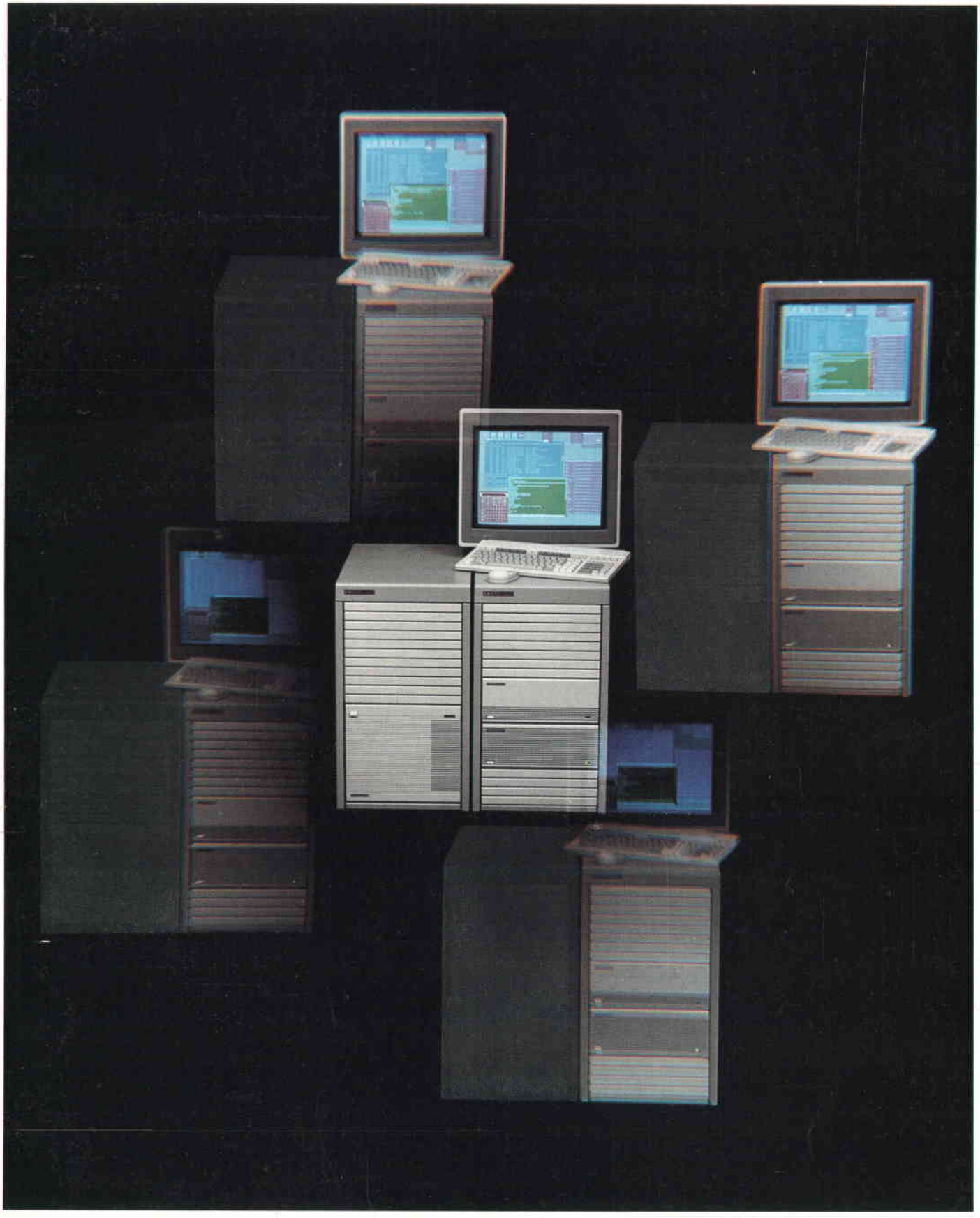
HEWLETT-PACKARD
JOURNAL

OCTOBER 1988



HEWLETT
PACKARD

HEWLETT-PACKARD JOURNAL

October 1988 Volume 39 • Number 5

Articles

Departments

In this Issue

In engineering workstations running under AT&T's UNIX® operating system or one of its many versions, such as Hewlett-Packard's HP-UX, a lot of disc space is used for system code and standard utilities that every workstation must have. When several UNIX workstations are clustered on a local area network, it's natural to think of lowering disc memory costs by storing these common programs at only one workstation and allowing the other workstations to access them over the network instead of having individual copies. To take the idea a step farther, user disc files can also be concentrated at a single workstation, so the other workstations don't need disc drives at all. This is the objective of the HP-UX 6.0 operating system for HP 9000 Series 300 Computers. With this system, tightly networked discless graphics workstations on an IEEE 802.3 local area network can share a single file system server. A simplified, proprietary networking protocol delivers discless workstation performance that comes close to stand-alone performance, while industry-standard networking services, such as ARPA/Berkeley and NFS, are provided for intervendor and intercluster communication and file sharing. HP-UX 6.0 also supports the industry-standard SCSI and VME interfaces, the X Window System, and high-performance HP 9000 graphics subsystems. Major features are the single-system view presented to users and the high degree of network transparency achieved. The system looks the same from any workstation in a cluster, and network operation is transparent to the user. While the idea behind the HP-UX 6.0 system is simple, the engineering was not. Following an introduction to the system on page 6, eight papers discuss the design challenges the development team had to deal with. These include the implementation of a discless file system (page 10), discless program execution and virtual memory management (page 15), network function and protocol design (page 20), crash detection and recovery (page 27), boot mechanism design (page 33), and system configuration (page 37). SCSI and X Window System support are described in the papers on pages 39 and 46.

The August 1988 issue featured the HP PaintJet Color Graphics Printer and its contributions to thermal inkjet printing technology, which include a second-generation printhead design, resolution of 180 dots per inch (nearly double that of the ThinkJet printer introduced in 1984), and full-color printing on paper or overhead transparency film. This issue presents the next chapter of this story, which stars the HP DeskJet printer. Delivering laser-quality printing at 300-dot-per-inch resolution on standard office papers, the DeskJet printer is priced competitively with noisier, less reliable personal printers offering much lower print quality. DeskJet features include merged text and graphics, multiple fonts, two slots for font or personality cartridges, 120-character-per-second letter-quality speed, and a built-in cut-sheet paper feeder. Beginning with management issues on page 51, eight papers in this issue tell the story of DeskJet development. The third-generation, high-resolution, thermal inkjet printhead is discussed beginning on page 55. Electrical connections to the print cartridge, and the systems that hold, move, protect, and maintain the cartridge and fire the ink drops are described in the paper on page 62. The multifunction chassis, designed using an HP CAD system, the paper handling system, an unusual transmission that lowers costs by making one motor perform three functions, and the paper drive motor and its control system are treated in the paper on page 67, while the paper on page 76 tells how a microprocessor-controlled custom integrated circuit manipulates character dot data to provide various text enhancements and graphics. Firmware design is the subject of the paper on page 81, and two manufacturing papers, one on robotic circuit board assembly and one on machine vision systems for printhead production, are on pages 87 and 91.

The shaft encoder that provides feedback for the DeskJet printer paper drive servo system is available to customers as a separate product line, the HEDS-9000 Shaft Encoder Module family. Designed for low cost, rapid assembly, and freedom from follow-up adjustments, this encoder module makes closed-loop operation feasible for low-cost products like the DeskJet printer, where it translates into higher speed and print quality. The HEDS-9000 design includes elements of integrated detector circuits, light-emitting diode technology, plastic optics, and high-volume manufacturing. The story begins on page 99.

-R.P. Dolan

Cover

The cover photograph represents the HP-UX 6.0 discless operating system. The photographer has used a special lens to multiply the image of this HP 9000 Series 300 workstation to simulate a cluster of workstations on a local area network. All but one of the disc drive images have been faded back, indicating that in an HP-UX 6.0 discless cluster, only one workstation needs to have a disc drive.

What's Ahead

The HP NewWave environment, a state-of-the-art user interface for personal computers, is the major subject in the December issue. There will also be articles on the HP 64700 Series host independent emulators for microprocessor-based system development, on the plain paper research that was done for DeskJet printer development, and on a technique for adjusting dual-channel data sampled by the HP 5180A Waveform Recorder. The annual index will also be presented.

The **Hewlett-Packard Journal** is published bimonthly by the Hewlett-Packard Company to recognize technical contributions made by Hewlett-Packard (HP) personnel. While the information found in this publication is believed to be accurate, the Hewlett-Packard Company makes no warranties, express or implied, as to the accuracy or reliability of such information. The Hewlett-Packard Company disclaims all warranties of merchantability and fitness for a particular purpose and all obligations and liabilities for damages, including but not limited to indirect, special, or consequential damages, attorney's and expert's fees, and court costs, arising out of or in connection with this publication.

Subscriptions: The Hewlett-Packard Journal is distributed free of charge to HP research, design, and manufacturing engineering personnel, as well as to qualified non-HP individuals, libraries, and educational institutions. Please address subscription or change of address requests on printed letterhead (or include a business card) to the HP address on the back cover that is closest to you. When submitting a change of address, please include your zip or postal code and a copy of your old label.

Submissions: Although articles in the Hewlett-Packard Journal are primarily authored by HP employees, articles from non-HP authors dealing with HP-related research or solutions to technical problems made possible by using HP equipment are also considered for publication. Please contact the Editor before submitting such articles. Also, the Hewlett-Packard Journal encourages technical discussions of the topics presenting in recent articles and may publish letters expected to be of interest to readers. Letters should be brief, and are subject to editing by HP.

Please address inquiries, submissions, and requests to: Editor, Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304, U.S.A.

Discless HP-UX Workstations

HP-UX 6.0 provides low-cost discless workstation operation over a local area network. It also provides a single file system view, intervendor file sharing, and conformance to UNIX® System V Interface Definition (SVID) semantics.

by Scott W. Wang

THE HP-UX RELEASE 6.0 SYSTEM is a major software contribution to the HP 9000 Series 300 workstation platform. This release of the HP-UX operating system provides discless workstation operation in a network and intervendor file sharing through the Network File System (NFS*).

The HP-UX 6.0 system enables tightly networked discless graphics workstations to share a single file system server transparently in an Ethernet or IEEE 802.3 local area network. Fig. 1 shows a typical HP-UX 6.0 system configuration and defines a few terms that are used here and in other articles in this issue. The terms discless cnode (cluster node) and discless workstation are used interchangeably in this article.

The standard ARPA/Berkeley networking services and NFS complement the tightly coupled workstations by offering intervendor and intercluster communication and file sharing capabilities. In addition to the discless and NFS capabilities, the HP-UX 6.0 system also offers:

- Industry standard Small Computer System Interface (SCSI) and VME support
- Enhanced graphics support for the new HP 98550A high-resolution graphics board and displays and the HP 98556A 2D integer-based graphics accelerator
- Commands and libraries from Release 1.0 of the HP 9000 Series 800 HP-UX system
- The X Window System

SCSI and the X Window System are discussed on pages 39 and 46, respectively.

*NFS is a product of Sun Microsystems, Inc.
UNIX is a registered trademark of AT&T in the U.S.A. and other countries.

Design Goals

There are many ways to implement a discless workstation capability. However, our design choices and implementation techniques were guided by the need to achieve the highest quality goals of functionality, usability, reliability, performance, and supportability. This resulted in the following design goals for our discless workstation implementation:

- Low-cost discless workstation operation over a local area network
- A single file system view
- Conformance to AT&T's *UNIX System V Interface Definition* (SVID) semantics and backward compatibility with previous releases of HP-UX
- A design that coexists with and complements NFS, HP's Network Services (NS), and ARPA/Berkeley network facilities
- At least 80% of the throughput performance of a standalone system (workstation with a disc)
- Flexible system configuration and dynamic reconfiguration
- Thorough usability and reliability testing.

Low-Cost Discless Workstations

Clustering discless workstations is a way to achieve lower cost per workstation, to meet certain environmental conditions (poor environment for discs), and to meet specific ergonomic requirements. To operate in a discless mode the workstation needs access to a remote file server for booting up, for gaining access to files, and for doing virtual memory swapping from the server's disc. Remote

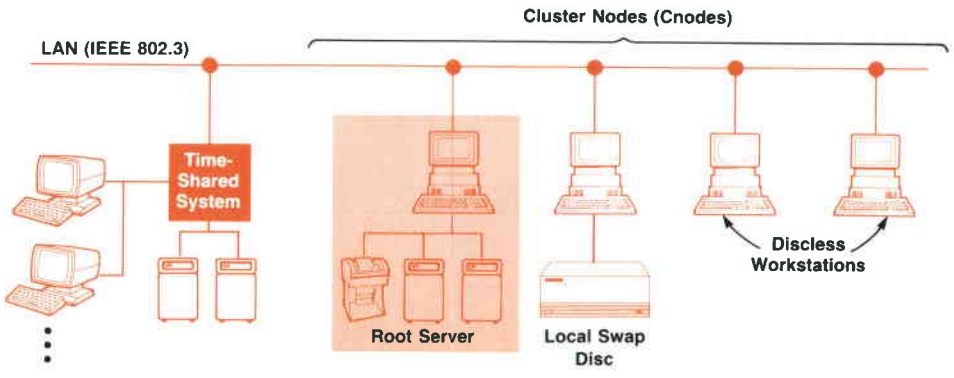


Fig. 1. *Major components of a cluster of discless workstations. A cluster is a group of workstations connected by a network that share a single file hierarchy. A cluster node (cnode) is one of the nodes, or workstations in a cluster. A discless cnode is a cnode that does not have a local file system; its file system resides on the root server. A root server is the cnode to which the disc containing the root file system is physically attached. There is only one root server for each cluster. In the HP-UX 6.0 system release the root server is also the file server.*

boot and virtual memory operations are described in detail in the articles "Boot Mechanism for Discless HP-UX," and "Discless Program Execution and Virtual Memory Management," on pages 33 and 15, respectively.

Single File System View

There are two basic computing environment models: time-shared systems and distributed systems. Time-shared systems allow multiple users to communicate with each other easily, and to share a single computer's environment and resources. The disadvantages of a time-shared system are poor response time, limited configuration and scalability, limited graphics capability, and limited system availability. Distributed systems alleviate many of the disadvantages of time-shared systems by distributing the computing and other resources onto networked full graphics workstations that are smaller and less expensive. However, sharing resources and communicating between users on separate workstations is usually more complex in a distributed system. For the HP-UX 6.0 system we wanted the best of both models: a high degree of network transparency between workstations and a single-system view.

A single-system view in a workstation cluster means the user sees a single file system from any workstation and there is a single point for system administration. A user can log in to the system from any workstation in the cluster and see the same environment in the same manner as seen when logging into a time-shared system from any terminal. Single-point system administration means the system administrator can administer the cluster of workstations from any workstation in the cluster, and the work involved is no more complex than a time-shared system with the same number of users.

Most important, a single-system view in a cluster means a single global file system. Each workstation user sees and shares the same file system just as in a multiuser time-shared system. The implementation of this concept means solving many interesting technical problems. For example, file synchronization needs to be maintained between workstations in the same standard HP-UX semantic exhibited in a multiuser HP-UX system. There are subtleties and implications in performance because of file system buffer caching that involves file accesses in both synchronous and asynchronous modes.

A single-system view also means shielding the user from differences in the workstations in the cluster. In a single cluster, workstations may have different types of CPU (e.g., 68010 and 68020), different floating-point processors (e.g., 68881 versus a floating-point card), and different graphics displays. To solve this problem the concept of *context dependent files* (CDF) was defined and implemented for discless workstations. Each workstation has a context file describing that workstation. CDFs reside in a hidden directory that holds context dependent objects (text files and executables), and maintain the same file path name from any cnode in the cluster. This allows a CDF to be accessed using the same file name from any cnode, with the system automatically differentiating and selecting the proper CDF based on the workstation configuration.

A single-system view in a cluster creates the problem of process ID (PID) collisions between independently executing HP-UX environments in the workstations. Collision must be avoided since HP-UX uses PIDs as unique identifiers in many places (e.g., temporary file names). Similarly, clocks in individual workstations in a cluster must be synchronized to have a consistent time in the cluster. The single file system demands that timestamps on files be consistent no matter which workstation puts the timestamp on the file. This has interesting implications for the `make` command if the clocks are not synchronized.

Additional details on the file system can be found in the article, "A Discless HP-UX File System," on page 10.

Compatibility

Conformance to AT&T's *UNIX System V Interface Definition* (SVID) and object code compatibility with previous releases of the HP 9000 Series 300 HP-UX systems were objectives in all design considerations for the HP-UX 6.0 system. For example, the process ID collision problem mentioned above cannot be solved by simply prepending a cnode ID number to the PIDs to make them unique. Instead, PIDs must remain five digits (1 to 32768) for compatibility. The problem is solved by a PID server process that manages and allocates PIDs in chunks to the discless cnodes while guaranteeing their uniqueness in a cluster. Other examples are file synchronization and file locking, which must be done in a way to preserve standard HP-UX semantics. See the article "A Discless HP-UX File System" for more details. Ensuring conformance to the SVID, the HP-UX 6.0 system has passed the System V Validation Suite (SVVS).

Other Network Protocols

While the discless capability is the primary objective of the HP-UX 6.0 system, another objective was to allow access to NFS, HP's NS, and ARPA/Berkeley network services concurrently with the discless functions. Implementation of these capabilities affects the file system and the network system. For example, the key to a single-system view is the file system. This means we had to integrate all the requirements for other network file systems into the same file system used for the discless implementation.

Discless Performance

In a discless environment, some performance loss is unavoidable because of remote file accessing and virtual memory swapping over the network. The performance goal we set for the HP-UX 6.0 system was 80% of a stand-alone workstation's throughput performance. Three areas were identified as key to achieving this performance: network protocol, virtual memory swapping, and file system caching.

A lightweight protocol was defined to handle the kernel-to-kernel communication between a discless cnode and the server. This resulted in a significant performance advantage compared to other discless implementations based on standard network protocols such as TCP/IP. The discless protocol is discussed in the article "The Design of Network Functions for Discless Clusters," on page 20. A performance analysis is also included in the article.

To address the performance bottleneck of remote swapping at the file server, we include support for local swap discs on a discless cnode. For virtual memory intensive applications running on a discless cnode, the user has the

options of adding a local swapping disc to improve performance while maintaining the single file system view, and of sharing resources with other cnodes.

Standard HP-UX file system buffer caching is maintained on the server and the client cnodes, thus maintaining the performance improvement file caching provides. This is discussed in more detail in the article "A Discless HP-UX File System."

Flexible Configuration

For HP-UX system 6.0, all models of the Series 300 family of workstations are supported. However, the server is restricted to the Series 350 only. Every workstation, including the server, runs the same version of the HP-UX 6.0 system. The server is not a dedicated server, in that it can also be used as a workstation. In addition, the discless cnodes and the server retain their ability to support multiple terminal users if desired. The cluster size and configuration depend on the requirements of users and applications.

Dynamic reconfiguration

Users can add cnodes to or delete cnodes from a cluster and move cnodes from one cluster to another. A cluster can dynamically grow or shrink as necessary.

A cluster can start from as little as two workstations and expand as required without unloading the file system and repartitioning the disc. Discless cnodes can join and unjoin a cluster at boot time without affecting the activity of the rest of the cluster. The new cnode is immediately recognized by other cnodes in the cluster. When a cnode leaves a cluster the rest of the cluster will automatically reconfigure and continue operation. Multiple clusters can be defined on a single LAN and each discless cnode on the LAN can easily choose to join any cluster during boot.

To maintain the single-system view, the configuration of discless cnodes must be as simple as adding a terminal to a multiuser time-shared system. Because of the single file system implementation it is not necessary to partition the server disc according to the number of discless cnodes in the cluster. The file system and swap area on the server disc are shared by all discless cnodes. This allows the system to pool a large swap area when large swap intensive application programs are executed.

Easy cluster definition and configuration are accomplished through a program called reconfig. This is described in the article "Discless System Configuration Tasks," on page 37.

Another example of flexibility and ease of configuration is sharing of peripherals on the server, and the ability to configure local devices on the discless nodes.

Program Management

The HP-UX 6.0 system release was a large team effort spanning many organizations and functional areas. The management of this release was an excellent example of the concept called program management. The organizations involved were HP's System Software Operation (SSO), Technical Workstation Operation (TWO), Corvallis Workstation Operation (CWO), Information Systems Operation (ISO), and Colorado Networks Operation (CND). The functional areas involved included several R&D labs including operating systems, languages, graphics, commands and libraries, networking, performance, system integration, and program management. Other functional areas were product marketing from the various organizations, marketing support, documentation, quality assurance, and manufacturing. In addition, there were many applications organizations such as the Electronic Design Division (EDD) and Logic Systems Division (LSD) that needed to be kept informed of our progress. These two organizations and others were collectively called the Engineering System Group (ESG*) partners.

The program management model centered on what was called the HP-UX team, which consisted of representatives from the various organizations and functional areas. The HP-UX team met weekly for status updates, information, and issue resolution.

Program management documents included the team meeting minutes, a system PERT chart, a program data sheet, a program requirements document, a delta document, commitment lists, and a milestone checklist. Several of these documents deserve further explanation. The delta document was published early for the ESG and other partners. It contained the differences between Release 5.5 and Release 6.0 of the HP-UX system, and items that could affect the partner application and subsystem development. For example, the need for a new boot ROM affected header file changes, object code compatibility issues, and code size estimates. The commitment list was important because it provided us with a central list of all known customer commitments in terms of early release requirements, who made the commitments, when they were required, and whether they required a discless system or just NFS. The milestone checklist was used to track all major action items and all known major and minor milestones. It was reviewed and updated at each team meeting. The checklist not only enabled us to check progress and follow up on action items, but also showed progress being made. The checklist was a great supplement to the system PERT chart which was also reviewed each week.

The partners were kept up to date by means of the delta document and in some cases the team meeting minutes. We also had a monthly (later bimonthly) meeting to share status. This was called the ESG information exchange meeting. It was an effective way to exchange data and keep each other informed. I also served as the major interface to people in California through the periodic HP-UX Steering Council meetings.

The HP-UX 6.0 system program life cycle included three early bird (EB) releases that were roughly two months apart. EB1 was used to tune the processes used to build and put the system through integration and test. EB1 turned out usable enough for distribution to partners and selected customers. EB2 was a functionally complete system for entry into final QA and for partner and customer commitment distributions. EB2 was also used in the first human factor usability testing. EB3 was the final refinement of the product and the release before the final system integration and test process. In essence, the EBs were trial runs for the real release and at the same time served as useful systems.

*ESG is currently called Engineering and Measurement System Group (EMSG).

Scott W. Wang
R&D Lab Manager
Information Software Division

Usability and Reliability

Features that contribute to the usability of the HP-UX 6.0 system include the single-system view, ease of configuration, and compatibility. We worked with human factors engineers to test our early releases for usability. This testing resulted in many changes to the documentation and enhancements to the reconfig program.

Reliability is achieved by extensive prototyping, design reviews, and testing. Besides the typical operating system testing done in the past, we designed and executed additional test cases specifically for the discless cluster configurations. Test clusters were set up to run a networking test scaffold at various stress levels. The HP-UX 6.0 system achieved 120 hours of continuous high-stress operation without a system crash.

