Automatic Signal Termination System for a Computer Bus
5,315,597	W. C. Mallard and H. S. Yang	Method and Means for Automatically Detecting and Correcting a Polarity Error in Twisted-pair Media

5,315,602	R. M. Arnott, E. S. Noya, and M. N. Rosich	Optimized Stripe Detection for Redundant Arrays of Disk Drives
5,315,696	K. Meinerth, C. Case, B. Fanning, and J. Irwin	Graphics Command Processing Method in a Computer Graphics System
5,315,698	K. Meinerth, C. Case, J. Irwin, and B. Fanning	Method and Apparatus for Varying Command Length in a Computer Graphics System
5,315,707	S. Bryant and M. Seaman	Multiprocessor Buffer System
5,316,642	D. Young, S. Randall, S. Shaw, and A. Wylde	Oscillation Device for Plating System
5,316,965	A. Philipossian, H. Soleimani, and B. Doyle	Method of Decreasing the Field Oxide Etch Rate in Isolation Technology
5,317,527	S. Britton, R. Allmon, and S. Samudrala	Leading One/Zero Bit Detector for Floating Point Operation
5,317,693	J.-C. E. Cuenod, and P. A. Sichel	Computer Peripheral Device Network with Peripheral Address Resetting Capabilities
5,319,385	F. Fernando	Quadrant-based Binding of Pointer Device Buttons
5,319,678	S. Ho and N. Darcy	Clocking System for Asynchronous Operations
5,319,743	S. Dutta, A. Roy, and N. Rao	Intelligent and Compact Bucketing Method for Region Queries in Two-dimensional Space
5,319,760	A. H. Mason, P. T. Robinson, R. Witek, and J. S. Hall	Translation Buffer for Virtual Machines with Address Space Match
5,319,766	B. A. Maskas, J. A. Metzger, N. D. Godiwala, and K. M. Thaller	Duplicate Tag Store without Valid Indicator
5,319,785	K. M. Thaller	Polling of I/O Device Status Comparison Performed in the Polled I/O Device
5,321,373	R. Curtis and B. Shusterman	Combined Differential-mode and Common-mode Noise Filter
5,321,693	R. Perlman	Multicast Address in a Local Area Network Where the Local Area Network has Inadequate Multicast Addressing Capability
5,321,703	L.-J. Weng	Data Recovery after Error Correction Failure
5,321,806	K. Meinerth, C. Case, J. Irwin, A. Masucci, S. Krishnaswami, and A. Moezzi	Residue Buffer for Graphics System
5,321,810	K. Meinerth, C. Case, J. Irwin, and B. Fanning	Address Method for Computer Graphics System
5,325,495	E. McLellan	Reducing Branch Delay in Pipelined Computer System
5,325,528	J. Klein	Distributed Computation Recovery Management System and Method
5,327,416	L. Neville, A. Jain, and A. L. Gutierrez	Surface Selection Mechanism for Optical Storage System
5,327,424	R. Perlman	Automatically Configuring Parallel Bridge Numbers
5,327,435	N. Warchol	Method for Testing a Processor Module in a Computer System
5,329,426	A. Villani	Clip-on Heat Sink
5,330,920	A. Philipossian, B. Doyle, and H. Soleimani	Method of Controlling Gate Oxide Thickness in the Fabrication of Semiconductor Devices
5,331,496	S. Batra and A. Wu	Thin Film Magnetic Transducer with a Multitude of Magnetic Flux Interactions
5,332,487	D. Young, S. Randall, S. Shaw, and A. Wylde	Method and Plating Apparatus
5,333,097	G. Christensen and J. Marceca	Disk Drive Holder and Interconnection System
5,333,098	A. E. Deluca, S. W. Stefanick, C. L. Leo, T. J. Orr, D. T. Symmes, and H. Wright	Apparatus for Storing Storage Devices
5,333,260	R. Ulichney	Imaging System with Multilevel Dithering Using Bit Shifter
5,333,262	R. Ulichney	Imaging System with Multilevel Dithering Using Two Memories
5,333,315	P. Stoppani and C. Saether	Data Storage System and Method with Device Independent File Directories
5,333,744	R.-A. Locicero, S. Morgan, M. Romm, E. Mangan, and M. Bantly	Modular Equipment Support System
5,334,043	G. Dvorak and L. Wolfe	Test Fixture for Electronic Components
5,335,226	R. A. Williams	Communications System with Reliable Collision Detection Method and Apparatus
5,335,235	R. M. Arnott	FIFO: Based Parity Generator (FBPG)