The dynamic reconfiguration capability also enhances cluster reliability. When a discless cnode crashes, the rest of the discless cnodes will continue to function unaffected. This requires extensive crash detection and recovery in the operating system. However, the entire cluster will cease to operate if the server with the root file system crashes. To ensure detection of and recovery from LAN cable disconnections without affecting other cluster operations, a cable break detection mechanism has been incorporated into the system. Refer to the article "Crash Detection and Recovery in a Discless HP-UX System" on page 27 for more details.

Acknowledgments

The technology for the discless capability started as a distributed HP-UX (called DUX[1]) project at HP Laboratories in Palo Alto. This research resulted in a prototype implementation of distributed HP-UX that was developed at the Information Software Operation in Cupertino, California and the System Software Operation in Fort Collins, Colorado. DUX incorporated a fully distributed file system and many advanced distributed operating system features.

I would like to acknowledge the many people who made the HP-UX 6.0 system a reality. It is not possible to list all the names here so the list is limited to the core operating system teams.

Xuan Bui and his kernel group: Drew Anderson, Jack Applin, Doug Baskins, Paul Stoecker, Paul Perlmutter, Pamela Marchall. Joe Cowan and his kernel group: Bruce Bigler, Dave Gutierrez, Bob Lenk, Jack McClurg, Bill McMahon, Perry Scott, Rober Quist. Ken Martin and his system integration group: Stuart Bobb, Paul Christofanelli, Jim Darling, Steve Ellcey, Bill Mullaney, Bruce Rodean, Kim Wagner. Marcel Meier and his kernel group: Debbie Bartlett, Mike Berry, Barbara Flahive, Ping-Hui Kao, Anny Randel, Fred Richart. Bonnie Stahlin and her program management and usability/test group: Rich Dunker, Lois Gerber, Dave Grindeland, Mike Steckmyer, Ron Tolley. Donn Terry and his commands and libraries group: Jer/ Eberhard, Gayle Guidry Dilley, Rob Gardner, John Marvin, Rob Robason, and Peter van der Steur.

In addition I would like to acknowledge the California contingent: Ching-Fa Hwang, Joel Tesler, Sui-Ping Chen, Chyuan-Shiun Lin, Doug Hartman, Jeff Glasson, Mike Saboff, and Ed Sesek.

I would especially like to acknowledge John Romano from Logic Systems Division for his early realization that DUX was a must requirement for his HP 64000 market, and Ching-Fa Hwang and his team at HP Labs that built the original DUX: Joel Tesler, Chyuan-Shiun Lin, John Worley, Sui-Ping Chen, Parviz Afshar, Curt Kolovson, and Ray Cheng. Their continued moral support for this project was invaluable.

Steve Boettner, Bill Eads, Gary Ho, Eric Neuhold, and Mike Kolesar provided management support. Finally, a special thanks to Sandy Chumbley, then System Software Operation manager, for sticking with us all the way.

Reference

1. Ching-Fa Hwang, J. Tesler, and Chyuan-Shiun Lin, "Achieving a One-System View for Distributed UNIX Operating Systems," *UniForum 1987 Conference Proceedings.*

A Discless HP-UX File System

by Debra S. Bartlett and Joel D. Tesler

THE MOST OBVIOUS REQUIREMENT of any discless system is a file service capability. All files must be stored on a file-serving node since the discless nodes normally would not have a local file system. The goal of a single-system view for an HP-UX discless cluster imposes an additional requirement—the file system should appear the same from all nodes in the cluster.

Several changes were made to the file system portion of the standard HP-UX kernel to support discless operations. These changes were made with the requirement of maintaining stand-alone HP-UX semantics and file system performance in a discless environment. Elements of the file system that were modified include: file system I/O, named FIFOs, file locking, and pathname lookup.

The discless file system operates in conjunction with the remainder of the kernel and other file systems. In particular, the discless system is designed to work together with the Sun Microsystems Network File System (NFS), which provides transparent access to files on remote machines in a heterogeneous environment. The discless file system design is such that it enhances the functionality of both file systems rather than requiring the user to choose between them.

To understand the discless file system, the reader should be familiar with the standard HP-UX file system. Fig. 1 explains several common file system terms used throughout the remainder of this article.

System Appearance

The simplest way to implement a discless system is to partition the server's discs into multiple subdiscs. Each subdisc would be allocated to one client. The client would treat that disc as if it were local, except that all I/O would be performed over the network rather than directly to disc. While this solution does eliminate the need to attach a disc to each CPU, it fails to meet many of the other needs of a discless system. It is still necessary to provide just as much disc space as it would be if each machine had its own physical disc. Such a system would also provide no file sharing; each machine would have its own set of files. Finally, each file system would need to be independently administered.

Since the above approach has many problems and little benefit, it is rarely used. Instead, a common approach to implementing a discless file system is to provide each node with a small root file system physically located at the disc server. This root file system is private to the node owning it, and contains enough files to boot up the system. After booting, the node issues remote mount requests to mount other shared file systems from the disc server. A remote mount is similar to a normal mount in that it mounts one file system under a directory in another file system. However, the file system being mounted is remote, and is usually shared by several clients. Typically some form of remote file service is used to access the files in a transparent manner.

This approach solves several of the problems of a discless file system, but there are still some limitations that make it unsuccessful in meeting the goals of the HP-UX discless system. Each node still has its own root file system, violating the single-system view. It is possible that the various root file systems will differ from one another. In particular, the disc server's root file system is likely to differ significantly from the client file systems. Each root file system must be independently administered, eliminating the possibility of single-machine administration. Finally, each machine must independently perform the remote mounts. It is possible that different machines will perform different mounts, leading to inconsistent views. Even if the system administrator tries to keep the views the same, it is necessary to guarantee that all updates to the mount table are propagated to all machines, a task that is error-prone.

In the HP-UX discless system, we have chosen instead to have a single root file system, residing at the disc server (also referred to as the root server). All nodes in the cluster (hereafter referred to as cnodes) share the same root. Whenever a file system is mounted at the root server, all other cnodes are notified that the mount has taken place. When a new cnode joins a cluster, it inherits the complete mount table from the server. Since the same mount table is used globally, we refer to it as a global mount table. By sharing the root and the mount table, we provide a one-system view. A user can sit at any cnode and perceive the same file system. A system administrator has only a single file system to administer, and need not worry about propagating changes between cnodes.

Providing a global file system is not sufficient to provide a single-system view. It is also necessary to guarantee that the semantics of file system access throughout the cluster are identical to the semantics used when accessing the files on a stand-alone HP-UX system. Commands to manipulate files must remain the same, the system call interface to the operating system should be unchanged, and applications should not need to know whether they are running on a discless cnode or on the disc server. Furthermore, the semantics used to access files from several cnodes should be the same as if all the accessing programs were running on the same cnode. For example, if one program is writing data to a file, a program reading from that file should see the data immediately after it is written, regardless of whether the reader is on the same cnode as the writer.

Context Dependent Files

The one-system view presented by the discless system has been stressed. Every cnode has the same view of the file system layout, and sees the same files. While this is an ideal situation, there are a few cases where this is actually not the ideal behavior. As an example, consider an application that can make good use of a floating-point co-

processor if it is present, but can run with floating-point libraries if necessary. Some cnodes may have the coprocessor and others may not. It is necessary that the application be able to run on both. While it is possible to link the program with a library that checks for the coprocessor and performs the correct operation for each cnode, this would be inefficient and would not take advantage of the compiler's built-in floating-point code generation capabilities. What is really wanted is two versions of the program: one compiled with the coprocessor code and one compiled without. The user should not need to determine which version to run, but should be able to give the same program name on either type of machine, with the operating system determining the correct program to run. Although there will not be an actual one-system view, since users on different machines will see different programs, there will appear to be a one-system view, since a single program name will attain the same functionality with only a difference in performance.

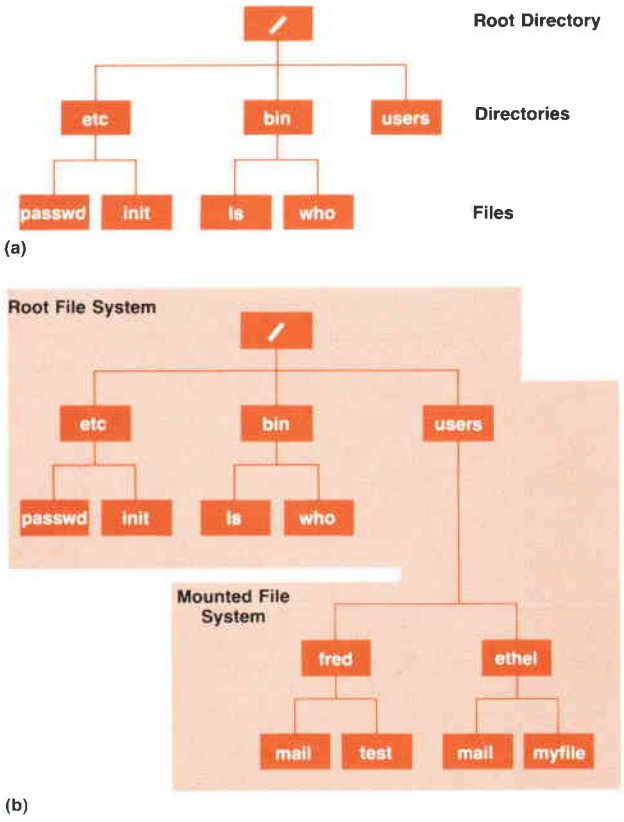


Fig. 1. *Standard HP-UX file system. HP-UX uses a hierarchical file system. One special type of file is a directory, which contains a list of files. These files may themselves be directories, or they may be simple files. The top directory of a file system is called the root, and is signified by /. A directory may be empty. (a) shows a miniature HP-UX file system. The root directory contains three directories,* etc, bin, *and* users. Etc *contains two files,* passwd *and* init. *When writing a file name, the components are separated by slashes, for example* /etc/init. *It is possible to attach other file systems by mounting them on a directory. (b) shows a second file system containing user files mounted under* /users. *Once the mount takes place, the second file system can be accessed as if it were part of the first, e.g.,* /users/ethel/myfile.

Another case where each machine may need a different file is when the file describes the machine configuration. For example, the file /etc/inittab describes, among other things, the terminals connected to the CPU. Each CPU may have a different set of terminals and need a different version of the file. Although it would be possible to modify the format of these files, or to rename them to include the cnode name, various programs depend on the format of the file and would need to be changed if the format or name changes. This could potentially include customer-written programs. Instead, we would like to supply a mechanism for automatically selecting the correct version of /etc/inittab based on the CPU requesting it.

To solve these problems, we have introduced a mechanism called a *context dependent file* (CDF), based in part on the hidden directory mechanism used in the Locus system developed at the University of California at Los Angeles.[1] Each cnode has a set of attributes, defined as the cnode's context. The attributes describe the type of hardware (68010 vs 68020, floating-point processor, etc.) and the cnode's name. A context dependent file consists of a specially marked directory named after the file is made context dependent. This directory is called a hidden directory, for reasons that will become obvious. Within the hidden directory are entries named after the attributes used for selecting the file. When a hidden directory is encountered during a pathname translation, the system searches the directory for an entry that matches one of the attributes of the cnode's context. If it finds one, it automatically "falls through" the hidden directory, selecting instead the matching file. An example may make this clearer.

Fig. 2 shows how /etc/inittab can be set up as a CDF. Fig. 2a shows how the file would normally appear within the /etc directory. Suppose that a cluster has three cnodes named athos, porthos, and aramis. The CDF would be set up as shown in Fig. 2b. The + after inittab indicates that the directory is specially marked as hidden. It is not actually part of the directory name. If a user on athos tries to open /etc/inittab the system will actually open the file athos within the directory. To the user on athos, the file system appears exactly as shown in Fig. 2a. The user on porthos would also see a file system that appears as in Fig. 2a, although the contents of /etc/inittab would be different. Thus, under normal circumstances, the directory is hidden.

Occasionally, the system administrator will wish to see all the contents of the hidden directory. In this case, a special escape mechanism is provided. If a + is appended to the CDF name, it will refer to the directory itself rather than falling through based on the context. Thus, a system administrator on porthos could modify the inittab belonging to aramis by editing /etc/inittab+/aramis. The pathname /etc/inittab+ refers to the hidden directory itself whereas /etc/inittab refers to the machine's own version, in this case porthos.

File System I/O

The standard HP-UX file system buffers I/O requests to increase file system performance. The buffer cache is composed of buffer headers which contain pointers to the actual data. The buffer header data structure also contains a block number and a pointer to a vnode (a data structure describing

a particular file). The block number and vnode pointer are used to identify any block of data pertaining to the file system. When a user makes a read request to the system, the file system first checks to see if that particular block of data is already in the buffer cache. If it is in the buffer cache, then the data can be transferred to the user without incurring the overhead and time it takes to read the data from the disc drive. Likewise, if a user makes a write request, the system will buffer the data and write it to the disc at a later time. This allows the system to buffer write requests into a block size and thus minimize the number of disc writes.

This design of the buffer cache presents some problems when dealing with the discless environment. If each cnode has its own buffer cache, then there is no longer a unique buffer in the cluster's memory for a particular block on the disc. This can lead to synchronization problems. If a user on cnode A writes to a file and if a user on cnode B is reading from that same file, then the data written by cnode A may not be seen by cnode B.

This synchronization problem can be avoided by eliminating the buffer cache on the client cnodes. However, this would create performance problems. The HP-UX discless solution is to implement a compromise. Whenever possible, the discless cnode uses the local buffer cache (asynchronous I/O). When synchronization problems may arise, then the discless cnode bypasses the local cache and reads or writes directly to the server (synchronous I/O). The server always uses its buffer cache.

The determination of whether a data request should be synchronous or asynchronous is calculated on an individual file basis. Each currently referenced file in memory is represented by a data structure called an inode. Part of this data structure contains some fields called cnode maps. There is a cnode map that describes which cnodes have this file open and a reference count for each site that has it open. Likewise, there is a cnode map that describes which cnodes have this file open for write and a reference count for each cnode that has it open for write. These cnode maps are maintained on the server node only. Whenever a file is opened, the referencing cnode's identifier is added to one or both of its cnode maps depending on whether the file was opened for reading or writing. When the open condition is added to the cnode map, a file system algorithm calculates whether this file should be in synchronous or asynchronous mode. If there are no cnodes that have the file open for writing or if the file is being opened for writing and no other cnode has the file open, then the file remains in asynchronous mode. However, if opening this file in the requested mode causes more than one cnode to have the file open with at least one cnode having it open for writing, then the file is switched to synchronous mode. In switching the file to synchronous mode, the system requests that all writing cnodes flush their write buffers to the server and notifies all open cnodes that the file is now to be switched to synchronous mode. The file remains in synchronous mode until a cnode closes the file and that action causes either no more writing cnodes or there is only one cnode with the file open. A cnode using the recently closed file will be notified that it can now switch back to asynchronous mode on the next read or write request to the server.

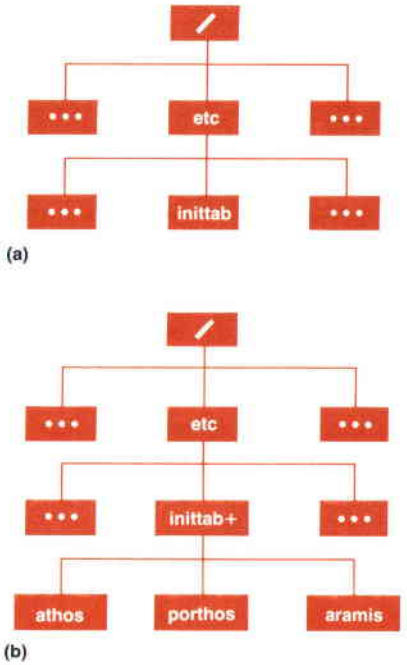


Fig. 2. *Example of context dependent files. (a) Directory structure for* /etc/inittab. *(b) Directory structure for* /etc/inittab *with CDFs.*

In a standard HP-UX system, the buffers associated with a file may stay in memory even after the file has been closed. Thus, if a process reopens that file and makes a read request, it can use the data that is already available in the cache. In a discless environment, this mechanism will not work. For example, suppose cnode A opens a file, reads from the file, and closes the file. Then cnode B opens the file, writes to the file, and closes the file. Now cnode A reopens the file. The buffers at site A no longer contain the correct data because cnode B has modified the data.

To take advantage of buffer caching and avoid this synchronization problem, there is now a version number associated with each file. When a file that was in asynchronous mode and has been written to is closed, the version number is changed on the server and at the cnode that closed the file. When the file is reopened and the inode is still in memory on the requesting cnode, then the old version number is checked with the current version number. If the version number is the same, then the old buffers can be used. However, if the old version number is less than the new version number, then the old buffers are invalidated.

Another consideration with buffering in a discless environment has to do with disc space allocation. In a standard HP-UX environment, when a write request is made, the system first checks to see if there is enough space on the disc for the write request. If there is not enough space left for the request, then the write fails with an error message. In a discless environment, it would help performance if each write request did not have to go to the server to ask for a disc block number. However, if it did not do this, then a user might think that a write has succeeded, but by

the time the actual asynchronous write operation goes to the server, it may fail because of no disc space. To avoid this problem, which does not occur on stand-alone systems, a nearly-full-disc algorithm has been established. The algorithm is based on knowing the number of total buffers in the cluster. Once the disc gets to the point where it does not have enough free disc space for all buffers, it notifies the discless cnodes. After this point, whenever a discless cnode makes a write request that would require space on the disc, it makes the write synchronously to the server.

FIFO Files

In standard HP-UX, named FIFO files, also known as named pipes, are a mechanism for processes on the same machine to communicate with each other. Each process opens the same named FIFO file. Then each process uses the read and write system call to send and receive information to and from other processes. The discless implementation extends this concept so that processes on different cnodes can communicate via the same named FIFO file.

The in-memory inode for a named FIFO file contains specific fields related to that FIFO file. The specific information associated with a FIFO file consists of the read count, the write count, the current read pointer, the current write pointer, and the number of bytes in the current FIFO file. The FIFO file is maintained as a circular 8K-byte buffer. On the serving cnode, the inode contains cnode maps which specify the cnodes using the FIFO file. If only one cnode is using a particular named FIFO file, then the FIFO file specific information is maintained on the cnode that is actually using the named FIFO file. This improves performance, because the cnode does not have to communicate with the server every time it accesses the FIFO file. If another cnode opens that same FIFO file, the server recognizes that there is now more than one cnode using the FIFO file. The server then requests that the current cnode that is using the named FIFO file send all of its FIFO file specific information and data to the server and that from now on it send its read and write requests to the server. In this way, the server acts as the focal point for all communication between the cnodes.

Lockf

The discless implementation of file locking maintains the full standard HP-UX semantics. HP-UX provides a byte-level locking mechanism for regular files. There are advisory locks and enforced locks. Advisory locks can be used as semaphores between processes. If a file region has an enforced lock, then only the process with the lock can read from that region or write to that region.

Advisory locks are implemented with the lockf or fcntl system call. These system calls allow a user to inquire if there is a lock on the file, to test and lock a region, to lock a region, and to unlock a region. In the nondiscless version of lockf, file locks for an open file are kept in the inode structure. In a discless environment, the inode can be on more than one cnode at any given time. Thus, it must be decided where the locks will reside for a file so that everyone will know about them. One possibility is to keep all locks on the server. This is a simple implementation; however, it has the disadvantage that if a cnode has a file open with locks, then all inquiries must go to the server. The implementation that was chosen is to have each cnode keep the locks that were originated by that cnode and to have the server keep track of both the local and remote locks. Thus, if a cnode with a lock on a remote file makes a lock inquiry, the lock will be found on that cnode and it will not be necessary to send a message to the server.

If a file has enforcement mode locks on it, then each read or write system call must check to see if another process currently owns a lock in the specified read or write region. If another process does own a lock, then the requesting process must wait until the region is unlocked. When checking for other processes, it is only necessary to check on the serving cnode when a file is opened by more than one cnode and there are enforced locks on that file. The same mechanism used for keeping track of file-open requests for asynchronous and synchronous file I/O is used in this situation as well.

In the standard HP-UX version of lockf, deadlock prevention checks are done before granting a lock to avoid potential deadlocks. The basic deadlock detection algorithm is as follows. The code first looks at the status of the process that owns the lock. If the process is not waiting or is waiting for something other than a file lock, then there is no deadlock. If the owning process is waiting on a file lock, a search is initiated using the lock this process is waiting on. If the search finds the lock owned by the calling process, then a potential deadlock has been found.

In a discless environment, there are more potentials for deadlock. Therefore, the deadlock detection algorithm was enhanced to account for these situations. The differences for finding deadlocks are the result of three conditions. First, processes in the waiting chain may be distributed throughout several cnodes. Second, a process may be sleeping on a lock or may be waiting for a cluster server process on the root cnode that is itself waiting on a lock. Third, more than one process may simultaneously try to wait on a given lock as a result of concurrent deadlock searches happening on more than one cnode.

Pathname Lookup

An important job provided by the file system portion of the kernel is the translation of a user-specified pathname into its location on the actual disc file system. For example, in the open system call, the user specifies the file name to be opened such as /dir1/dir2/dir3/file. The system then internally translates each component of /dir1/dir2/dir3/file until it has found the inode number representing /dir1/dir2/dir3/file. The system then reads this inode from the disc to determine its characteristics and the location of its data blocks. Many of the system calls pass a pathname. Examples of pathname system calls are open, creat, stat, link, and exec.

For the discless implementation, the pathname lookup code was modified. First, the code recognizes whether any component of the pathname is remote, that is, it belongs to a file system physically attached to another cnode. If the pathname is remote then the code sends the entire remaining pathname to the serving site.

To reduce the number of messages that must be communicated between the server and the requesting client, the pathname lookup code was also modified to send not

only the pathname, but also all the necessary information to complete the system call while it is still operating on the serving site. This mechanism is table driven. Associated with each pathname lookup system call there is an opcode and a structure which describe the request size, the reply size, the function on the client side that will package the required information, the function on the server side that will perform the requested operation, and the function on the client side that will unpack the request.

For example, the opcode for open is 1. Its packing function is open_pack(). Its serving function is open_serve(). Its unpacking function is open_unpack(). The function open_pack() establishes the mode to be used for opening the file and the file mode to be used if the file needs to be created. The function, open_serve() handles the requirements for the opening, such as permission checking on the file and creation of the file if necessary. The function open_unpack() allocates an inode for the file, marks it as asynchronous or synchronous, and opens the device if it is a device file.

Interactions with NFS

In addition to the discless product, another form of remote file sharing is available with the HP-UX 6.0 system, namely NFS. NFS provides the ability to mount remote file systems. This raises a couple of questions. First, why are both NFS and the discless system needed and why can't discless be based on NFS? Second, given that both systems exist, how do they interact?

NFS is a de facto industry standard for sharing files among heterogeneous machines running different operating systems. Being general-purpose, however, it tends to impose constraints. For example, the network protocol used with NFS needs to be able to deal with routing. Also, to keep NFS simple, it does not obey full UNIX semantics. For example, it does not provide file synchronization. Finally, NFS uses a remote mount model, preventing a true single-system view. The discless system is designed for a cluster of machines with a high degree of sharing. It provides a single-system view within a cluster, but does not provide any access to machines outside the cluster. Because it has a specialized purpose, it can be optimized for that purpose. For example, because it only operates over a single LAN, it uses a very-low-overhead networking protocol with minimal need for error detection and routing. Also, the discless system maintains full HP-UX semantics including all UNIX semantics.

Since both NFS and the discless system exist within the same system, they need to coexist, preferably in a mutually beneficial manner. Indeed, each system complements the other. A cluster of workstations can replace the traditional single time-shared machine, with the workstations sharing the view of the file system, just as users at terminals on a single machine share that view. In the same manner that a user can move between terminals on a time-shared machine without noticing a difference, a user can move between workstations in a discless cluster without noticing a difference. NFS can then be used to access machines outside the cluster, just as it can be used from a time-shared machine to access other machines. To maintain the single-system view within the cluster, the NFS mounts must be global in the same manner that local mounts are: when one cnode mounts a remote NFS file system all other cnodes must see that mount also.

Acknowledgments

Sui-Ping Chen, Barbara Flahive, Ping-Hui Kao, Curt Kolovson, and Fred Richart all contributed to the development of the discless distributed file system. Sui-Ping worked on context dependent files, Barbara worked on lockf and nonpathname related system calls, Ping-Hui and Curt worked on pathname lookup and pathname lookup related system calls, and Fred worked on mount and buffer management. Mike Berry and Fred Richart developed the distributed test tools and helped write the distributed test suites for the file system.

Reference

1. G. Popek and B. Walker, *The Locus Distributed System Architecture*, MIT Press, 1985.

Discless Program Execution and Virtual Memory Management

by Ching-Fa Hwang and William T. McMahon

MANY DISTRIBUTED SYSTEMS based on the UNIX® operating system offer some form of remote file access capability. However, only a few of them provide discless workstation capability, particularly in the area of paging, swapping, and execution of programs over a network. Almost all the remote file access systems assume a well-defined client/server model. Some of them have been implemented in a machine or system independent fashion and adopted as industry standards for porting to different vendors' systems. Discless workstations, on the other hand, have been offered only as proprietary systems up to this point. It is unclear at this time if any implementation will be successfully adopted as an industry standard.

The disparity between the remote file access and remote program execution implementations can be attributed to several things. Unlike the UNIX file system which has a well-defined and machine independent structure to facilitate the definition of a client/server model, the implementation of virtual memory (VM) for paging, swapping, and execution of programs over a network is not isolated from machine architecture. For example, 4.2BSD and AT&T System V are two primary bases for most vendors' UNIX implementations, but their virtual memory implementations are based on quite different machine architectures, and their performance characteristics are tuned to their native machine architectures. This creates more difficulties in defining a client/server VM model for implementing paging, swapping, and execution of programs over a network.

The key technical challenges for implementing paging, swapping, and execution of programs over a network in an HP-UX environment include: preservation of the behavior and semantics of existing program types (such as preloading versus demand paging), efficient and flexible global swap device management, and performance that is good enough to justify the cost of discless workstations. This paper describes some of the design issues and our solutions in overcoming these technical challenges.

Overview of the HP-UX Discless Cluster

To support the one-system view, an HP-UX discless cluster has one global file system. The file system on the server node supplies all the program files that can be executed from any node (called cnode) in the cluster—as transparently as if they were executed on one system running the standard HP-UX operating system. To complete the one-system view, it should be possible to execute program files from standard HP-UX releases of the past on an HP-UX discless workstation without a recompilation. This backward compatibility applies to the various types of loading, paging, and swapping mechanisms available in the standard HP-UX environment. Loading refers to bringing a program from the file system and setting up the appropriate process control and memory mapping structure for program execution. Swapping refers to copying some of the process control structure and all the remaining pages to or from a swap disc. Paging means copying some of the referenced pages of a program to or from a swap disc.

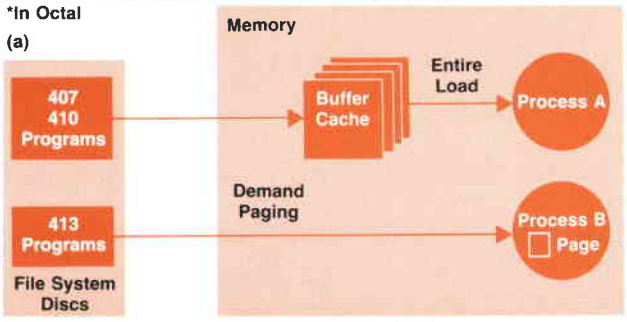
For the discussion in this paper, an HP-UX discless cluster may consist of two kinds of cnodes: (1) swap servers with local swap devices to provide swap space to their clients, and (2) swap clients without local swap discs. A swap server can also be used just like a client cnode. The swap server/client relationship is analogous to the file server/client relationship. The former describes swap space and device services and the latter file services. The common swap space pool is shared equally among all clients of a swap server (including itself as the local client). The common swap space can be expanded to multiple discs by using the HP-UX command swapon to add more swap discs. A swap client can be dynamically added and removed from the server without bringing down the entire cluster for swap space or disc reconfiguration. The server dynamically allocates swap space to its clients when needed and deallocates it when not needed. On detecting the failure of a client site, the server automatically returns the space allocated to the failed client to the common swap space pool.

The features mentioned above are considered design objectives for supporting the one-system view and achieving the high performance requirement for HP-UX discless clus-

*Execution Type	Execution Characteristics			
	Non-sharable	Sharable	Entire Load	Demand Paging
407	X		X	
410		X	X	
413		X		X

*In Octal

(a)



(b)

Fig. 1. *(a) Standard HP-UX program types and their execution characteristics. (b) Loading behavior of the different program types.*

UNIX is a registered trademark of AT&T in the U.S.A. and other countries.

ters. However, several simplifications and restrictions were placed on HP-UX 6.0. Specifically, there can be only one swap server per cluster and the swap server must be the same cnode as the root/file server. As an option, a nonroot cnode is allowed to have local swap discs for improving local swapping performance. In this instance, only the specific cnode has access to the local swap disc.

Program Execution

To help understand the complexity of handling program execution and the interactions between the file system and virtual memory in a discless environment, we use the following scenario to illustrate the interaction in a stand-alone standard HP-UX environment. The scenario describes what happens externally in the user environment and internally in the system when an application program called foo is updated. In the rest of this paper the term "update," when used in the context of program or executable files, refers to the point at which an existing executable file is replaced with a new version of the program.

To begin the scenario, a programmer has just completed a new version of foo with the file name of foo.new, and is ready to release it while some users may still be in the midst of executing the old foo. Internally the system may have kept the program data and control information in several places, depending on the exact stage of execution. For example, the program file in the file system on disc may or may not have been fully brought into memory for execution, and part or all of the program may have been paged or swapped to a swap disc to free the memory for other process executions.

To release the new foo, the programmer types in a mv foo.new foo command. The HP-UX system will detect that the program is currently busy for execution and therefore reject the command by returning an error (ETXTBSY). Not until the last user has completed the execution of foo will the programmer be allowed to update foo. When the system detects that no other user is executing foo it will honor the mv command by copying from foo.new to foo as a normal file system operation. While doing this, the system will also invalidate all the memory pages that may still have cached data of the old foo. If a user tries to execute foo before the mv replacement operation is finished, the system will prevent execution since foo is in an inconsistent state.

In the HP-UX discless cluster, our goal was to preserve the features described in the scenario to support the one-system view and to satisfy performance requirements. This required us to consider that a program being executed may have parts paged or swapped out to the swap discs of the swap server or cached in the memory of different cnodes. In addition, the program may be requested for update; therefore, it was necessary for us to consider mutual exclusion as explained later. Many of the standard HP-UX internal mechanisms and algorithms are not adequate for handling these situations, so enhancements and new algorithms were added to handle the discless environment.

Program Loading/Swapping/Paging

In the standard HP-UX system, when programs are compiled or loaded (through cc or ld commands) the control options 407, 410, or 413 (octal) can be included in the command string to designate the loading, swapping, and paging characteristics of the program. A 407 program requires each process invocation to have its own copy of the program in memory and not be shared by multiple processes. 410 and 413 programs can be shared among multiple processes to save memory space. 407 and 410 programs need to be loaded in their entirety from the file system before the execution can begin, while a 413 program can be loaded in by pages on demand (i.e., page fault). The loading of a program file in its entirety from the file system for execution is handled through file system I/O via the buffer cache, while the paging activities, either with the file system discs or the swap discs, use device I/O directly, bypassing the file system and buffer cache (see Fig. 1).

To maintain the identical behavior and semantics of the three program types for backward compatibility and performance, the discless file system provides a mechanism for bringing in remote program files from the file server for execution. We also implemented a remote device access mechanism that allows devices at a remote cnode, or the device server, to be accessed over the network. This remote device mechanism provides the necessary mechanism for handling paging I/O directly with either the remote file system discs or the remote swap discs. These two mechanisms provide a means for loading and/or paging the three program types.

Mutual Exclusion with File Update

In the standard HP-UX system, executable files are usually in one of two mutually exclusive states: update and execute. A file can be brought from disc to memory for either updating or execution by one or more processes, but never for both updating and execution at the same time. However, a file can still be opened for reading while in either state. Before allowing the execution of a program, the system internally checks that no process is opening the file to write to it. Likewise, when a process is ready to open the file for writing the system also checks to see that the file is not being executed by any process before it enters the update state.

In a discless environment maintaining mutual exclusion is more complex because the processes that want to execute

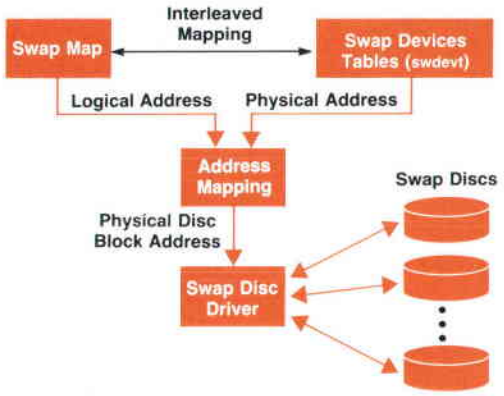


Fig. 2. *Mapping from logical address to physical disc blocks on the swap discs in standard HP-UX.*

or update the file may come from multiple cnodes in the cluster. Therefore, mutual exclusion must be enforced in the context of the entire cluster. To address this issue, the root server was selected as the place to enforce and coordinate mutual exclusion for the files it serves. For each file being referenced or executed, the file inode, which is an internal data structure containing a description of the file, contains entries called cnode maps. The cnode maps are used to track program execution and program file updates. The cnode map for execution keeps track of the cnodes in the cluster executing the program and keeps a reference count of the number of instances of execution of a particular program at each cnode. The execution cnode map and the write (update) cnode map together provide the root server with the necessary information to enforce mutual exclusion. For more information about the inode and cnode maps for file updates refer to the article "A Discless HP-UX File System" on page 10.

Client Caching for Performance

In the standard HP-UX system, caching for program execution is provided to improve execution performance. When a process is terminated or when its pages are paged or swapped to a swap device, the memory pages are freed but also marked as reclaimable. This denotes that these pages can be reactivated if the data on the pages is referenced again before the pages are allocated to other programs. Like the buffer cache for minimizing file system I/Os, reclaimable pages are intended to minimize I/O overhead for paging and swapping. The effect is especially significant when a file is repeatedly executed by one or more processes.

To maintain cache consistency, when a program file is updated, the system automatically invalidates the file's reclaimable cache pages left from previous executions. This ensures that no future executions of the same file will get out-of-date data from the reclaimable cache pages. Similarly, when a file is to be executed by a process, any file data remaining in file buffers resulting from delayed or asynchronous file system writes will be flushed to disc first. This is necessary to ensure that when the program is paged in directly from discs, the file on the disc is up to date.

The standard HP-UX system keeps cached data around as long as possible. Flushing file buffers or invalidating reclaimable pages is always delayed until it is absolutely needed to maintain system consistency. This is the kind of optimization policy that we wanted to keep for HP-UX discless clusters. However, cache consistency in a discless environment is much more complicated than in the standard HP-UX system because buffers for file updating can potentially exist on multiple cnodes, and reclaimable pages for an executing program file can also exist on multiple cnodes. To maintain cache consistency in a discless environment we had to ensure that:

- For program file updates, all reclaimable pages for a particular file are invalidated throughout the entire cluster.
- Before a 413 program enters the execute state, all the file buffers associated with the program file are flushed over the network to discs at the server.

To improve performance further an extension is included in our reclaimable page invalidation mechanism. When a file is updated, instead of starting a global operation to invalidate all the reclaimable pages on all cnodes, the invalidation is individually handled and deferred for each cnode until that cnode is ready to execute the file again. Basically, we include a version number in the in-memory inodes at both the server and the clients. The version number is incremented in the inode at the server whenever the file is closed for update, but is incremented at a client only when the client is ready to access the file. When a client is about to execute a file, the version number of the file at the server is compared with that at the client. Only when the two numbers are identical will the local reclaimable pages at the client cnode be kept for possible reuse. All the other cases will cause these reclaimable pages at the client cnode to be invalidated.

Swap Space Management

In the standard HP-UX system information about swap discs is set up at boot when the system is reconfigured, and is kept in the swap device table (swdevt). By default, swap space is first set up on the root disc and other discs can be added by the use of the swapon command. The information in the swdevt data structure is used to build the system swap map which is used to represent and keep track of the pool of available swap space (see Fig. 2). Swap space is interleaved among all the swap discs, and when swap space is allocated, the first chunk of swap space is taken from the first swap disc in the swdevt, the second chunk of swap space is taken from the second swap disc, and so on.

When a process needs swap space it grabs it from the system swap space pool. When a page is paged out to the disc the logical address in the swap space pool is mapped onto a physical disc block using the information in the swap devices table. When a process requests swap space

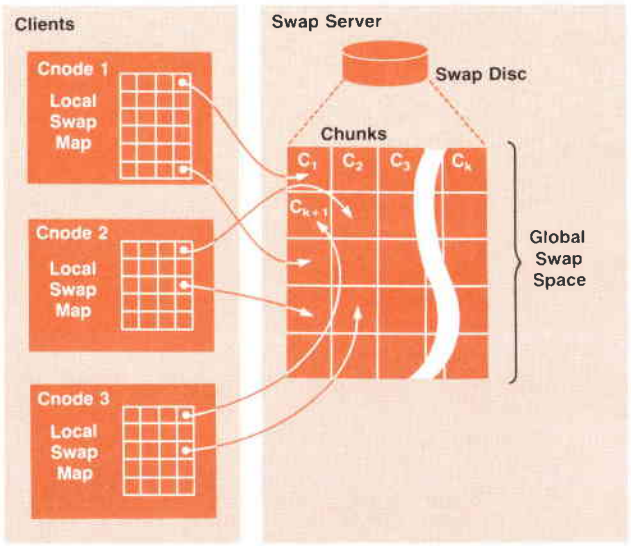


Fig. 3. *The swap space is physically located on the server and is allocated to the client cnodes in chunks.*

and there is no more space in the swap space pool the process is killed or an ENOMEM error is returned to the process.

Design Considerations

One of the primary considerations in designing the remote swap mechanism for the HP-UX 6.0 system was to maintain as much of the current interface as possible. We wanted the swap device table to continue to specify the swap discs, and the use and availability of swap space to be represented by a swap map. Efficiency was important, so we had to consider methods to minimize the number of requests for swap space made from a client cnode. If the swap maps were maintained only on the server all requests for swap space would be a remote request, whereas if each cnode maintained its own swap map it would only need to make a remote request when its local swap map indicated that it was out of swap space. For these reasons two new concepts were introduced: the global swap space and the local swap space. The global swap space represents the total amount of swap space that is allocated to all clients and exists on the root server. The local swap space is the portion that is allocated to a particular cnode. Each cnode has a local swap map which is used to map from the local swap space to the global swap space (see Fig. 3). Efficiency was also considered in determining the granularity of requests for swap space. In the HP-UX 6.0 system we adopted a wholesaler/retailer allocation scheme. The swap server, functioning as a wholesaler, allocates the swap space in large chunks (in megabytes) to its clients; each client in turn allocates the space (in tens or hundreds of kilobytes) to each local process.

Another must objective was to allow dynamic reconfiguration of the cluster without bringing the entire cluster down. We did not want the cluster to be wasteful of space, and a fixed, permanent allocation of swap space to every cnode would have been very wasteful. This meant that allocation of swap space to a client had to be dynamic and that swap space had to be returned when not used or when a cnode crashed or rebooted.

We wanted to provide a way to limit the amount of swap space a cnode can consume and also to provide a way to ensure that a minimum amount of swap space was always available. Other design questions that were raised but not implemented in the HP-UX 6.0 system were whether one cnode should swap to more than one swap server and whether a cluster should support more than one swap server.

Local and Global Space Mapping

To understand how paging and swapping works in a discless environment it is necessary to understand the division between local and global swap space, and how the local swap space maps into the global swap space. A new data structure called a *chunk map* was established to represent the global swap space. The chunk map exists on the server only and its size and initial swap space information are derived from the swdevt at the server as had been done with the swap map in the standard HP-UX system. Each entry in the chunk map represents one chunk of swap space on the disc. Mapping from chunk map address to disc block is done by using the swdevt as it is done in the standard HP-UX system when mapping from the swap map to the disc block. The information maintained in the chunk map consists of the chunk size, the number of the cnode owning the chunk, a bit to indicate if the chunk is valid, and a bit to indicate if the chunk is allocated (see Fig. 4). The sizes of all chunks are defined by a system global variable called dmmax. The number of the cnode that owns a particular chunk is kept as a sanity check and for crash recovery. The valid bit is set when a swapon is done and the chunks on the new swap disc become available for swapping. The allocated bit is set when a chunk is assigned to a cnode.

It is still necessary to complete the mapping from local swap space to global swap space. This is accomplished through the use of a data structure called a *chunk table* which exists on each client cnode. Since the swap discs are located at the server the chunk table acts as a logical swdevt for the client cnodes and provides the first step in the mapping from the local swap map to the global swap space on the server (see Fig. 5). Each entry in the chunk table represents one chunk of space in the local swap map. Conversion from a logical swap address to a chunk table entry is done by dividing by dmmax. Each entry in the chunk table contains the chunk size, a chunk map index, a valid bit, and a reference bit. The chunk size is the same as the size of the corresponding chunk map entry. The chunk map index is a pointer to the corresponding entry in the chunk map on the server. The valid bit indicates if the entry is valid (and hence represented in the local swap map), and the reference bit indicates if it is being used by any process. The chunk table is maintained at every cnode including the swap server. The chunk table allows the swap map to be more easily maintained at each cnode. The reference bit provides the information to return swap space from a cnode that is not using it.

The mapping from a client's local swap map to disc blocks on the server can be briefly summarized as follows. When a request is made from a client cnode to send some data to swap, the system converts the logical address in the local swap map to an index into the chunk table (divi-

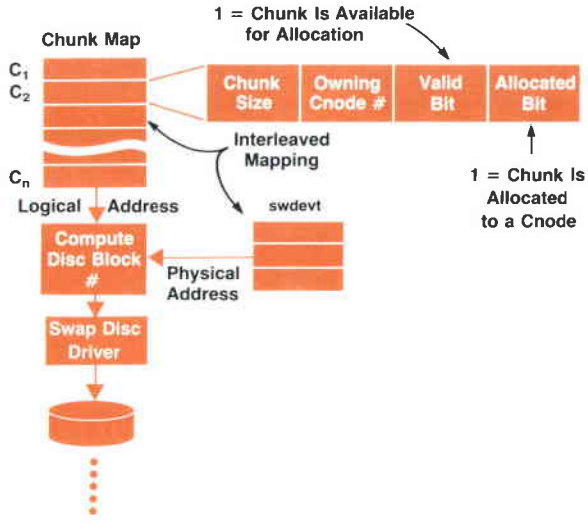


Fig. 4. *The chunk map entry.*

sion by dmmax) and an offset (the remainder). The entry in the chunk table yields the index of the chunk map entry at the swap server. This index and the offset are sent over to the swap server along with the data. On the swap server the system takes the chunk map index and offset and uses the rules for interleaving swap space and the information in swdevt to generate the disc block number for the request. The request is then sent to the disc driver for that device.

Allocating Swap Space

When a process requests swap space and does not find any in its local swap map the process is not immediately killed or an error returned. Instead, the process makes a remote request to get more swap space from the global swap space pool. The swap server allocates another chunk to this cnode and the cnode adds an entry for this chunk to its local swap map and grants the process's request.

When a process makes a remote request for another chunk of swap space it goes to sleep at that point. It is then possible for another process to come along and request a piece of swap space and go to sleep. This could result in much more swap space being requested than is needed because client cnodes allocate swap space to local processes in sizes that are much less than the chunk sizes from the server (i.e., 10 or 100 kilobytes versus several megabytes). To prevent this from happening, a lock was introduced to serialize the requests. When the first remote request is made the lock is grabbed and is not released until the additional swap space is added to the local swap map. When each of the other processes acquires the lock, each one reevaluates whether there is sufficient swap space available.

Returning Swap Space

One of the design decisions was to allow a cnode to return swap space that it is not using. However, it is not efficient for a cnode to return swap space immediately since it may just turn around and request more space. To prevent this type of thrashing, the reference bit and a daemon process are used to check for unused chunks. If an unused chunk is found, the reference bit is cleared. On the next invocation of the daemon, if the chunk is still unused, then it is returned to the swap server; otherwise the reference bit is set. The daemon is set to run once every 30 seconds, so unused swap space is returned between 30 seconds and one minute after it becomes unused.

When a cnode is removed from a cluster either by crashing or by being rebooted, it is necessary to return the swap space to the swap server. This happens when recovery is done for that cnode on the swap server. Recovery is very simple for swap space. When recovery is conducted on the swap server the system routine simply goes through the chunk map, and when it finds an entry that was allocated to the crashed cnode it marks that entry as being available again (allocate bit = 0). Crash recovery is discussed in the article "Crash Detection and Recovery in a Discless HP-UX System" on page 27.

Controlling the Amount of Swap Space

Two configurable parameters are provided for controlling the amount of swap space allocated to a cnode. These are MINSWAPCHUNKS and MAXSWAPCHUNKS. MINSWAPCHUNKS

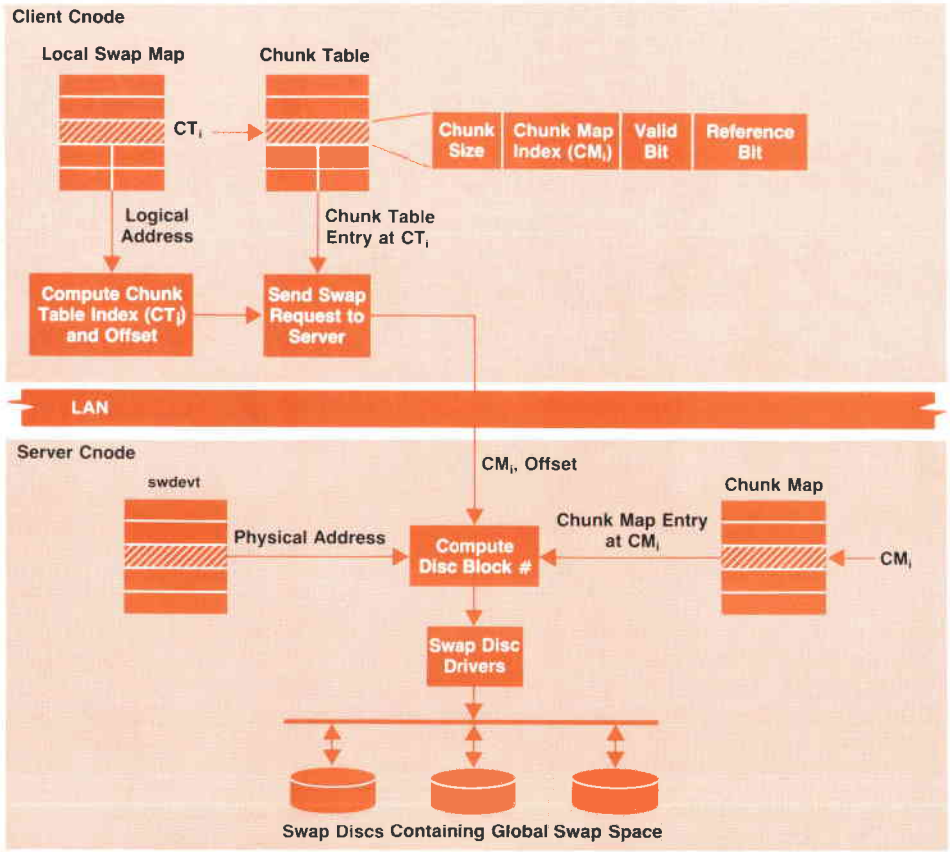


Fig. 5. *The relationship between the local swap map and the chunk table on a client cnode and the chunk map and swap device table on the server.*

specifies the minimum number of chunks of swap space a cluster can have even when the space is not actively used. It is the amount requested at boot and it is never returned. It ensures that a particular cnode will have at least that amount of swap space. MAXSWAPCHUNKS specifies the greatest number of chunks of swap space that a cnode can ever have.