Call for Authors from Digital Press

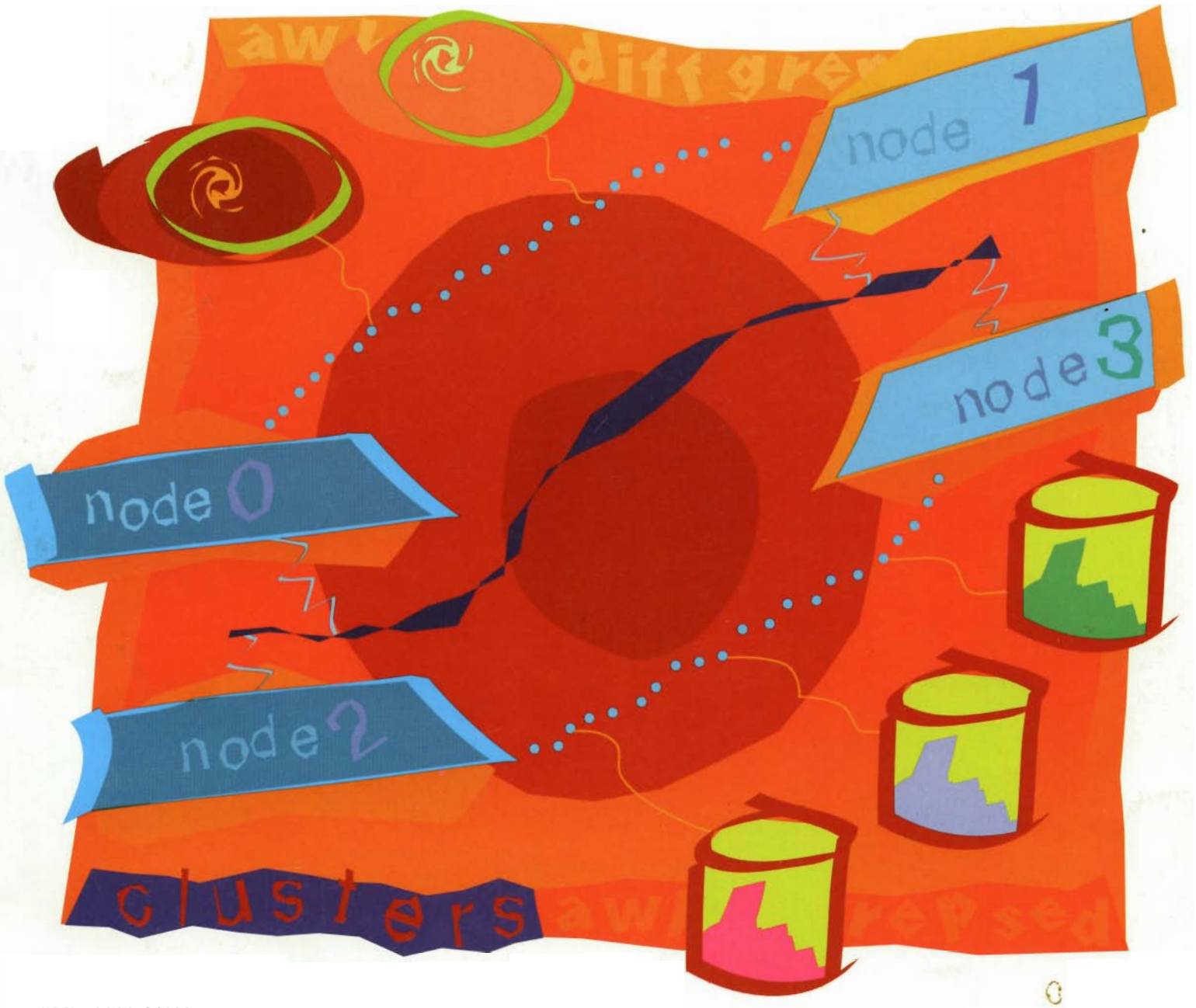
Digital Press is an imprint of Butterworth-Heinemann, a major international publisher of professional books and a member of the Reed Elsevier group. Digital Press is the authorized publisher for Digital Equipment Corporation: The two companies are working in partnership to identify and publish new books under the Digital Press imprint and create opportunities for authors to publish their work.

Digital Press is committed to publishing high-quality books on a wide variety of subjects. We would like to hear from you if you are writing or thinking about writing a book.

Contact: Mike Cash, Digital Press Manager, or
Liz McCarthy, Assistant Editor

DIGITAL PRESS
313 Washington Street
Newton, MA 02158-1626
U.S.A.
Tel: (617) 928-2649, Fax: (617) 928-2640
E-mail: Mike.Cash@BHein.rel.co.uk or
LizMc@world.std.com

digital™



ISSN 0898-901X

Printed in U.S.A. EY-U025E-TJ/96 6 14 20.7