Summary

In summary, the HP-UX 6.0 system provides a fairly complete implementation for HP-UX discless program execution and virtual memory management. Among the features provided to this end are backwards compatibility for executable files, remote swap services, and HP-UX semantics for executable files. New mechanisms are included to minimize performance degradation over a network.

The Design of Network Functions for Discless Clusters

by David O. Gutierrez and Chyuan-Shiun Lin

WITHIN AN HP-UX DISCLESS CLUSTER, the kernel of the client and server machines uses a simple, fast, and reliable network protocol to communicate through a single IEEE 802.3 10-Mbit/s local area network (LAN). The discless protocol is based on the request/reply model and its interface to the HP-UX operating system is specially tailored for efficient data transfer. To become a viable product, a discless system must provide a level of performance comparable to that of systems with local discs. Measurements on HP 9000 Model 350s show that remote file I/O throughput performance of the HP-UX 6.0 discless implementation using an HP 7958A Disc Drive is 91% of stand-alone performance in read operations and 87% of stand-alone performance in write operations when transferring large files. This performance level is achieved by a low-overhead network protocol, efficient network buffer management, cluster server processes, and carefully implemented read/write algorithms.

A cluster consists of a single file server and a number of discless client machines connected by a single LAN cable or several cables connected by LAN bridges, hubs, or repeaters. Multiple clusters may exist on the same cable. Each node of a cluster is called a cluster node (cnode) and has its own hostname and internet address. The central file server is called the root server (shortened to server in this paper) and is where all file systems and disc storage reside. The operating system software is designed to handle up to 255 client machines and each cnode is assigned a number from 1 to 255. The kernel network functions map the cnode number to the appropriate source/destination address.

The discless network protocol is designed specifically for the HP-UX discless kernel and not for general-purpose network communication. Experimentation indicates that packet loss on a single local area network is rare, and by limiting the design scope to providing intracluster network service on a single local network, we can function with a simple network protocol. The discless network functions and the kernel functions are closely tied together to minimize the path length for sending and receiving messages. General-purpose networking services are still available throughout the discless cluster to provide standard communications with the outside world.

A major source of communication overhead in normal network operations is copying data between network buffers and user buffers. It is important to minimize such copy operations. In most network systems, messages are copied from an operating system buffer into network software buffers and then into the network I/O hardware buffers. In the

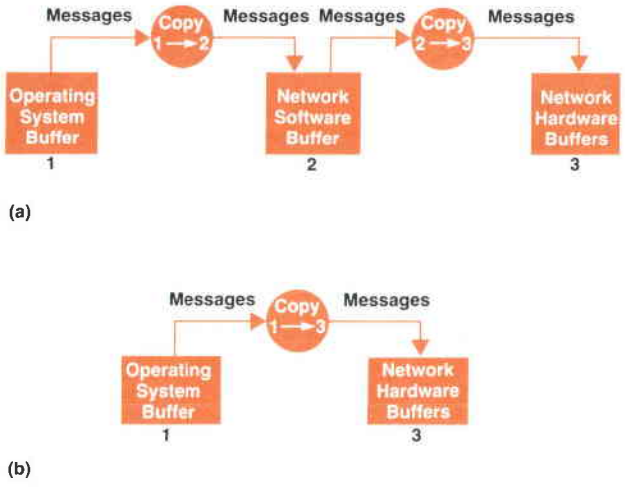


Fig. 1. *Reducing network communication overhead. (a) Typical message copying scheme for network communication. (b) Discless implementation.*

HP-UX discless implementation, data is copied directly between operating system buffers and hardware buffers, thus eliminating one level of copy operation. These different copying schemes are illustrated in Fig. 1.

Performance measurements on a pair of HP 9000 Model 350 client/server machines indicate that our simple discless protocol and the buffering scheme have enabled us to meet the established performance goals for discless machines. To understand the distribution of overhead among the operating system and network functions, the kernel was instrumented, and the processing time spent in the kernel and network functions on the server during read and write operations was measured. To compare the implementation of two distributed systems on the same machine the performance and overhead profile of the Network File System (NFS) functions was also measured under the same conditions. NFS provides transparent access to remote files in a heterogeneous network. The performance results are discussed on page 24.

Overview of the Network Functions

The network functions are designed only for the discless kernel, performing intracluster kernel-to-kernel communications and linking the client cnodes with the disc facilities on the server. Users can still use the general-purpose network facilities such as the ARPA/Berkeley services, HP's NS network services, and NFS to access resources both within and outside a cluster. Since the discless cluster provides a single-file-system view, users have no need to use any of the general-purpose network functions such as ftp, rcp, or NFS to access resources within the same cluster. Intracluster remote process execution can be achieved by using the functions remsh, rlogin, or rt.

The discless network protocol coexists with other general-purpose protocols. The discless messages conform to the IEEE 802.3 link level protocol header format. Fig. 2 shows the relationship between the discless network protocol and the NFS protocol stack. The discless implementation and other general-purpose protocols share the same network hardware and device driver. Sitting above the driver level, the discless network functions are completely independent of other network functions. The network functions use the cnode number and the link level address of the LAN card for source/destination addresses. The mapping of cnode number to the link level address of each cnode's LAN card is kept in the root server's cluster configuration file. If a machine has multiple network I/O cards, only one can be used for discless communications.

In the current implementation, the client and server machines are all HP 9000 Series 300 machines. Therefore, the protocol does not need to translate the data representations from one machine's format to another. The design is extensible to accommodate heterogeneous machines within a cluster.

Discless Message Interface Functions (DM Layer)

The discless message interface functions provide the interface between the HP-UX kernel functions and the HP-UX discless protocol. To send a message, an HP-UX kernel function (e.g., read or write) sends a request to a network function called dm_send. The parameters for dm_send include the destination cnode number, the message buffer, an optional outbound data buffer, an optional inbound data buffer, a set of control flags, and the function to be called when the reply is received back at the client. Fig. 3 shows the activities involved in processing a message at the client cnode and at the server, using the discless request/reply protocol. These activities and the discless protocol are described in the following sections.

Message Buffers. A network message may contain a small message buffer and, optionally, a large data buffer. The message buffer holds the commands and their associated parameters. If the request/reply message includes a large data block (e.g., a file system block), then the data buffer is used. The discless buffer management functions maintain their own pool of message and data buffers.

Before a kernel routine can send a request message it must allocate a message buffer. The discless buffer management functions provide facilities for allocating a buffer chain depending on the size of the request, and for filling the buffers with commands and parameters. Buffer management functions are discussed in detail on page 23.

Inbound/Outbound Data Buffers. When a file block is written to the server, the kernel write function includes the file system buffer as the outbound data buffer in the send call. For a file read the kernel read function will preallocate the file system buffer for receiving the remote file block and include the file buffer in the send call's inbound buffer parameter. This guarantees that reply messages will not be lost or delayed because of buffer shortage problems.

Control Flags. The control flags contain the information that enables the discless protocol functions to determine the protocols for delivering the request messages. For instance, a client may wait (go to sleep) for a reply or continue without waiting, thus enabling the discless protocol to sup-

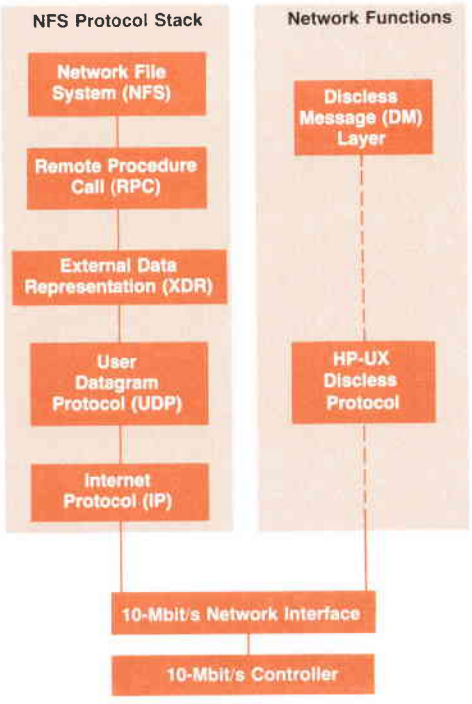


Fig. 2. *NFS protocol stack versus HP-UX network functions.*

port asynchronous or synchronous-mode I/O operations when accessing remote files. The client can also specify if the request is idempotent (repeatable) or nonidempotent. Idempotent messages and discless protocol are discussed in more detail in the next section.

Arrival at the Server. When a request message arrives at the server, the request is either processed as part of a network interrupt service function or by a server process, and then the kernel file system function specified in the command buffer is invoked and the request is executed. After the request is processed, the server calls a reply routine to send back to the client the status and, optionally, the selected file block.

Return to the Client. When the reply message arrives back at the client machine, a network function directly copies the reply messages from the LAN card buffer to the preallocated receive buffers. After the reply message is fully reassembled in the receive buffers, a network function wakes up a sleeping request process (if it had been placed in the wait state), delivers the reply message, and returns to the kernel function that made the send call. For asynchronous operations, the network also releases the request/reply messages automatically. The control flow diagrams in Fig. 4 show the flow of messages between client and server.

Discless Network Protocol

The discless network protocol is based on a simple request/reply model. For each request message, the serving cnode sends back a reply message to the requesting cnode. There is no acknowledgment for receiving the request; instead, the reply is used as the acknowledgment. On an IEEE 802.3 network the maximum packet size is 1514 bytes, and it may take up to 6 packets to transmit a maximal-sized discless message (a 1K-byte message buffer plus an 8K-byte data buffer). Since messages are rarely lost in the local area network, the protocol does not need to acknowledge each individual packet. By using the reply message as the acknowledgment for the multipacket messages, it reduces a significant amount of overhead in protocol handshakes used to prevent message loss during transmission.

As mentioned above, there are two types of messages: idempotent (repeatable) and nonidempotent. These message types determine the sending/receiving request/reply protocol between the client and server cnodes. Fig. 4 shows the relationship between these two types of messages. For idempotent messages the requesting cnode continues to transmit requests until a reply is received from the server. The nonidempotent requests are processed by the server exactly once and the server repeatedly transmits replies until the client acknowledges the reply. This protocol provides excellent performance and helps handle lost messages.

For idempotent messages, when the reply is sent to the client, the server releases both the request and the reply messages. If the reply is lost, the client machines will retransmit the request and repeat the request/reply cycle again. If the same request arrives at the server just after the requested operation is finished but before the reply is returned to the client the request is considered a duplicate

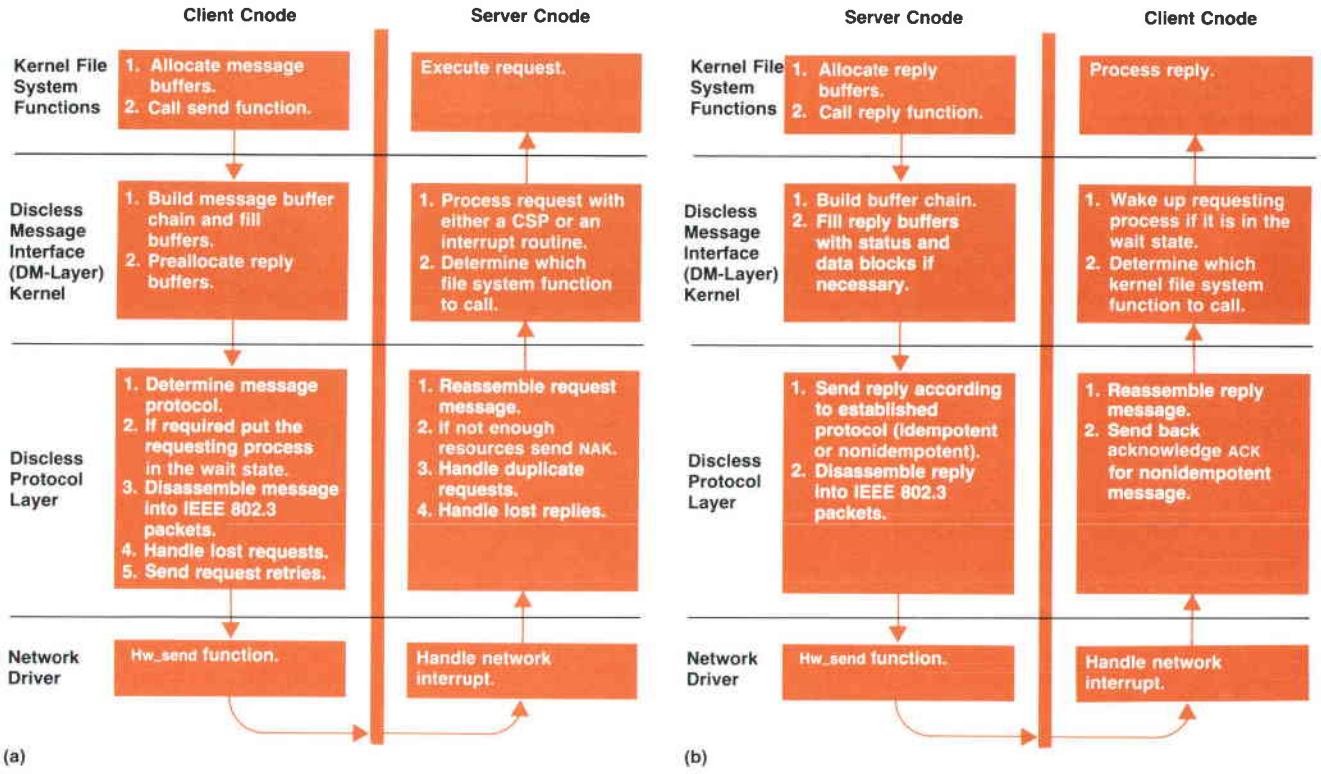


Fig. 3. *Network control flow. (a) Originating a request from the client and receiving a request at the server. (b) Sending a reply from the server and receiving a reply at the client. CSP = cluster server process.*

and is ignored. For the nonidempotent requests the server cannot release the request and reply messages until the client acknowledges that the reply has been received.

Some requests take an indefinitely long time to complete—for example, reading from or writing to a locked file. A request of this type (called a slow request) is indicated by the arrival of duplicate requests before the operation requested is finished. To eliminate the unnecessary retries, the serving cnode sends back a special acknowledgment to the requesting cnode whenever a slow request is detected. Upon receiving such an acknowledgment, the requesting cnode stops retrying. Since the client stops retransmitting slow requests, and the reply could be lost, we use the nonidempotent request protocol to handle the reply messages of slow requests.

To handle lost message problems, the requesting cnodes retransmit the request forever at intervals of 2, 3, 4, 5, ..., 5 seconds until the reply is received. Similarly, the server cnode retransmits the reply messages of nonidempotent requests every half second until the acknowledgment is received. The retries continue until the crash detection function detects that the destination cnode is down and aborts the requests and retries.

Messages can be lost in a receiving cnode for two reasons: the LAN card's I/O buffer has overflowed or there is a shortage of discless networking resources such as networking buffers, data buffers, or protocol table slots for keeping track of messages. Since the resources for the reply messages are preallocated, the lost message problem will not happen to the requesting cnode. To prevent excessive message loss on a heavily loaded server, the serving cnode sends back a special negative acknowledgment message (NAK) to the requesting cnode whenever it fails to allocate any network resource for a new request. The message sending functions on the requesting cnode delay sending new requests or retries to the server cnode when a NAK message is received.

The discless request/reply protocol model provides a reliable message delivery mechanism. However, some discless kernel functions, such as crash detection and distributed clock synchronization, only need quick access to the network without the request/reply protocol. These functions are not concerned with the lost message problem. To support such a requirement, the discless protocol provides a datagram service, which bypasses the request/reply protocol and directly calls the network driver to transmit a datagram over the network.

Discless Networking Buffer Management

A design goal of the discless implementation was to achieve the highest level of distributed intracluster communication possible. Efficient networking buffer management is critical to achieving this goal. Copying network data is an expensive operation and manifests itself in various places. The protocol efforts would suffer if the overall implementation did not address and attempt to minimize data copying operations.

The first problem to overcome occurs when the server receives a write request containing a large data block. In this case a file system buffer is required, but the standard HP-UX file system buffer pool cannot be accessed by discless network functions during an interrupt. To solve this problem, a separate pool of data buffers called fsbuf was implemented for the discless system. This resource is similar to the standard HP-UX file system buffer pool except that it is available to discless network functions during an interrupt. An fsbuf is used only on a serving cnode, and the pointers for the file system and fsbuf are identical. Therefore, when the server starts processing a write request we can simply switch the fsbuf and file system buffer pointers, instead of doing a buffer-to-buffer copy.

Double-buffering and data copying operations need to be minimized for discless buffer management. Towards this end, the LAN device driver has two special, discless specific functions that provide the necessary support. These functions allow the protocol layer to copy data directly between the file system buffers and the LAN card's hardware buffers, eliminating intermediate buffering operations. The first function handles inbound messages resulting from read requests, and the second function handles outbound messages resulting from write requests. For read requests a file system buffer is preallocated before generating the request to handle the reply data. This helps to minimize delays in processing reply messages during an interrupt.

Besides fsbuf, two other data structures used for discless buffer management are mbuf and cbuf (collectively called network buffers). The number of these buffers is set at boot. These resources are the fundamental set of data structures used for all discless messages. The data structure mbuf was originally developed by the University of California at Berkeley as a general-purpose buffer management mechanism. The complete Berkeley design was deemed unnecessary for the more limited discless situation. The discless mbuf/cbuf is superficially similar in many respects to the Berkeley design but is implemented and used in a different manner. The networking buffers encapsulate the request/reply message, which contains the commands and their associated parameters. The mbuf is relatively small (128 bytes each)

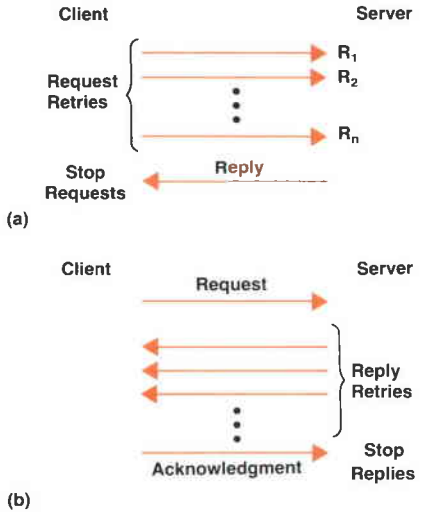


Fig. 4. *The relationship between idempotent and nonidempotent messages using the request/reply protocol. (a) Idempotent. (b) Nonidempotent.*

and sometimes cannot hold sufficient information. In these situations, the cbuf (1024 bytes each) is used and mbuf will contain a pointer to the allocated cbuf. For requests that involve data blocks for reading or writing, the mbuf/cbuf combination will contain an optional pointer to the fsbuf for the data block.

Discless Cluster Server Processes (CSP)

There are many existing networking models that deal with interprocess communication in distributed environments. A common paradigm for such applications is the server/client relationship. In this model a daemon process normally listens at some well-known address for requests. Upon receiving a request for service, a daemon process forks an image of itself to handle the request. Meanwhile, the original parent server process resumes listening for additional connections. Forking the new child process and context switching from the user to the system environments are expensive operations, and would be unacceptable for handling requests in a discless environment.

The discless implementation is also based on the server/client model, but a different method of handling requests was developed. The serving cnode maintains a small dynamic pool of kernel processes called general cluster server processes (GCSPs). GCSPs avoid the overhead of forking and still satisfy requests from the client cnodes in the cluster. These processes can be quickly and efficiently created, destroyed, and context switched. When a request arrives at the server a GCSP is allocated from the pool to handle the request. When a GCSP is run it is locked into memory and cannot be swapped out. GCSPs are created from user space when the server is initially configured as the cluster server. The number of GCSPs is also determined at this time.

Special Types of CSPs. In general, the discless cnodes do not require GCSPs. However, certain operations must be performed by the discless cnode to maintain its membership in the cluster, such as synchronizing the mount table or converting to synchronous I/O on a file. The limited capability CSP (LCSP) solves this problem. This solitary LCSP is created when a cnode first joins a cluster.

Most client requests are handled by the GCSPs. However, in a few instances the server needs to run user-level code to service a request. For example, the server needs to read the file-system-resident cluster configuration file to determine whether to grant a client's request to join the cluster. In such cases the server dynamically creates a user-level cluster server process (UCSP), services the request, and then exits. Currently, only a single UCSP has been implemented, and it is is used to read the cluster configuration file mentioned above.

Slow or Indefinite Operations. Certain operations may take an indefinite amount of time to complete. For example, a read from a FIFO file could wait indefinitely. During this waiting period, a GCSP is tied up. If all the GCSPs were used in such a manner, there would be none available to service other requests and the system would come to a halt. One solution would be to dedicate certain GCSPs to slow operations while others would be available for the fast operations. In practice, it is difficult or impossible to determine in advance whether an operation will be fast or slow. For example, it is impossible to distinguish between a write to a FIFO file and one to a disc without examining the message and various file system data structures that may be implicitly referenced. Even if it can be determined that the access is to a fast device like a disc, the operation could still take a long time because of system calls like lockf() and other system interactions. For the discless implementation, a more reliable mechanism was chosen. Whenever a GCSP is invoked it sets a time-out, the duration of which is dependent on the number of free GCSPs. If the GCSP completes its assigned task before the time-out period, it clears it. Otherwise, a replacement GCSP is created and the slow GCSP is set to terminate itself upon completion. This ensures that an adequate pool of GCSPs is maintained.

Performance Measurement Results

The environment chosen for evaluating the discless implementation for this paper was a completely isolated two-cnode discless cluster. Both the root server and the discless cnode were identically equipped, 16M-byte HP 9000 Model 350 Computers. The server used an HP 7958A Disc Drive for both the file system and the swap areas. After the discless cnode was remotely booted over the LAN, all unnecessary processes were killed on both machines. The root server had four GCSPs running and standard default-configured kernels were used. Both systems were rebooted after every benchmark to ensure that no data was cached from a previous run that might affect the current benchmark.

Sequential Write Statistics (10M bytes transferred)

Block Size (K bytes)	Throughput* (K bytes/s)	% of Stand-Alone	Protocol Path
16	423.57	100%	Stand-Alone System
8	423.14	100%	
4	420.76	100%	
1	401.82	100%	
16	365.13	86.20%	Discless Protocol Path
8	354.80	83.80%	
4	372.84	88.60%	
1	359.47	89.50%	
16	77.51	18.30%	NFS Protocol over an NFS Mount Point
8	78.06	18.45%	
4	45.30	10.80%	
1	43.63	10.90%	

*All throughput numbers are the result of averaging 10 data transfers.

Sequential Read Statistics (100M bytes transferred)

Block Size (K bytes)	Throughput (K bytes/s)	% of Stand-Alone	Protocol Path
16	428.33	100%	Stand-Alone System
8	426.10	100%	
4	428.30	100%	
1	428.52	100%	
16	390.52	91.17%	Discless Protocol Path
8	388.67	91.22%	
4	388.71	90.76%	
1	387.47	90.42%	
16	315.43	73.61%	NFS + Discless Protocol over an NFS Mount Point
8	312.81	73.41%	
4	309.06	72.16%	
1	299.46	69.88%	

Fig. 5. *HP-UX Series 300 release 6.2 discless versus NFS throughput statistics. (a) Throughput for sequential writes. (b) Throughput for sequential reads.*

Two benchmarks were run from the discless cnode. The first benchmark performs repetitive sequential reads, using various block sizes, of a 10M-byte file located on the root server's file system. The file is read ten times, resulting in a total of 100M bytes transferred. The second benchmark performs sequential writes, using various block sizes, to a file on the server's disc. The writes continue until a 10M-byte file has been written. For both benchmarks the user process allocates an incore buffer of the specified block size. This buffer is repetitively read into or written from. No intermediate files are created by the benchmarks.

For comparison, the benchmarks were run in completely discless and discless-plus-NFS environments. The NFS environment was established by mounting from the discless client the root server's test and data directories across an NFS mount point. By executing the benchmarks with the source and target files crossing the NFS mount point, NFS protocols are used and a clean separation from the otherwise discless environment is achieved. The benchmark executables were small and there was enough RAM on both systems to avoid swapping. For this scenario, four NFS block I/O daemons (biods) were run on the client and four network file server daemons (nfsd) were run on the root server.

In this way a situation was established for comparing two different distributed networking implementations. The HP-UX discless implementation uses special-purpose networking protocols, buffer management functions, and CSPs, whereas NFS is designed for a heterogenous multivendor environment and uses a different protocol stack and general-purpose HP networking and buffer management functions.

Throughput Results

Fig. 5* represents the throughput of the read and write benchmarks for the discless and NFS protocol paths. They are compared to stand-alone results of running the same benchmarks on just the root server (i.e., no networking involved). The throughput for the discless system with this environment is encouraging. The client was able to read at a rate of approximately 389K bytes/s or 91% of the stand-alone rate for this disc drive. The write statistics are also encouraging, achieving a rate of approximately 363K bytes/s or 87% of the stand-alone numbers. The root server's file system buffer cache for this experiment was only 2.4M bytes, so few if any cache hits occurred.

The more general-purpose NFS networking path was able to achieve rates of approximately 309K bytes/s on reads, or 72% of the stand-alone rate. The NFS write statistics are less encouraging, averaging only 61K bytes/s or 15% of stand-alone. This can be directly attributed to the lack of delayed asynchronous writes in the standard implementation of NFS. Comparisons between discless and NFS writes are somewhat meaningless, since they represent different design methodologies.

Given that the benchmarks were tightly controlled, repeatable, and run on identical hardware, it can be safely stated that the overall throughputs for this experiment seem to favor the discless implementation. Performance measurements are very application dependent. These benchmarks do not address large-cluster performance and the resultant clusterwide throughputs. The data should only be considered within the context of the established experiment. Different hardware, disc drives, and numbers of discless cnodes all play a role in evaluating clusterwide performance. It is beyond the scope of this paper to address these issues.

*These results are for the latest HP-UX Series 300 release, 6.2. The NFS throughput performance in release 6.2 is much better than release 6.0. There are no significant differences in the discless read or write throughput performances between the two releases.

Kernel Measurement Method

To improve our understanding of the throughput data for the benchmarks, the server's kernel was instrumented and statistics were gathered and examined in great detail.

Functional Area Breakdown of Time Spent in the Server's Kernel for an NFS Read

Kernel Functional Area	Total Elapsed Time	Total Clock Ticks	Percentage Non-Idle Time in Kernel	95% Confidence Interval (L ≤ P ≤ U)
bcopy:	1m04.40s	3220	30.21%	29.3 – 31.1%
General Kernel:	0m38.06s	1903	17.85%	17.1 – 18.6%
Disc I/O — DMA:	0m23.32s	1166	10.94%	10.3 – 11.5%
LAN Driver:	0m30.20s	1510	14.17%	13.5 – 14.8%
Network Buffer Management:	0m12.52s	626	5.87%	5.4 – 6.3%
File System:	0m10.44s	522	4.90%	4.9 – 5.3%
NFS protocols: (UDP, XDR, RPC, NFS, and IP.)	0.32.62s	1631	15.29%	14.6 – 16.0%
Discless Overhead:	0m01.02s	31	0.29%	.2 – .4%
Totals:	3m33.20s	10,660	100.00%	
*Kernel Idle:	7m08.08s	21,400		

(a)

Functional Area Breakdown of Time Spent in the Server's Kernel for a Discless Read

Kernel Functional Area	Total Elapsed Time	Total Clock Ticks	Percentage Non-Idle Time in Kernel	95% Confidence Interval (L ≤ P ≤ U)
bcopy:	1m13.70s	3685	42.65%	41.6 – 43.7%
General Kernel:	0m33.44s	1672	19.35%	18.5 – 20.2%
Disc I/O — DMA:	0m22.40s	1170	13.54%	12.8 – 14.3%
LAN Driver:	0m21.02s	1051	12.16%	11.5 – 12.9%
File System:	0m09.30s	465	5.38%	4.9 – 5.9%
Discless Protocol:	0m08.56s	428	4.95%	4.5 – 5.4%
Buffer Mgmnt.:	0m02.40s	120	1.39%	1.1 – 1.6%
GCSP:	0m01.64s	82	0.95%	0.7 – 1.2%
DM Layer:	0m00.76s	38	0.44%	0.3 – 0.6%
Totals:	2m53.22s	8641	100.00%	
*Kernel Idle:	2m12.06s	6603		

(b)

*This is time spent in the kernel idle loop waiting for such things as disc I/O.

Fig. 6. *Kernel server measurements by functional area. (a) NFS read of 100M bytes using 8K-byte blocks. (b) Discless read of 100M bytes using 8K-byte blocks.*

At every clock tick (every 20 ms on an HP 9000 Series 300), the system sampled the processor's program counter (PC), the active process identifier (PID), the type of process (CSP versus regular), and other system parameters. The samples were written to an incore kernel buffer, extracted from the buffer by a user-level process, and then written to disc. The data was postprocessed by associating the sampled program counters with specific kernel procedures. The incremental cost imposed on the system by the monitoring actions was minimal, approximately one fifth of one percent of all the time spent in the kernel. The data was postprocessed on another machine after the entire benchmark was completed. Each benchmark was repeated ten times, and the results were completely consistent. The postprocessed data was segmented into separate kernel functional areas: byte copy (bcopy), disc I/O, LAN device driver, file system, networking protocol stack(s), and buffer management.

As a result of using a 20-ms sample rate, some level of confidence with the derived numbers was required. A 95% statistical confidence level was chosen and upper and lower statistical confidence intervals were calculated.[1] The confidence intervals were derived on the basis of the number of observed clock ticks in each kernel functional area as a function of the total number of samples taken. All numbers were rounded up to three decimal places. The 95% confidence intervals were derived as follows:

P = (functional_area_ticks) / (total_sample_ticks_N)
Lower Bound L = P − 1.96 * sqrt ((P * (1 − P))/N)
Upper Bound U = P + 1.96 * sqrt ((P * (1 − P))/N)

Discless Server Kernel Functional Area Profiles

Fig. 6 shows the profiles for the functional areas measured in the server's kernel. The profiles show a read benchmark (executed from the client) using both the discless and NFS protocol paths to transfer 100M bytes of data in 8K-byte blocks.

The NFS write strategy does not make allowances for delayed asynchronous writes, while the HP discless implementation does. Since the HP discless and NFS client write strategies are so different, it seemed unnecessary to present data that could not be realistically compared.

An enlightening set of observations can be extracted from the server's kernel profiles compared with the client throughput results. Of all the time spent in the server's kernel routines, byte-copying data (bcopy) is by far the largest consumer of CPU resources. This is predominately the CPU cost of copying data to or from the LAN card's hardware buffers and the target networking buffers.

The next interesting set of numbers shows the amount of kernel time spent in performing discless protocol and buffer management functions: 4.95% and 1.39% respectively, with a combined elapsed time of 10.96 seconds. Comparing these numbers with the more general-purpose NFS path, we get 15.29% for the NFS protocol stack and 5.87% for buffer management with a combined elapsed time of 45.54 seconds. These comparisons must also be weighted by acknowledging that the entire 100M-byte read took 247 seconds for the discless system and 441 seconds for the NFS path.

The combined total of the discless specific components—protocol, buffer management, CSPs, and DM layer—accounts for 10.73% of the server's total kernel time, or an elapsed time of 13.36 seconds. This is only 41% of the time spent in just the NFS protocol stack to accomplish an equivalent transfer of data.

Conclusions

High-level algorithms play a key role in the performance of distributed systems. Special-purpose networking protocols, server processes, and network buffer management routines must all play together in the design of such a system. Good performance requires not only a system view of the goals, but also an efficient implementation of the design.

Special-purpose designs like the HP-UX discless implementation have their advantages and disadvantages. The advantages are considerable in the context of a closely knit work group where a single-system view, high-speed intracluster communication, and transparent sharing of files and access of data are extremely important. As long as the special-purpose design allows a peaceful coexistence and complete interconnectivity with the outside world via standard and evolving networking services (ARPA/Berkeley, NFS, etc.), the user is provided with a powerful combination of capabilities. It is only in the more limited context of wide-area connectivity for discless cnodes that the special-purpose design shows disadvantages. Specifically, the inability to operate across a gateway limits the range of interconnectivity possible. It is this type of situation that places undesirable limits on the design of discless systems and tends to hinder their performance.

Acknowledgments

Special acknowledgments are extended to the following individuals for their considerable contributions: Joel Tesler for his efforts in CSPs, DM, and protocol slow requests, Ching-Fa Hwang for slow request and initial project management, Bob Lenk for CSPs, Bruce Bigler for DM, Joe Cowan for project management, Doug Baskins for kernel instrumentation, and Ray Cheng for early protocol prototype.

References

1. P.Z. Peebles, Jr., *Probability, Random Variables and Random Signal Principles*, McGraw-Hill, 1980.

Crash Detection and Recovery in a Discless HP-UX System

by Annette Randel

HP-UX DISCLESS CLUSTERS depend on close, kernel-to-kernel communication across a local area network to maintain high performance and transparent file system access. This kernel-to-kernel communication relies on state information maintained on all nodes of the cluster. When a cnode is removed from the cluster because of an expected or an unexpected failure, this kernel state information must be cleaned up to reflect the new cluster configuration. Because this state information is at a very low level in the HP-UX implementation, failure to clean up state information after a crash can cause other cnodes in the cluster to hang indefinitely, waiting for the crashed cnode to complete a transaction. This is unacceptable, and therefore, prompt and reliable detection and cleanup of crashed cnodes are required at the kernel level.

For the purpose of this article, a crash or failure can be defined as the removal of a cnode from an HP-UX cluster. Two types of crashes or failures can occur on a cnode in a cluster: an expected failure or an unexpected failure. An unexpected failure may be caused by a hardware failure, a loss of power, or a software failure. When an unexpected failure occurs, the failing cnode may be unable to notify other members of the cluster of its demise. An expected failure occurs when a system is intentionally and properly shut down by the operator. During an expected failure, the cnode being shut down should notify all other cnodes that it is leaving the cluster. Reliable detection of both expected and unexpected failures provides the HP-UX system with resiliency in the face of an unexpected failure as well as dynamic reconfiguration around an expected failure.

There are four general requirements for crash detection and recovery in HP-UX clusters. First, it is required that the operating system maintain HP-UX semantics in the face of a failure. This requires that file system consistency and reliability be maintained, even though a cnode is removed from the cluster. Second, it is required that a consistent view of the cluster membership be maintained from all cnodes. A third requirement is that the detection of a crashed cnode and the recovery of that cnode's resources be transparent to users of other cnodes in the cluster. This means that no user action is required and the performance impact on the user is minimized. Finally, it is required that the rest of the cluster be resilient in the face of a client cnode failure. This means that the failure does not cause a chain reaction of failures in other cnodes and that no data loss occurs on nonfailing cnodes. The exception to this requirement is the root server cnode. Because client cnodes cannot recover from the failure of a root server cnode, a root server failure will cause all nodes in the cluster to fail.

Crash Detection

Because of the state dependent nature of communication in HP-UX clusters, cnode failures must be detected quickly to prevent long system delays on functioning cnodes needing resources tied up by the failed cnode. In addition, failures must be recognized before allowing the failed cnode to rejoin the cluster. If the failed cnode were allowed to rejoin the cluster before crash recovery had taken place, valid state information could be improperly cleaned up, or invalid state information could be acted upon by the newly clustered node (e.g., it could respond to the retry of a request sent before the failure occurred). Neither is acceptable.

The crash detection mechanism must also ensure a consistent view of cluster membership for all cnodes in the cluster. All cnodes maintain their own cluster status table, and, while most cluster communications occur between the root server cnode and client cnodes, there are some kernel messages that are passed from one client cnode to all other cnodes. State information for these messages must be cleaned up on the client cnodes as well as on the root cnode after a failure occurs.

One final requirement of the crash detection mechanism is that it must never incorrectly declare a nonfailed cnode to have crashed, even in the face of a LAN failure. This requirement conflicts somewhat with the requirement to detect unexpected crashes quickly, and multiple detection mechanisms are employed to fulfill all the requirements.

Detection Mechanisms. Five different mechanisms are employed to detect both unexpected and expected cnode failures reliably in a minimal period of time:

- The failing cnode broadcasts a datagram indicating its expected failure.
- The server and each client cnode exchange status messages.
- When the server detects a cnode failure, it informs the cluster by broadcasting a reliable message.
- If a failed cnode attempts to rejoin the cluster before its failure has been detected, the clustering operation will be postponed until the crash recovery has been completed.
- LAN cable failures are detected, and the crash detection mechanism is disabled until the LAN is correctly configured. This prevents cnodes from being incorrectly declared failed because of LAN cable failures.

When all of these mechanisms are used together, they provide reliable detection of both expected and unexpected failures within a reasonable period of time. A more detailed description of each mechanism and their interactions is presented below.

Broadcast Failure Datagrams. When an expected failure or an orderly shutdown occurs, the failing cnode cluster-casts (broadcasts to the entire cluster) a failure datagram. This tells the other cnodes that it is about to shut down and that no further communication from this cnode should be expected. Since this message is a datagram, we cannot depend on cnodes receiving this message. The failure datagram is an attempt to get the word about the failed cnode out quickly, rather than a reliable information mechanism.

Server/Client Status Messages. To detect both expected and unexpected crashes reliably, the server and client exchange status messages. This message exchange occurs only when there has been no communication with a cnode for a given period of time. A state transition model (see Fig. 1) is used to track communication with other cnodes and to determine when a given cnode can no longer be contacted. This state transition model is based on an internal data structure, the cluster status table, which maintains a given cnode's view of the status of the entire cluster. A cnode entry in the cluster status table may be in one of the following states:

- ACTIVE: A message has recently been received from this cnode.
- ALIVE: No messages have been received from this cnode in several seconds.
- RETRY: A message has not been received from this cnode in several seconds, and we are currently requesting status information from this cnode.
- FAILED: This cnode has failed. If executing on the root server, all cnodes have not yet been informed of the failure.
- CLEANUP: This cnode has failed, and its resources are being recovered.
- INACTIVE: This cnode has never joined this cluster, or it has failed and recovery is complete.

Every time a message of any type is received from an ACTIVE, ALIVE, or RETRYing cnode, that cnode's status is updated to ACTIVE in the cluster status table. Every few seconds a kernel-level status checking process is executed which downgrades the status of each cnode according to its current state (see Fig. 2). On the root server cnode, this process downgrades the status of all cnodes. On discless nodes, however, this process only affects the state of the root server cnode. States are affected by this process as follows:

- ACTIVE: Downgraded to ALIVE.
- ALIVE: Downgraded to RETRY.
- RETRY, FAILED, CLEANUP, and INACTIVE: No change.

A status of ALIVE or RETRY may be upgraded at any time to a status of ACTIVE by the receipt of a message from that cnode.

When the status checking process downgrades a cnode's status from ALIVE to RETRY, it also sends a status request message to the cnode whose status is being downgraded. The status request messages are based on the datagram service. Datagrams are fast, but unreliable, so they must be retried manually. To do this, another process, the retry status request process is executed every second (See Fig. 3). This process determines, via a global variable if any cnodes are currently in the RETRY state. If there are none, this process simply exits. If, however, one or more cnodes are in the RETRY state, then the retry status request process searches the cluster status table for the RETRY cnode entries. When a cnode in the RETRY state is found, a retry counter is incremented for that cnode, and another message requesting a status update reply is sent. When a cnode whose retry counter has exceeded its maximum is found, that cnode is considered to have failed. However, before declaring the cnode as failed, the LAN failure detection mechanism, described below, will be invoked.

When a cnode receives a status request message, a status

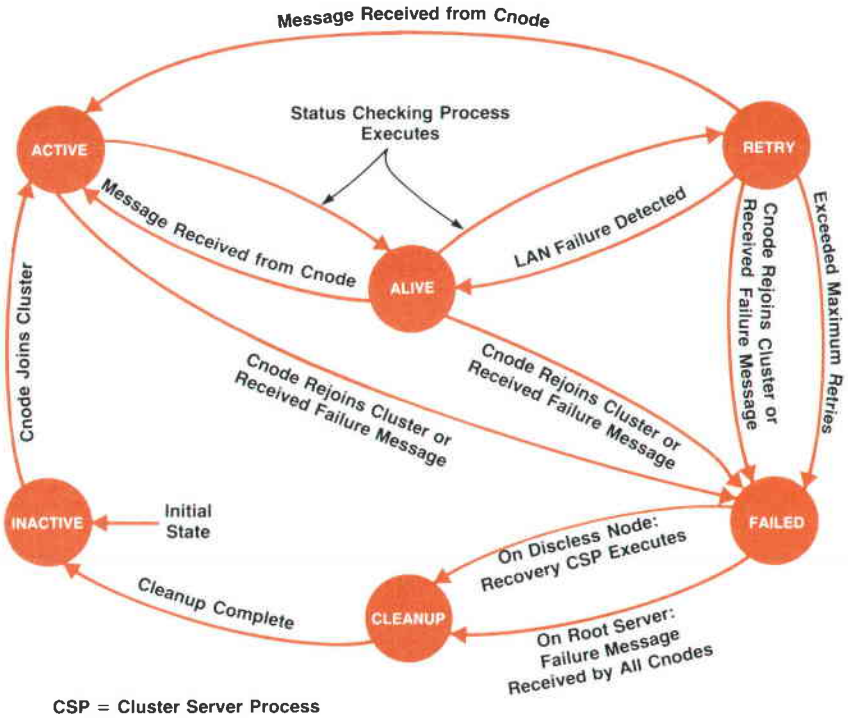


Fig. 1. *States of crash detection and recovery.*

reply message, also based on the datagram service, is immediately sent in reply. This message is also unreliable (i.e., not retried by the networking service), and the crash detection mechanism relies on retries from the requester to cover lost messages.

Server Broadcast Failure Mechanism. The broadcast failure datagram mechanism is unreliable, and because the server/client status message exchange does not inform the entire cluster of a failure, another mechanism is needed to ensure that the entire cluster is aware of a cnode failure. The mechanism used for HP-UX is a reliable failure message which is clustercast (broadcast to the entire cluster) by the root server.

The root server broadcasts the failure message from the crash recovery mechanism. In the recovery process on the root server cnode, if a cnode has a status of FAILED, then the root server broadcasts a message to all other cnodes informing them of the failure. The recovery mechanism uses a single process to clean up the resources of all failed cnodes. This process is serial, and because multiple cnode failures may occur simultaneously or close together, the recovery process cannot sleep waiting for the cnodes to reply. If it were to do so and if one of the cnodes that had not yet replied were to fail, then the root server would be deadlocked, because the recovery process would be waiting for a failed cnode that would never reply and whose resources would never be recovered. The crash recovery mechanism is discussed in more detail on page 30.

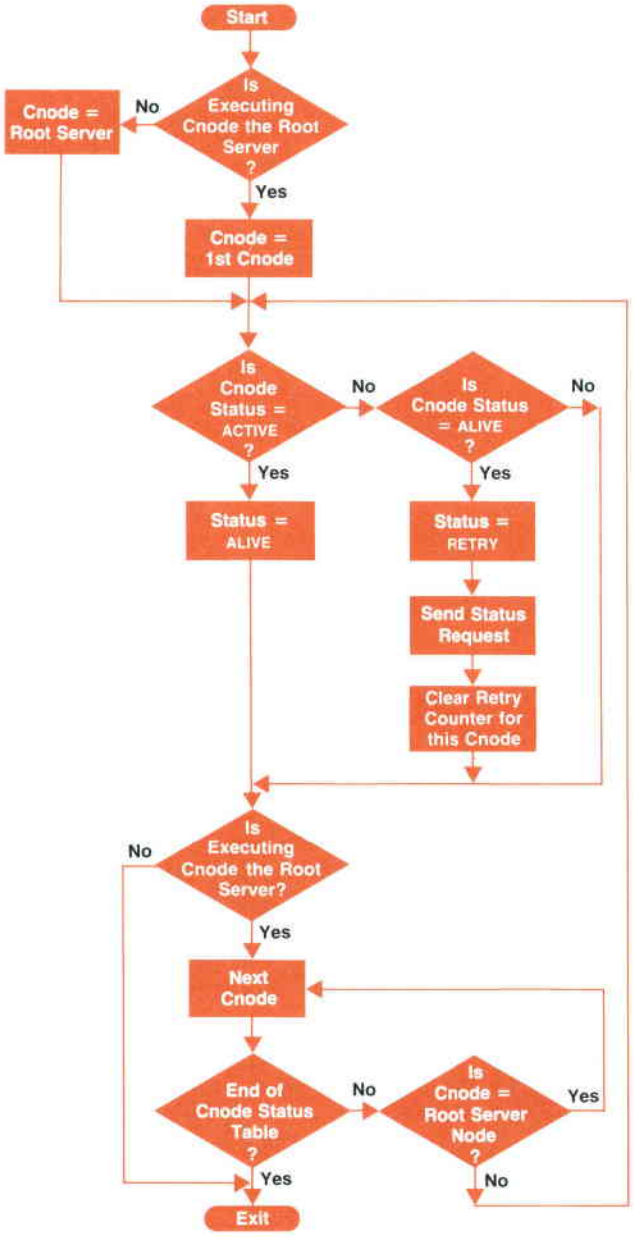


Fig. 2. *Crash detection: status checking process.*

Failure Detection at Boot. As mentioned, a failure must be detected and recovered before the failed cnode can be allowed to cluster (rejoin the discless cluster). Since all cluster requests are directed to the root server, the root server can block the cluster request until all recovery is complete. When a cluster request is received from a cnode whose status in the cluster status table is currently ACTIVE, ALIVE, or RETRY, the server process handling the cluster request sets the status of the requesting cnode to FAILED and invokes the recovery process.

Race conditions between the clustering process and the recovery process are prevented by not allowing more than one cnode to join the cluster at one time and by not allowing the joining process to continue until the recovery process has been completed.

LAN Failure Detection. Because detection of unexpected cnode failures is based on the exchange of status messages, an undetected LAN failure could be misinterpreted as a cnode failure if messages could no longer be received from a given cnode.

If a LAN failure were misinterpreted as a cnode failure, it would cause all cnodes on the failed LAN to panic (experience a system crash) because of loss of contact with the root server. The root server must detect the failure of a client cnode to ensure that resources held by the failed cnode are released, but it is less obvious why a client cnode must detect the loss of contact with the root server, especially when the only result is for the client cnode to panic. The client cnode must detect loss of contact with the root server because it is possible that, because of a hardware failure of some sort, the client cnode has become deaf (incapable of receiving messages) to the network. If a deaf cnode were allowed to remain in a cluster, it would continue to send requests to the root server, which would continue to mark the cnode as ACTIVE, even though the cnode could not participate in the cluster. This means that the deaf cnode could tie up cluster resources indefinitely, hanging the cluster. Therefore, client cnodes must panic on loss of contact with the root server to ensure that they are not tying up cluster resources.

A LAN failure is detected by running the LAN card hardware diagnostics from a LAN card driver level. These diagnostics are invoked by the retry status request process previously described, just before setting a cnode's status to FAILED. If the LAN card diagnostics indicate a LAN failure, a warning message will be displayed on the system console and the cnode's status will be upgraded to ALIVE. The process of downgrading all cnodes' status to RETRY,

running the LAN diagnostics, and upgrading all cnodes to ALIVE will be repeated until the LAN failure is repaired. All file system access from discless nodes will hang until the LAN is corrected, and it is possible that the root server cnode will hang waiting for a resource held by one of the isolated cnodes. Once the LAN failure is corrected, the system will continue normally.

The LAN card hardware diagnostics cannot detect a LAN failure that occurs on the other side of a LAN bridge. If a cluster is spread across such a bridge, a LAN failure (such as a break in the cable) on one side of the bridge will cause all discless cnodes on the other side of the bridge to lose contact with the root server and panic.

Crash Recovery

Once a cnode failure has been detected, crash recovery is invoked. Recovery from the crash of a cnode requires that certain cluster resources being used by that cnode be released and cleaned up so they can be used by other nodes in the cluster. There are four basic resources that need to be cleaned up in the discless HP-UX system:

- File system data structures (inode)
- Swap space
- Process ID blocks
- Discless networking data structures.

In the current release, the root server maintains most of the cluster's resources, so most of the recovery function occurs on the root server. Discless networking resources are necessary on all cnodes, however, so crash recovery is invoked clusterwide with the root server performing most of the work.

A separate recovery function is called for each basic resource to be cleaned up after the failure of a cnode. The general tasks performed by each function are described below.

File System Recovery. When a failure occurs, the crashed cnode may have file system resources locked up, or those resources may be in an inconsistent state. Each currently referenced file (including named FIFO files, directories, and regular files) in memory is represented by a data structure called an inode. There are two situations where file system cleanup must be performed after a cnode fails, both of which must be performed on the inode:

- The file is locked by the failed cnode, either via a kernel inode lock or by a lockf(2) or fcntl(2) call. An inode lock is indicated by two fields in the inode, one that indicates that the inode is locked and another that indicates what cnode is responsible for the lock. If the file is locked by the failed cnode, then the recovery routine unlocks it. A lock created by lockf(2) or fcntl(2) is indicated by a lock list in the inode structure. If the lock list indicates that the file is locked by the failed cnode, then the recovery routine frees the lock.
- A cnode map field in the inode indicates that the file:
  - is being referenced by the failed cnode
  - is opened by the failed cnode
  - is opened for write by the failed cnode
  - is a FIFO file opened for read by the failed cnode
  - is a FIFO file being executed by the failed cnode.

A cnode map field in an inode is a table that maintains a count for each cnode in the cluster. Cnode maps only

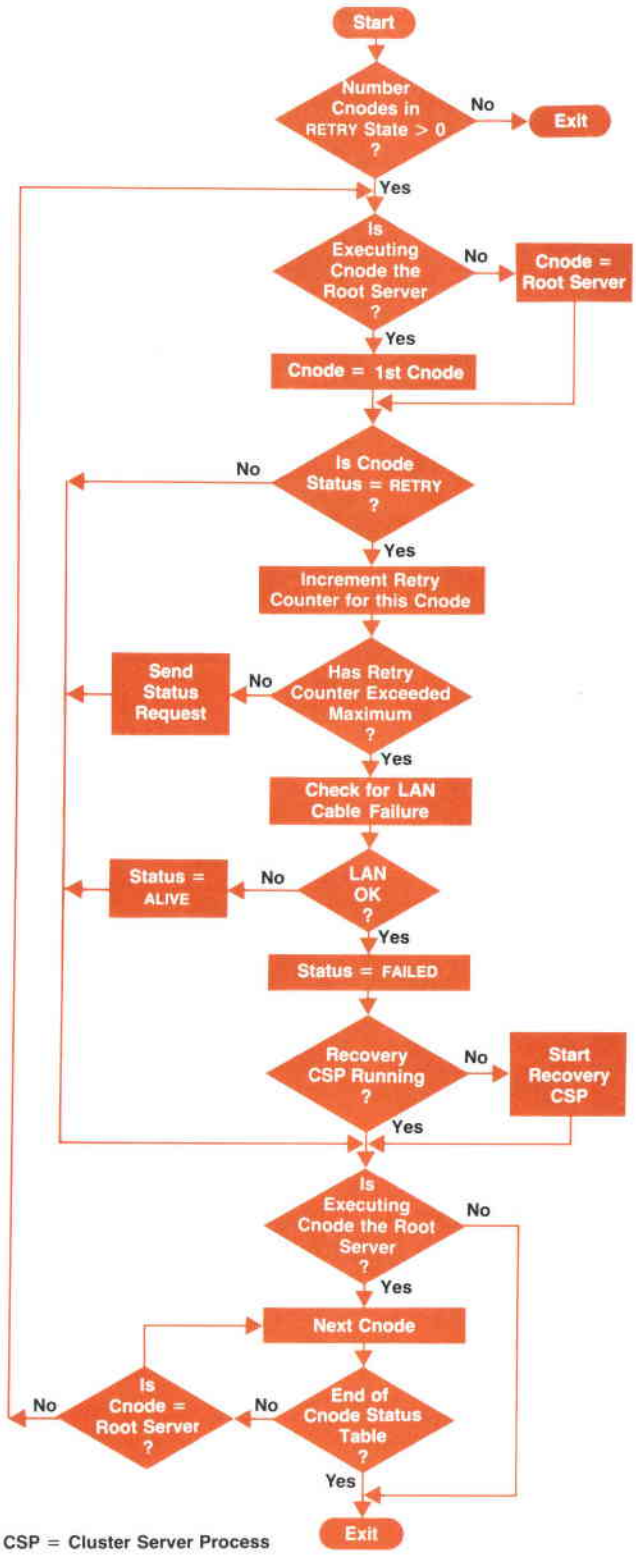


Fig. 3. *Crash detection: retry status request process.*

appear in inodes on the root server. Five different types of cnode maps may be present in an inode, one for each of the cases listed above. The recovery function checks each applicable cnode map (there is no point in checking the FIFO read count if the file is not a FIFO file) and, if the cnode map entry for the failed cnode is nonzero, then the appropriate action (e.g., closing the FIFO file or releasing the text segment) is taken, and the cnode map entry is cleared.

The file system recovery function looks through the memory-resident inode table and cleans up each inode that was being used or referenced by the failed cnode.

Swap Space Recovery. The swap space recovery function looks through the table of allocated swap space on the root server cnode and releases any swap space that was allocated to the failed cnode.

Process ID Recovery. In the discless HP-UX system, the root server cnode acts as a process ID (PID) allocator to guarantee unique PIDs throughout the cluster. The process ID recovery function goes through the PID allocation table on the root server cnode and marks any PIDs allocated to the failed site as available for use.

Discless Networking Recovery. Three basic discless networking resources must be cleaned up when a cnode crashes: cluster server processes (CSPs), networking state information on outstanding requests to other cnodes, and outstanding requests from other cnodes.

CSPs acting on behalf of the failed cnode are killed by a routine that scans the table of active CSPs for those with a cnode ID field matching the failed cnode. All such CSPs are sent a signal indicating that they should abort the current process. Since it may take some time for all the CSPs to abort, this routine is reinvoked until it does not find any CSPs acting on behalf of the failed cnode. The CSP cleanup routine also removes requests for CSP service by the failed cnode from the CSP service queues.

Networking resources for requests to the failed cnode and for remote requests being serviced on behalf of the failed cnode must be recovered. The list of outstanding networking requests is scanned, and all requests destined for the failed cnode are marked undeliverable. Retries on these requests are stopped, associated resources are freed, and a local reply is generated. Requests being serviced on behalf of the failed cnode are cleared, and replies to these requests are stopped and all associated resources are freed.

The Recovery Mechanism

When a failure is detected, a cluster server process (CSP) is invoked in the kernel to execute the recovery functions. This process will be referred to as the recovery CSP. When more than one failure occurs simultaneously, or when a failure occurs while the recovery CSP is still cleaning up a previous failure, the same CSP is used to clean up the resources of all the failed cnodes. The cleanup of each failed cnode's resources is done serially. This means that we cannot allow any recovery function to sleep waiting for a resource that may be held by another cnode that may have failed, or we risk a possible deadlock situation.

To prevent this potential deadlock situation, each recovery function can return an error code that indicates that it was blocked from completing cleanup of a cnode because

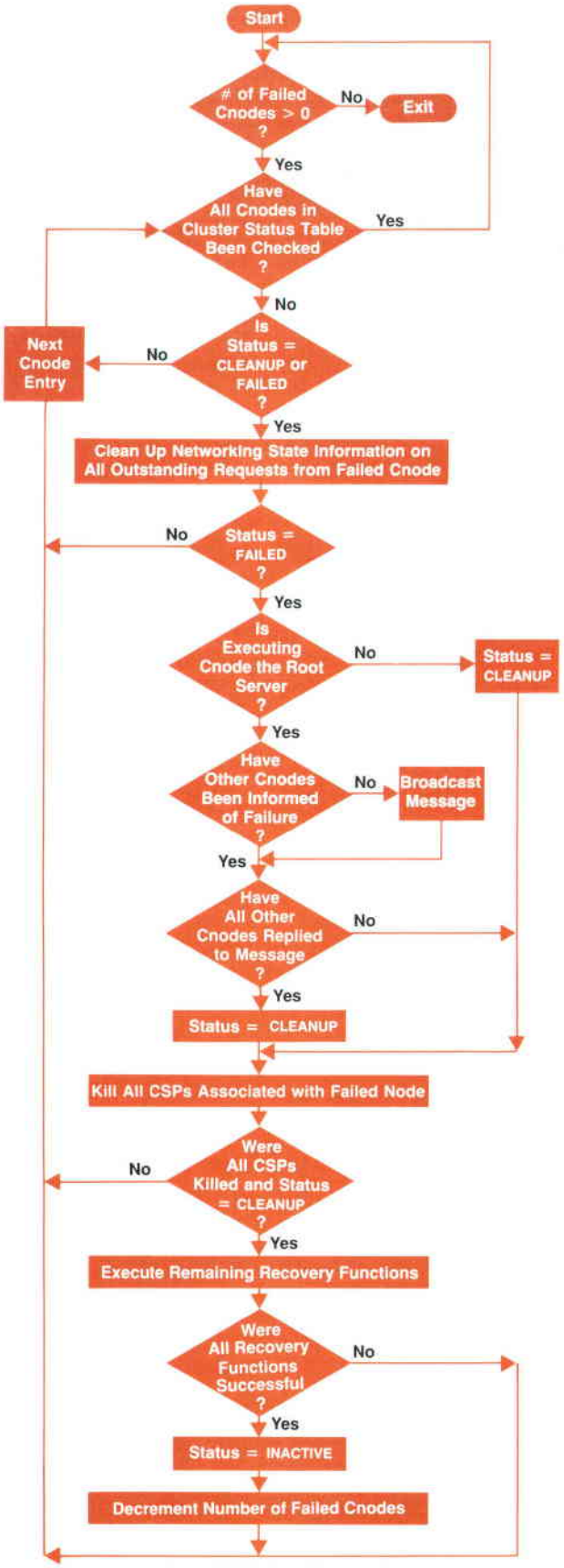


Fig. 4. *General crash recovery mechanism (recovery CSP).*

of a locked resource. When such an error code is returned, the failed cnode is not marked as cleaned up and the recovery functions will be rescheduled for that cnode after the recovery CSP has had an opportunity to execute the recovery functions on other crashed nodes in the cluster.

In addition, to prevent another deadlock situation, we must always guarantee that a CSP will be available to execute the recovery function. This is done by allowing the recovery function to be executed by a special CSP called the limited CSP. This CSP is limited because it only executes processes that are necessary to maintain membership in the cluster. None of the processes that are allowed to execute on the limited CSP should cause the limited CSP to sleep indefinitely waiting on a failed cnode. Therefore, we can always assume it will eventually become available for use in crash recovery. The recovery functions can also be executed on a general CSP if one is available.

The mechanism for crash recovery (see Fig. 4) is closely tied to the crash detection mechanism. Both use the state transition model (see Fig. 1) based on the cluster status table. To execute the recovery mechanism, a CSP is invoked by the crash detection function unless one is already running. This CSP checks a global variable, set up by the crash detection function, to see if there are any failed cnodes. If there are failed cnodes (there is always at least one on the first pass), then the process looks through the cluster status table searching for cnodes with a status of FAILED or CLEANUP (for the first pass for a given cnode, its status is always FAILED). When such a cnode is found, all outstanding kernel networking requests from the failed cnode are cleaned up. This must be done before the failed cnode's CSPs can be successfully aborted. The failed cnode's CSPs must be aborted before any other cleanup can be done to ensure that these CSPs do not make changes after the resource recovery has been executed.

If the failed cnode has a status of FAILED, then a discless cnode will update the failed cnode's status to CLEANUP. On the root server cnode, however, the root server will notify all discless cnodes of the failure and wait for all clustered cnodes to respond to this notification before it updates the failed cnode's status to CLEANUP.

The recovery CSP will then attempt to kill all the CSPs serving requests on behalf of the failed cnode. If all such CSPs have been killed, and if the failed cnode's status is now CLEANUP, then the rest of the cleanup functions will be invoked. Otherwise, the recovery CSP will return to the top of the loop and search the cluster status table for the next failed cnode. If the cleanup functions are successful, then the cnode's status will be updated to INACTIVE and the global count of failed nodes will be decremented. However, if some cleanup function was blocked from completing because of a potential deadlock situation, then the cnode's status will remain at CLEANUP and the recovery functions will be reexecuted for this cnode on the next pass through the cluster status table.

When a pass of the cluster status table is complete, the recovery CSP once again checks the global count of failed cnodes. If this count is nonzero, then the cluster status table is rescanned and the recovery algorithm above is repeated. When there are no more cnodes left in the FAILED or CLEANUP state, the recovery CSP terminates. When the recovery CSP terminates, the resources of all failed sites have been recovered and normal execution continues.

Summary

Crash detection and recovery are an important part of making HP-UX clusters dynamic and resilient. Fast and reliable detection of failures allows users to add and remove cnodes dynamically without affecting users on other cnodes. LAN cable failure detection allows changes in the LAN configuration to be made without shutting down the cluster. Different mechanisms work together to make crash detection and recovery in HP-UX clusters both reliable and fast.

Acknowledgments

Chyuan-Shiun Lin and Joel Tesler (then of HP Labs) did the initial design and implementation of crash detection and recovery on which the final implementation is based. Dave Gutierrez did the design and implementation of the LAN failure detection mechanism and was very helpful in the refinement of the networking resource recovery. Bob Lenk, Debbie Bartlett, Barb Flahive, and Fred Richart were all a great help in the refinement of various areas of resource recovery. Mike Berry and Fred Richart developed distributed, coordinated failure simulations, which were invaluable in the testing of this functionality.

Boot Mechanism for Discless HP-UX

by Perry E. Scott, John S. Marvin, and Robert D. Quist

THE IMPLEMENTATION OF A DISCLESS WORKSTATION requires three distinct services: a remote file system, a remote swapping capability, and the ability to load and initialize the operating system from a remote source. All of these services are implemented for the HP-UX 6.0 system with the goal of maintaining a single-system view. For the boot mechanism this means that although the operating system and its loader are on a remote system (i.e., the root server), a user can power up any workstation in a cluster and get the same boot sequence that is experienced with a stand-alone system. A stand-alone system is a workstation that uses a local disc for booting and file system operations. This article describes how the standard HP-UX boot mechanism works, and the modifications made for the HP-UX 6.0 discless implementation.

Overview

The major modules and interfaces involved in the HP-UX system boot mechanism are shown in Fig. 1. Fig. 1a shows the boot components for a conventional stand-alone HP-UX system and Fig. 1b shows the components for a discless configuration. The following sequence outlines what happens when a discless workstation is powered on and booted. A more detailed description of these steps and the components shown in Fig. 1 is given later.

- After power-up, the boot ROM searches for and assigns an input device (keyboard) and an output device (display) to use as a console.
- The boot ROM checks for and tests interface cards, RAM, and other internal peripherals. It then displays the information shown in the left side of Fig. 2. This is called self-test.
- The boot ROM loader polls all supported mass storage devices and LANs connected to the computer for an operating system, and the message SEARCHING FOR A SYSTEM (RETURN To Pause) appears on the display (see Fig. 2).
- If the user strikes the keyboard during self-test the boot ROM assumes the user wants to control the selection of the operating system to boot. This is called the attended mode. When this is done a list of available operating systems appears on the right side of the display (see Fig. 2). The user selects a system by entering one of the two character codes (e.g., 1H). If a key is not struck the boot ROM loader automatically selects the first bootable system it finds. This is called the unattended mode.

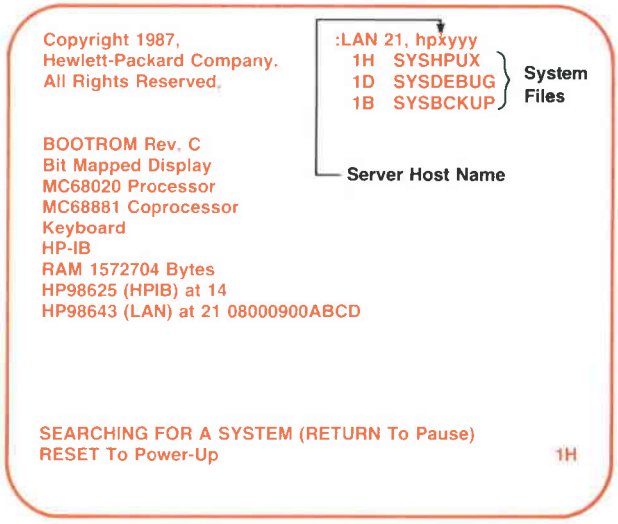


Fig. 2. *A typical screen the user sees during the boot process.*

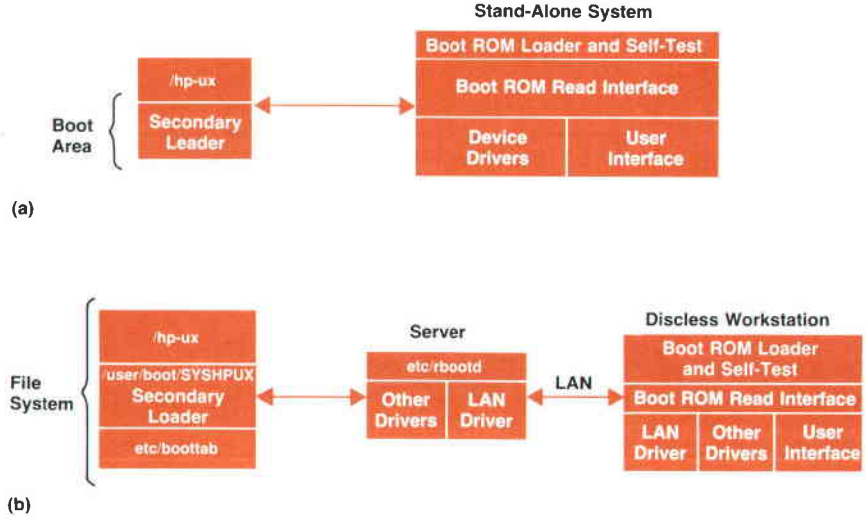


Fig. 1. *(a) The major components involved in the boot process for a stand-alone HP-UX system. (b) The major components involved in the boot process in a discless environment.*

- Once the operating system is chosen (assume 1H) the boot ROM retrieves the secondary loader from the server and loads it into RAM on the discless cnode. Control is then transferred to the secondary loader.
- The secondary loader retrieves the operating system (e.g., /hp-ux) from the server, loads it and transfers control to the operating system.
- The operating system initializes the discless kernel.

The first five steps in this sequence are called the boot ROM phase, and the last two steps are called the secondary loader phase and the HP-UX initialization phase, respectively.

Except for searching the LAN connection and loading the secondary loader from the server, these same actions also take place when a stand-alone HP-UX system is booted. The difference is that the stand-alone system accesses files directly from its local disc instead of going over the LAN. From the user's perspective, the boot process looks the same.

There may be more than one cluster of workstations connected to a LAN cable, and therefore more than one server may exist on the LAN. One of the main features of the discless boot mechanism is that when a booting cnode is polling the LAN connection for an operating system it is able to select the correct server. The mechanism for doing this is explained later.

Discless Workstation Boot Modules

Boot ROM Loader. The HP 9000 Series 300 boot ROM loader is one of the boot ROM modules located in EPROM on the CPU board. After self-test the boot ROM loader initiates communication with the server to retrieve the bootable system files. During the boot sequence, when the boot ROM loader finds a LAN interface it broadcasts a server identify request packet. Typically a cnode belongs to one server; however, there is the possibility for a cnode to be configured with more than one server. Each server has a process called /etc/rbootd listening to the LAN. Based on the information in the server's configuration file (/etc/clusterconf), etc/rbootd decides whether to respond with the server's host name. The host name is then displayed on the cnode's system console. The process /etc/rbootd, which is discussed later, is a server daemon that handles communication with discless cnodes during boot.

For each server responding, the boot ROM loader sends a file list request packet containing a file number. The file number is incremented for each file list request sent to a particular server. As the file names are sent to the requesting cnode they are displayed on its system console (see Fig. 2). This is done until the file number exceeds the number of boot file names the server has available to send. At this point the server responds with a reply packet that indicates there are no more file names to send. When a bootable file is selected (e.g., 1H) the boot ROM sends a request to open the file. This file (e.g., SYSHPUX) is the secondary loader and resides on the server as /usr/boot/SYSHPUX.

In addition to opening the boot file, the boot ROM records several global variables in RAM that are used by the secondary loader and the HP-UX kernel. These values include:

- MSUS (mass storage unit specifier). Information about the boot device, such as the directory format, device type, and select code.
- SYSNAME. The name of the selected operating system (e.g., SYSHPUX).
- SYSFLAG2. The name of the processor type on the cnode (e.g., 68020).
- LOWRAM, HIGHRAM. The low and high limits of system memory.
- F_AREA. A driver scratch area where the LAN link level address of the server is stored. The link level address is retrieved from the IEEE 802.3 packet containing the server's host name.

After the boot file is opened, the boot ROM loader issues a read request packet to the server to read the secondary loader into the discless cnode's memory. When the secondary loader has been loaded, a boot complete packet is sent to close the boot file and terminate the session. The boot ROM then passes control to the secondary loader.

Boot ROM User Interface. The displays produced during boot and the handling of user input are the responsibilities of the boot ROM user interface modules. When a key is struck during self-test (attended mode) the interface module is responsible for assigning the two-character codes (e.g., 1H, 2B) to each bootable operating system that is found. All prompts and error messages go through the user interface routines.

Boot ROM Read Interface. The read interface provides file open, read, and close facilities to the boot ROM loader and the secondary loader, and it functions as an interface to the driver modules. The boot ROM loader uses the read interface to load the secondary loader, and the secondary loader uses it to load the HP-UX system.

The read interface operates in either an absolute mode or a file mode. In file mode, file relative addressing is used to access files on the server. The booting cnode relies on the server to resolve the logical address into physical disc blocks. In absolute mode, device relative addressing is used and the calling routine is responsible for performing the logical-to-physical disc block mappings.

For the discless implementation one of the design goals was to make the read interface to the LAN driver look like other devices so that existing secondary loaders would not have to change. The original HP-UX loader was built on the assumption that it was always booting from a local disc; therefore, it uses the absolute mode. The absolute mode proved impractical for the LAN driver. The HP-UX secondary loader was modified to recognize nondisc devices and use the file mode. We already had secondary loaders for our BASIC and Pascal workstations which use the file mode for boot over the Shared Resource Manager (SRM). The SRM has characteristics similar to the LAN.

Root Server Boot Modules

/etc/rbootd (remote boot daemon). /etc/rbootd is a process that runs on the root server and handles all of the boot protocol requests between the server and the discless workstations. Rbootd uses two files to determine how it should respond to requests from the discless cnodes: a configuration file /etc/clusterconf and a boot table /etc/boottab. The configuration file contains the names and link level addresses of the cnodes associated with the server. /etc/boottab contains a list of boot files available to each cnode in the cluster. Rbootd

detects when changes are made to either of these files and reconfigures itself using the new information.

To allow context dependent boot files (files tailored to the capabilities of the workstation), rbootd emulates the pathname lookup code used by the HP-UX 6.0 kernel to handle context dependent files.The emulation is not perfect since rbootd cannot determine some of the hardware-specific context (e.g., whether the discless cnode has an MC68881 floating-point coprocessor installed). Therefore, hardware-specific context elements are not supported for boot files. Context dependent files (CDF) are discussed in detail in the article "A Discless HP-UX File System," on page 10.

Rbootd supports four levels of error and information logging, ranging from logging only fatal errors to recording the beginning and end of every boot session. The logging level is set with a command line option.

The communication protocol used by rbootd is based on a simple request/reply model. When a packet arrives, rbootd wakes up and processes the packet, usually by sending a reply, and then goes back to sleep. Requests are queued by the link level access driver in the kernel. Because queue space is limited, rbootd uses HP's real-time priority feature to ensure that boot (especially unattended boot) does not fail because of dropped packets.

Several boot protocols were investigated for our discless implementation. The Trivial File Transfer Protocol (TFTP) was considered, but could not be used. First, the boot ROM read interface is random-access and TFTP is sequential-only access. Second, TFTP is built on top of IP, which would require more code in the boot ROM. Finally, the boot ROM must obtain a list of file names, which is not provided by TFTP. We could have worked around many of these limitations; however, we decided to use a version of the Remote Maintenance Protocol (RMP) boot capability. This protocol was already in use within HP and the only capability missing was the ability to obtain a list of files from the server. Investigation showed that special interpretation of certain fields in the boot request packet would allow this feature to be implementated.

Rbootd services five types of requests: server identify, boot file list, boot request, read request, and boot complete. The boot request, read request, and boot complete packet types are standard RMP requests. The server identify and boot file list packet types are extensions to the RMP boot request packet.

- Server Identify Request. In the boot ROM phase the discless cnode uses the server identify request to get a server's hostname. At the same time the server's link level network address is obtained from the IEEE 802.3 packet header sent by the server's LAN driver.
- Boot File List Request. The boot file list request is sent by the boot ROM to obtain the names of the files listed in /etc/boottab. The request packet contains an index number that is used by rbootd to respond with the name of the file. If the number is greater than the number of files available, rbootd responds with a packet indicating that there are no more boot files.
- Boot Request. A boot request opens the requested boot file and allocates a session number. This session number is used by the discless cnode for the read request and boot complete request. Session numbers are used to support concurrent boot requests.
- Read Request. A read request is used to read a boot file. The request packet contains an offset and the number of bytes to be read from the file. This enables the discless cnode to access data randomly from the boot file. Rbootd responds with a packet containing the number of bytes actually read.
- Boot Complete Request. Boot complete causes rbootd to close the boot file and deallocate the session number.

Secondary Loader. In a stand-alone system the secondary

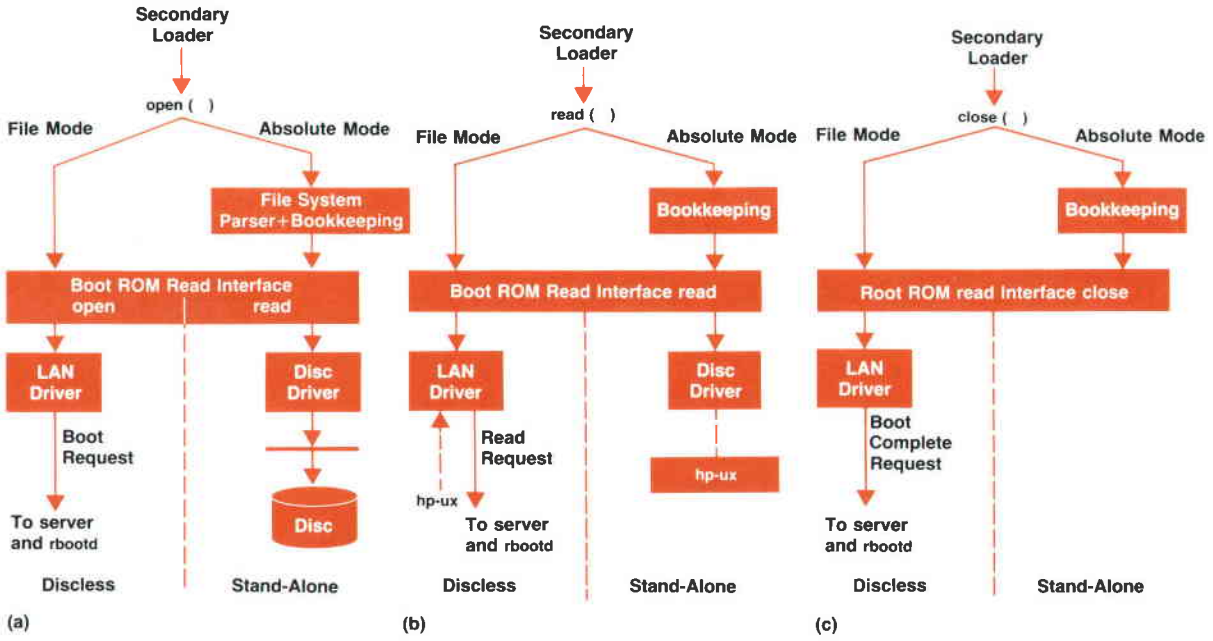


Fig. 3. *Secondary loader control flow on discless and stand-alone system during a) a file open, b) a file read, and c) a file close.*

loader resides in Logical Interchange Format (LIF) in the first 8K bytes of the boot disc. It is transferred to memory by the boot ROM interface routines at the end of the boot ROM phase.The purpose of the secondary loader is to load the /hp-ux a.out file (i.e., the HP-UX operating system) into low memory and execute it. Fig. 3 shows the secondary loader's flow of control and the processes involved for discless and stand-alone loading situations. The open( ), read( ), and close( ) routines emulate the behavior of the HP-UX system routines by the same name, and provide the secondary loader with an interface to the boot ROM read interface open, read, and close routines. The file system parser is a routine that understands the HP-UX file system structure and is responsible for resolving HP-UX pathnames during a boot file open in the absolute mode. Bookkeeping functions include the activities performed to keep track of data being transferred from disc (for instance, keeping a count of the number of blocks and current file offset and size, or processing partial or multiblock data transfers).

The secondary loader starts the loading process by examining the LOWRAM variable to determine the load point for the HP-UX kernel, and then uses the variable MSUS to determine the boot device. The name of the boot file is retrieved from the variable SYSNAME and the boot file name is translated to an HP-UX pathname and the open( ) routine is called. For instance, the boot file SYSHPUX is translated to /hp-ux.

The open( ) routine selects either the absolute or the file mode of the open operation depending on the type of boot device. For local boot the file system parser resolves the HP-UX pathname by using the boot ROM read interface read routine to perform pathname lookup. For a remote boot, as in the discless situation, the LAN driver is invoked through the boot ROM read interface open routine and a boot request is sent to the server where it is processed by rbootd.

The read( ) routine makes the same selection as the open( ) regarding absolute or file mode and uses the boot ROM read interface read routine to access the drivers. For absolute mode the loader uses the bookkeeping function to keep track of character counts, number of blocks read, and block addressing. For the discless situation a read request is sent to the server to be processed by rbootd. The read( ) operation results in transferring the selected operating system (/hp-ux) to the discless cnode's memory. The loading sequence for the operating system proceeds as follows: first the /hp-ux a.out header, which contains the sizes of the text, data, and uninitialized data areas, is read into a temporary area, and then the file /hp-ux is read into memory in two read calls, one for text and one for data.

When the operating system is loaded the close( ) routine is called. For the discless situation this results in a boot complete request being sent to rbootd. For the stand-alone situation the loader does some internal bookkeeping without calling the boot ROM. When the close operation is complete the secondary loader transfers control to the HP-UX kernel.

Kernel Debugger Considerations

The above process changes slightly if SYSDEBUG is chosen instead of SYSHPUX. The kernel debugger is loaded just like the HP-UX kernel. When the debugger is started, it opens the a.out file /SYSDEBUG to find its relocation information, then moves itself into high RAM, adjusting all of its jump points. It then adjusts the HIGHRAM boot ROM variable, effectively protecting itself from being overwritten.

The debugger uses the secondary loader open( ), read( ), and close( ) routines, which are left in high RAM. After the user selects the kernel to boot, the debugger loads the HP-UX kernel like the secondary loader loads the HP-UX kernel.

HP-UX Discless Kernel Initialization

The HP-UX discless kernel finds its server's LAN card address in the boot ROM F_AREA. This value is used to initialize several discless kernel pointers, which effectively turns on the discless message interface. The discless message interface provides the protocol for communication between a discless workstation and the server. The discless message interface is described in detail in the article "The Design of Network Functions for Discless Clusters" on page 20. Once the discless message layer is operational the discless cnode sends a cluster request message to the server. The cluster message contains the discless cnode's LAN address, which is used for security purposes, and its kernel release number, which is used to prevent server or client kernel mismatch.

The server validates the discless cnode's request by comparing the cnode's LAN address against the list kept in /etc/clusterconf. If it is not there the request is rejected. Likewise, the request is rejected if the kernel release numbers do not match. Otherwise, the server broadcasts a message to the rest of the cluster and the discless cnode is admitted. The server then sends a message to the cnode that contains the current system time, a description of the rest of the discless cnodes in the cluster, and the ID of the cnode's root and swap servers. At this point, the discless cnode can use the root server's file system, and control is passed to the /etc/init program. The discless file system is used to execute programs started by /etc/init, and kernel initialization is complete.

Acknowledgments

The authors would like to thank the following individuals who contributed to the discless boot mechanism: Anny Randel for her work on the original /etc/rbootd design and prototype, David O. Gutierrez for his patient explanation of the HP-UX LAN driver, discless messages, and kernel initialization, and Joe Cowan for project management in bringing together the resources to complete the discless boot mechanism.

Discless System Configuration Tasks

by Kimberly S. Wagner

GOING FROM A GROUP of stand-alone machines to a clustered environment is not a particularly difficult task, but because of the large number of steps required to configure the system, an automated tool called reconfig is provided with the HP-UX discless system to simplify the process. Reconfig enables the system administrator to set up the cluster server node and add or delete cluster nodes (cnodes) as necessary.

Reconfig was originally developed for the HP 9000 Series 200 and 300 Computers' HP-UX 5.1 operating system. The tool contains a collection of monotonous and terse system administration tasks within a user-friendly menu-driven environment. Basic tasks such as setting up user access to the system and reconfiguring kernels can be easily accomplished. With the advent of discless workstations in a clustered environment, changes were made to enhance the original reconfig tool.

Cluster Setup

For creating a cluster configuration, the minimum system includes an HP 9000 Model 350 for the root server with at least 8M bytes of RAM, at least a 130-Mbyte disc drive, and the HP-UX 6.0 operating system (or later). The setup process begins by running /etc/reconfig, and when the main menu appears selecting the option Cluster Configuration. This selection will bring up the menu shown in Fig. 1, which shows the four values required to set up a cluster server: the server node name, the link level LAN address, the internet address, and the number of cluster server processes (CSPs).

Cluster Node Name. The server's cnode name is the system's hostname and it is used to identify the server cnode within the cluster. All discless cnodes refer to the root server by this name.

LAN Card's Link Level Address. Each LAN interface card has a unique link level address. This value is set by the factory and cannot be changed. Reconfig will display the address for each LAN attached to the system. If there is only one LAN card on the system its address is used by default; otherwise, one of the available cards must be selected using the NEXT, PREVIOUS, and SELECT softkeys.

NS-ARPA Internet Address. The internet address enables communication with other networks and uniquely indentifies the server within a network. The internet address is not required for discless interaction, but provides a minimal NS-ARPA networking capability within the cluster to handle remote process execution for system administrative tasks. If a value is automatically displayed, that value is the internet address associated with the cnode name that already exists in the system's /etc/hosts file.

Cluster Server Process (CSP). The CSP is a special process that is used to handle interprocess communication in a discless envrionment. Except for one limited CSP (LCSP) which exists on each discless cnode, all the other CSPs exist on the server. The default value is 4 and the amount entered will be the minimal number of CSPs running at all times. If more CSPs are needed they will be created automatically. For an in-depth discussion of CSPs see the article "The Design of Network Functions For Discless Clusters" on page 20.

When all the entries in the menu have been entered reconfig will tell the user what it is about to do to build the system and then ask if the user wants to continue. A yes will cause reconfig to begin configuration. Reconfig performs five steps in transforming the stand-alone system to a clustered environment. The steps are done in a particular order,

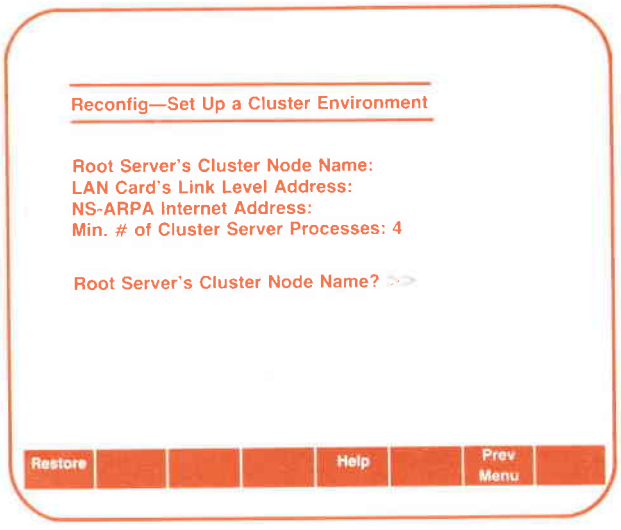


Fig. 1. Reconfig *menu for creating a cluster environment.*

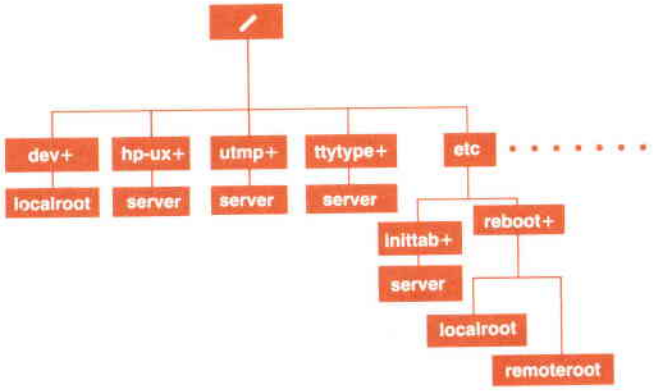


Fig. 2. *Partial system model for the server cnode. "Server" is the cnode name given to the root server.*

so if an error occurs during the process, the user can correct the problem and then reexecute reconfig from where it left off. Generally, each step in the process will complete without error unless certain required files and/or directories are missing. The activities that take place at each step are as follows:

- Context dependent files (designated by a + appended to the file name) are created for the cluster and root server based upon a predetermined system model (see Fig. 2). Context dependent files (CDF) are used to describe the various attributes (e.g., machine type, coprocessors) of a particular cnode. For more information on CDFs see the article "A Discless HP-UX File System" on page 10.
- A fully loaded root server kernel is built. The directory /hp-ux is turned into a CDF and the new version of the kernel resides in /hp-ux+/<cnode_name>. Cnode_name is the name entered earlier for the cluster node name.
- The NS-ARPA files for remote process execution are set up and an entry is made for the root server for the following files: /etc/hosts, /etc/hosts.equiv, and $HOME/.rhosts (root's home directory).
- The cluster configuration file /etc/clusterconf is created and the following information is entered in the file for the root server: the root server's cnode name, the server's link level LAN address, and the number of CSPs to start at boot.
- The rc file, which initiates the boot service for the discless cnodes, is modified to state which LAN device file to use if the default LAN device file is inappropriate.
- Reboot system.

Adding and Deleting Cnodes

Once the root server of the cluster has been set up, discless cnodes can be added or deleted at will by running /etc/reconfig and selecting the Cluster Configuration option from the main menu. If the root server has already been set up (e.g., /etc/clusterconf exists), reconfig will present two menu choices for adding or deleting discless cnodes.

Adding a Cnode

The input required for adding a cluster node is similar to that required for initial cluster setup: the cnode name, an internet address, and the link level address of the cnode's LAN card. Each discless cnode always runs exactly one limited cluster server process (LCSP) so there is no need to prompt for the number of cluster server processes. The process for adding a cnode is similar to that for setting up the clustered environment on the root server. The four steps are as follows:

- Context dependent files are created for the new discless cnode based on the system model for client cnodes (see Fig. 3).
- A minimally loaded discless cnode kernel is built. The directories /hp-ux+/<cnode_name> and /etc/conf/dfile+/<cnode_name> are created.
- NS-ARPA files for remote process execution are set up for the discless cnode. The files /etc/hosts, /etc/hosts.equiv, and $HOME/.rhosts (root's home directory) are modified to include the new cnode.
- The cluster configuration file (clusterconf) is modified to include an entry for the discless cnode. The entry includes the new cnode's name and its link level LAN address.

Removing a Discless Cnode

Only the discless cnodes can be removed with reconfig. All that is required to remove a discless cnode with reconfig is the cnode name. The menu shown in Fig. 4 is used to select the cnode to be removed. There is an option to remove or not to remove all CDFs associated with the cnode. Unless there is a good reason for leaving the CDF elements around, the CDFs should be removed when the discless cnode is removed. The cnode removal process involves the following steps:

- Remove the ability to do remote process execution by deleting the entries for the cnode from the NS-ARPA files /etc/hosts.equiv, and $HOME/.rhosts. The cnode name and its associated internet address remain in the file /etc/hosts for later use.
- Remove the entry in the cluster configuration file (/etc/clusterconf) for the deleted cnode.

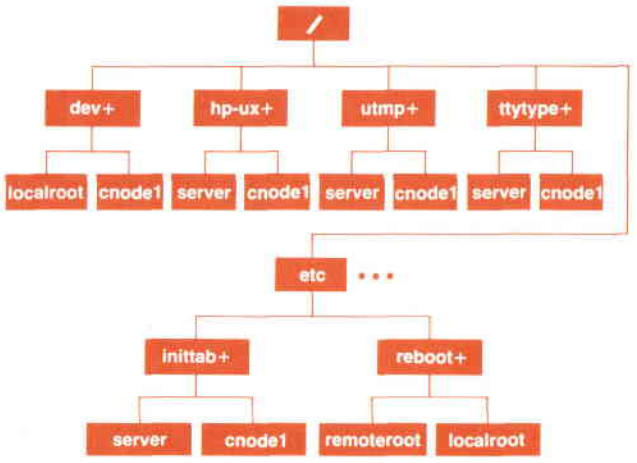


Fig. 3. *Partial system model for a client cnode. "Cnode1" is the cnode name given to the new discless cnode.*

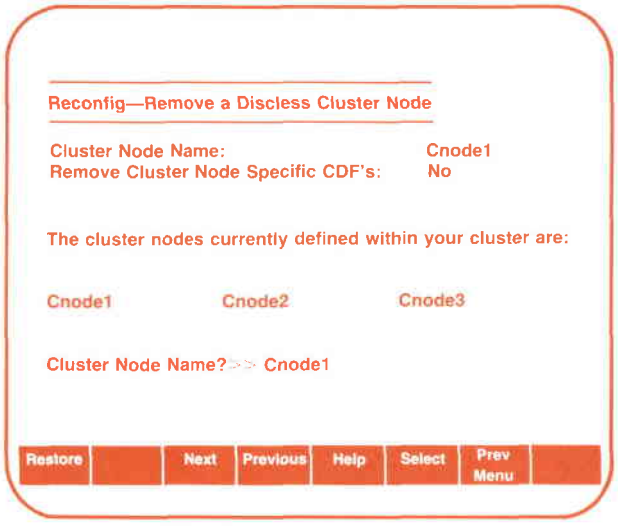


Fig. 4. Reconfig *menu for deleting a cnode.*

- If requested, remove all context dependent file elements of the form : <file>+/<cnode_name>.

Conclusion

The Reconfig tool provides features that make the tasks of setting up and maintaining an HP-UX discless cluster much easier. In addition, the time required for reconfiguration is much lower with Reconfig than it would be to administer each system individually. This is one of the advantages of the HP-UX discless system.

Acknowledgments

Special thanks go to Stuart Bobb and Dave Grindeland for their usability testing efforts, and to Paul Christofanelli and Paul Van Farowe for their invaluable NS-ARPA networking assistance.

Small Computer System Interface

The SCSI standard is the newest interface for the HP 9000 Series 300 family of HP-UX workstations. It offers improved performance, simplicity in design, a wide choice of controller chips, and wide acceptance in the UNIX® community

by Paul Q. Perlmutter

DURING THE PAST FEW YEARS manufacturers of small computer systems and intelligent peripheral devices realized the need for an industry standard I/O interface for their systems. This interest resulted in the Small Computer System Interface (SCSI) standard. HP introduced an SCSI interface in April of 1988 for a family of high-performance disc drives. The SCSI standard is the newest interface for the HP 9000 Series 300 family of HP-UX workstations. It offers improved performance, simplicity in design, a wide choice of controller chips, and most important, wide acceptance in the UNIX community. Marketing data predicts that by mid-1989, approximately one half of all UNIX workstations will have an SCSI interface. This article provides an overview of the SCSI standard and the implementation of SCSI on the Series 300.

What is SCSI?

The Small Computer System Interface—or SCSI—is an intelligent, general-purpose I/O bus. The entire spectrum of requirements for SCSI is specified in one document: ANSI committee standard X3.131. This standard defines the physical layer, the logical interface layer, and the device command set level for peripherals used with small computers. SCSI is very popular partly because all levels were designed and specified together, resulting in an I/O system that is integrated in a consistent and homogeneous style.

A critical design goal for SCSI was to provide the host computer with device independence within a certain class of peripheral devices. For direct-access drives (i.e., discs) this means the features that distinguish different discs are hidden from the software. The disc dependent characteristics such as the disc geometry, timing, protocol, and feature set are elements that make a disc less compatible. SCSI tries to hide these elements from the software without compromising product performance or quality. This significantly simplifies the development of the disc software driver, and enables the software to achieve a high degree of autoconfigurability. It also improves plug-and-play possibilities between different vendors' disc drives. For instance, many disc drive manufacturers have developed command sets that have many common features. SCSI extracts the common ingredients of these command sets and creates an industry standard format, command syntax, and command set. To simplify addressing, SCSI discards the older 3-vector addressing mode (sector, track, cylinder) and adopts the simpler single-vector addressing mode. In single-vector addressing, the disc is viewed as a logical single-dimensional array of blocks, and the software merely specifies the block offset from the beginning of the device.

This approach serves to bring divergent disc-like peripherals such as write-once-read-many optical discs (WORMs), CD ROMs, and flexible discs much closer together.

The Series 300 interface card (host adaptor in Fig. 1) uses the Fujitsu MB87030 controller chip to interface to the SCSI bus. This chip simplifies the software interface to the bus. It contains an 8-byte FIFO buffer, and provides DMA, asynchronous, and sychronous methods of data transfer.

Our SCSI disc driver is very flexible and highly autoconfigurable. Almost no assumptions are made about the disc. When a command first accesses the device, the driver checks to determine if the disc is alive (test_unit_ready command), then asks who it is (inquiry command), and finally, asks the disc drive for two basic geometric parameters: the logical block size in bytes, and the size of the drive in logical blocks. These two values are saved in the buffer header associated with the device, and are used for the duration of the transaction.

The SCSI Bus

Only two devices are allowed to communicate on the SCSI bus at any time. Up to eight devices can be connected to the bus and a unique SCSI ID bit (0-7) is assigned to each device. One of the devices must be the host or initiator. When two devices communicate on the bus, one acts as the initiator and the other acts as the target. The initiator (usually a host system) originates an operation (e.g., read or write) and the target (e.g., disc controller) performs the operation. This operation is similar to the HP-IB talker/listener protocol. An SCSI device usually has a fixed role either as an initiator or a target. However, some devices can perform both roles. A typical SCSI configuration is shown in Fig.1.

An important assumption made by SCSI forces the target to drive the bus phases, and the target is allowed to disconnect from the bus when it anticipates a significant delay during data transfer. This fundamental assumption allows multiple drives to be active simultaneously, enhancing total bus bandwidth. The idea is this: since we can have only one initiator and one target active at any given time, we do not want a device to tie up the bus unless data is actively being transferred. Thus, devices are allowed to disconnect while internal-only activities (such as seeks or command parsing) are occuring. Typically, after a command has been transferred, a device will disconnect while it parses and decodes the command, seeks to the appropriate cylinder, and prepares itself for data transfer. In addition, if in the middle of a data transfer the drive anticipates dead time (such as a seek to a spared track), the device will get off the bus to allow other peripherals to access the host. As soon as the target is ready to resume data transfer, it can actively arbitrate for the bus (when the bus is free) to reattach to the host. The disconnect/reconnect option in SCSI can boost overall system performance.

SCSI Bus Signals

The SCSI bus consists of eighteen signal lines. Nine are used for control and nine for data. The control signals are used to establish the logical bus phases (discussed in the next section) for the SCSI bus protocol, and control the transfer of data. These bus signals are shown in Table I.

Table I
SCSI Bus Signals

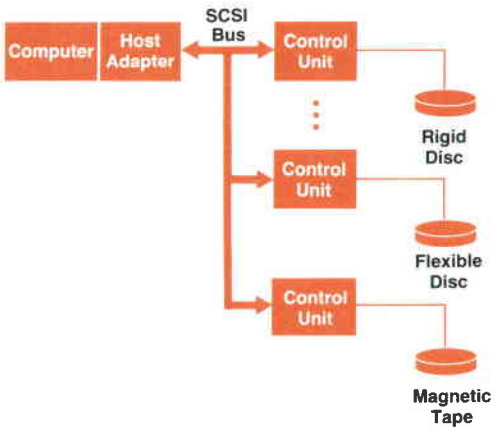
Signal	Description	Driven by Host	Driven by Target
REQ	(Request) Data handshake line: requests data byte on bus.		X
ACK	(Acknowledge) Data handshake line: acknowledge data byte on bus.	X	
BSY	(Busy) Indicates the bus is busy.	X	X
SEL	(Select) Used during selection and reselection to establish communication link.	X	X
I/O	(Input/Output) Indicates direction of data flow on the data bus. If I/O is true the flow is from target to initiator.		X
MSG	(Message) Indicates the data on bus is a message (only valid if C/D asserted).		X
C/D	(Control/Data) Determines whether control or data information is on bus.		X
ATN	(Attention) Requests message out phase (initiator has message for target).	X	
RST	(Reset) Hard reset of all devices.	X	X

In addition there are 8 data lines and one parity line that are driven by both devices.

SCSI Bus Phases

The SCSI architecture defines eight distinct bus phases that define the logical characteristics of the SCSI bus:

- BUS FREE phase. Indicates when no SCSI device is actively using the bus.
- ARBITRATION phase. Allows one SCSI device to gain control of the bus.
- SELECTION phase. Allows an initiator to select a target.
- RESELECTION phase. Allows the target to reconnect to the initiator.
- COMMAND phase. Allows the target to request command information from the initiator.



- DATA phase. Provides data transfer between the initiator and the target.
- STATUS phase. Provides status information from the target to the initiator.
- MESSAGE phase. Allows the transfer of messages between the initiator and the target.

The first four phases allow devices to contend for access to the bus and establish the physical data path between the initiator and the target. The last four phases are called the information transfer phases because they are used to transfer data and control information over the data lines. The SCSI bus can never be in more than one phase at any given time. However, all devices can arbitrate for access to the bus. The following is a simple example of phase sequencing during a disc transaction:

Phase	Comments
BUS FREE	No device on bus.
ARBITRATION	Initiator (host) arbitrates for bus.
SELECTION	Host establishes contact with target.
MESSAGE OUT	Host identifies itself to the target.
COMMAND	Host issues command (e.g., read or write).
MESSAGE IN	Target indicates it will disconnect and then drives the bus to BUS FREE.
BUS FREE	Target is detached from the bus while a seek is in progress.
ARBITRATION and RESELECTION	Target is read and reestablishes a link to the host.
MESSAGE IN	Target identifies itself to host.
DATA IN or OUT	Data is transferred.
STATUS	Target reports on transfer status.
MESSAGE IN	Command complete (done).
BUS FREE	

Bus Access Phases

BUS FREE Phase. This phase indicates that no device is using the bus and that it is available for use. BUS FREE is detected by the BSY (busy) and SEL (select) signals being false.

ARBITRATION Phase. Arbitration allows a device to gain control of the bus to initiate a transaction such as a data transfer, or to send a message or command. To gain control of the bus, a device (the initiator) first checks to see if the bus is free. If the bus is free the device asserts the BSY signal and sets its own device ID on the data lines. If more than one device is contending for the bus, the device with the highest priority gains access to the bus. The device that loses arbitration starts all over again and the device that wins asserts the SEL signal to end arbitration.

SELECTION Phase. After a device has gained control of the bus the SELECTION phase is entered by selecting the target device for the transaction. Target device selection is accomplished when the initiator sets the data bus lines to the OR of its SCSI ID bit and the target's SCSI ID bit, and asserts the ATN signal. The target will respond by asserting a MESSAGE OUT phase requesting an Identify message from the initiator. The Identify message establishes the physical data path between the initiator and the target. The initiator replies with a message indicating to the target whether it can handle target disconnection, and it determines if the target can handle synchronous data transfer. If an SCSI implementation does not support messages the target will go directly to the COMMAND phase.

RESELECTION Phase. When the target decides to disconnect from the bus temporarily, the RESELECTION phase is used to reestablish connection with the initiator to continue a transaction. The target device disconnects to free the bus for other devices to use when it anticipates a significant delay during the next data transfer. For instance, during a disc I/O operation the disc can disconnect from the bus while it switches heads, does a seek, or empties its controller's buffers. Before disconnecting from the bus, the target sends the messages Save Data Pointers and Disconnect to the initiator. The Save Data Pointers message tells the initiator to save the pointers to the current locations in its memory where the data is being transfered. The pointers are restored when the target reconnects to the initiator. When the target is ready to resume data transfer it must wait for BUS FREE, arbitrate for the bus, and then reselect the initiator. In implementations where there is no ARBITRATION phase the RESELECTION phase cannot be used. This also implies that the target device is not allowed to disconnect from the bus.

Series 300 and Bus Access. The Fujitsu chip controller used on the Series 300 interface card provides a very flexible SELECT command. For the host (initiator) to arbitrate and select a target, the host first writes the target ID bit to the TEMP register on the chip, and then issues the SELECT command to the chip. The chip handles the ARBITRATION and SELECTION phases, and will interrupt with one of three possible conditions:

- Command complete interrupt (selection completed). This indicates that the arbitration was successful and the target device responded to the SELECTION phase.
- Command complete interrupt (arbitration for the bus failed).
- Time-out interrupt (the target device did not respond, possibly because the device was powered off, or the device at the bus ID is not present).

Information Transfer Phases

The information transfer phases are used to transfer data or control information over the data lines. These four logical phases are distinguished by three control lines: MSG, C/D, and I/O (see Table II).

Table II
SCSI Information Transfer Phase Coding

Control Line			Phase	Direction of Transfer
MSG	C/D	I/O		
0	0	0	Data Out	Initiator to target
0	0	1	Data In	Target to initiator
0	1	0	Command	Initiator to target
0	1	1	Status	Target to initiator
1	1	0	Message Out	Initiator to target
1	1	1	Message In	Target to initiator

An important characteristic of SCSI is that the target drives the three control lines, and therefore controls all

changes from one information transfer phase to another. The initiator can request a MESSAGE OUT phase by asserting ATN. The initiator might use this feature to gain the attention of the target under special circumstances. For instance, suppose the host has decided that some type of catastrophic failure has occurred during the DATA phase (e.g., a parity error). The host (initiator) will assert ATN, and when the target recognizes the ATN condition, it switches the bus phase to MESSAGE OUT and allows the host to send a message. When this situation occurs for the Series 300 an Abort message is sent that tells the target to clear the current command and all status and data buffers, and allow the bus to go to BUS FREE immediately. The host may then attempt to retransmit the entire transaction or issue an error to the process that made the transaction request.

The REQ/ACK signal lines provide the handshake protocol used to control the asychronous and synchronous transfer of information during the information transfer phases. Each REQ/ACK handshake allows the transfer of one byte.

Asynchronous Data Transfer. The target uses the I/O signal to control the direction of transfer. If I/O is true the direction is from target to initiator (e.g., read), and if I/O is false the direction is from initiator to target (e.g., write). Fig. 2 illustrates the REQ/ACK handshake protocol which is repeated for each byte transferred. The SCSI asynchronous data rate is 1.5 Mbytes/s.

Synchronous Data Transfer. Asynchronous data transfer is the primary mode available in SCSI. However, synchronous data transfer is possible during DATA IN and DATA OUT if before transfer the initiator and target agree to this mode. When the synchronous mode is established the devices also agree on a minimum period between REQ and ACK signals and the maximum REQ/ACK offset. The REQ/ACK offset is used to determine the number of REQs the target will send in advance of the number of ACKs received from the initiator. During synchronous transfer the target does not wait for the ACK signal from the initiator before sending the next REQ signal to send or receive the next byte. The target will continue in this loop until the specified REQ/ACK offset. It will then compare the number of REQs with the number of ACKs to verify that all data has been transfered. The SCSI synchronous data rate is 4 Mbytes/s.

COMMAND Phase. When the target is ready to accept a command from the initiator it will drive the SCSI bus to COMMAND phase. The initiator will then send a command such as a read or write to the target.

DATA Phase. When data flows from the target to the host, we refer to this as the DATA IN phase, while DATA OUT indicates that data is going from the host to the target. This is the only information transfer phase that allows the synchronous data transfer option described above. For all other phases data must be transferred asynchronously.

MESSAGE Phase. When the MSG line on SCSI is asserted the data on the bus is interpreted as message bytes. Like the DATA phases, MESSAGE IN indicates that the target is sending a message to the host, while MESSAGE OUT indicates the message is going from the host to the target. Message bytes are used to help establish and coordinate the environment for data transfer. For instance, the Identify byte sent by the host during MESSAGE OUT identifies the host to the target and also indicates to the target whether it can handle disconnects and reconnects during data transfers. In a similar way, the target sends the host a Disconnect message to alert the host that it will immediately disconnect from the bus and drive the bus to BUS FREE. Messages are usually single bytes, but under certain situations are multiple bytes. For instance, extended messages are sent by the initiator to the target to determine whether synchronous transfer is feasible, and if so, to establish the synchronous data rate. During a transaction the MESSAGE OUT phase is initiated by the target in response to the ATN signal.

STATUS phase. The STATUS phase enables the target to inform the intitiator of the status of a transaction. After the target has completed a data transfer it sends one status byte back to the initiator. If the target sends a zero, the transaction completed normally. A nonzero status indicates that the target has additional status to send.

In the Series 300 implementation, if the status byte returned by the target is nonzero, we always request extended status. The Request_Sense command provides complete diagnostic results of the previous transaction.

In addition to bad status, other types of problems may occur. The Fujitsu chip may report parity errors or hardware errors that occurred during a transfer. Another error occurs when a time-out occurs. Whenever any hardware activity is initiated, a timer is started, and if the timer times out we assume a hardware failure has occurred.

Our error recovery philosophy is to give most transactions a second chance and no more. For instance, if at any point during a transaction a parity error occurs or a timer times out we always retry the transaction. We do not try again after a failure on the retry. The only exception to this second-chance rule is when the target reports through extended status that it could not recover from a drive error. In this situation we assume that the device's controller is smart enough to retry the transaction itself.

Series 300 and Information Transfer. The controller chip provides three methods of data transfer:

- Manual transfer. The host processor controls handshake lines.
- Hardware transfer with fast handshake. The chip controls the data transfer and the processor feeds the bytes to the controller's buffers.
- Hardware transfer with DMA.

Manual transfer is currently used for transferring messages and status bytes over the SCSI bus. The fast handshake option of the Fujitsu chip is used for transferring commands, while the hardware transfer with DMA is used for transferring data buffers. When a DMA channel is unavailable, the hardware fast handshake option is used to transfer data buffers.

A Disc Transaction

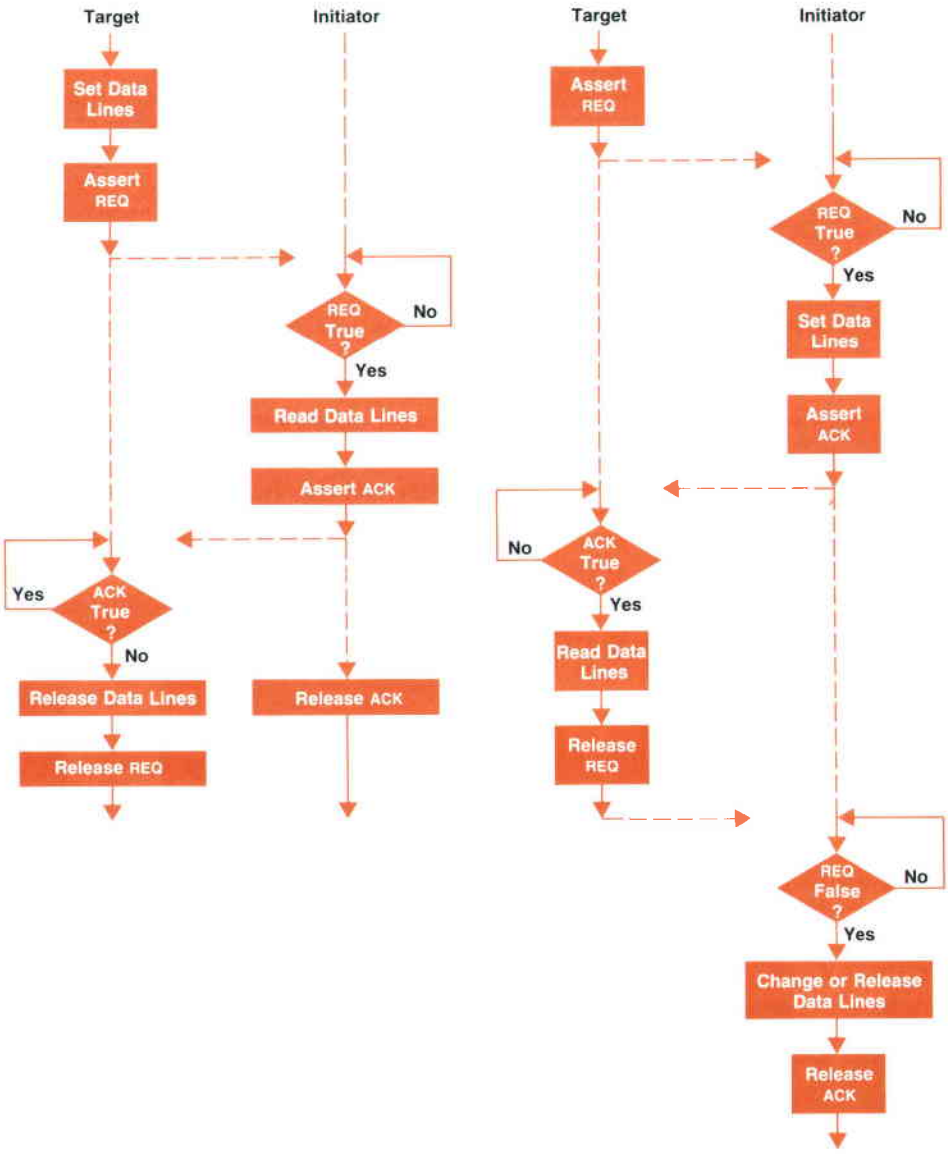
The following discussion summarizes the interactions that occur in the operating system (HP-UX), in the disc driver, and on the SCSI bus when a typical disc transaction is performed.

A disc transaction starts with a disc I/O request from the file system or other higher-level portion of the operating system. The request is passed to the driver via a buf header. This structure includes such information as the system device identifier, the data buffer address in memory, the

block offset on the disc, and the byte count of the buffer. The system device identifier encodes the major and minor numbers that are used by the UNIX system to specify special device files. The major number selects the appropriate device driver (in this case SCSI) and the minor number specifies the select code, bus ID, and unit number. The block offset on the disc is the logical offset viewing the disc as a linear array of blocks. The byte count given in the buf must be converted to a block count appropriate for that device.

When the system begins to service the I/O request and the driver gets permission to use the interface, the driver goes through the ARBITRATION and SELECTION phases to gain access to the disc. The disc (now the target) responds with a MESSAGE OUT and gets the Identify message from the host (initiator). Through the Identify message the driver tells the disc whether it can handle disconnect and reconnect during subsequent DATA phases. The driver asserts the ATN line to maintain the MESSAGE OUT phase and determines whether the disc is capable of synchronous data transfer.

Once the environment for data transfer is established, the driver issues to the disc the 6-byte or 10-byte command (COMMAND phase) that designates whether the operation is a read or write. When the disc receives the command it sends Disconnect and Save Data Pointers messages to the host, and then disconnects from the bus. The driver frees the interface for other processes to use while the disc is busy processing the command. The disc controller decodes the command to determine if it is a read or write operation, and then causes the physical mechanism to perform a seek operation. When the disc is ready to start the data transfer it reselects (RESELECTION) the host and data transfer begins. The disconnect (MESSAGE OUT), reconnect (RESELECTION) and data transfer (DATA IN, DATA OUT) phases may happen several times during the transaction. When all the data has



(a) Read (target to initiator, I/O signal = true)

(b) Write (initiator to target, I/O signal = false)

Fig. 2. REQ/ACK *handshake protocol.*

SCSI and HP-IB

People are often comparing the HP-IB interface bus that is the standard bus for HP 9000 Series 300 Computers with our newest interface: SCSI. Both buses have many features in common, as well as some significant differences. The following is a short summary comparing the two interfaces.

	HP-IB	SCSI
Design Goal	Instrument Bus (later adapted for high-speed interface)	Disc Interface
Transfer Protocol	Asynchronous only	Asynchronous and Synchronous
Bandwidth	~1.2 Mbytes/s burst rates	Async. ~1.5 Mbytes/s Sync. ~4.0 Mbytes/s burst rates
Features	Devices may be powered off	Devices must always be powered on
	8 devices per bus	7 devices per bus
Parity	No	Yes
Bus Topology	Any style (e.g., star) except closed loop	Must be linear
Physical	One connector per device	Two connectors per device: one for input and one for output
	Connectors are sexless	Connectors have male and female ends
	Devices can be daisy-chained; connectors are stacked	Devices must be daisy-chained serially
	Termination not required (open-collector drivers eliminate need for terminators)	Bus must be terminated
Cable Length	Maximum cable length 8 meters total and at most 1 meter per device.	Maximum cable length 6 meters—single-ended only. The differential option offers up to 25 meters but is unavailable on the Series 300.
Handshake	3 handshake lines: single talker and multiple listeners. High-speed peripherals only use single talker and single listener.	2 handshake lines: single talker and single listener.
Addressing	Only hosts can select devices. Devices can respond to parallel polls. Thus, no bus arbitration is required. Primary HP-IB addressing establishes point-to-point communication.	Hosts can select targets, and targets can reselect hosts. Bus arbitration is required.

In addition to the interface comparison, we can also compare the two disc protocols. In fact they are quite similar. Command Set 80 (CS-80), which is the HP-IB disc protocol, uses a command packet of variable size that allows a combination of transparent commands, addressing commands, and CS-80 commands. SCSI uses a fixed-format packet of either 6, 10, or 12 bytes. Message bytes are used in SCSI to complement the command by establishing the environment.

Example comparing read commands:

	CS-80 (HB-IB)	SCSI
Device Addressing	Unlisten Bus Talk (controller) Listen (device)	Arbitrate Select device Identify **Message In** byte
byte 0	Set unit 0	Read Command
byte 1	Set volume 0	Address byte (set unit)
byte 2	Set address	Logical block address
byte 3	Data address	Logical block address
byte 4	Data address	Logical block address
byte 5	Data address	Logical block address
byte 6	Data address	Reserved
byte 7	Data address	Transfer length (upper byte)
byte 8	Data address	Transfer length (lower byte)
byte 9	**No-Op**	Control byte
byte 10	Set length	
byte 11	Data length	
byte 12	Data length	
byte 13	Data length	
byte 14	Data length	
byte 15	Read command	

The HP-IB device addressing is almost equivalent to the **ARBITRATE** and **SELECTION** phases of SCSI. The **Message In** byte in SCSI selects the unit and also indicates to the target whether the host can accept disconnects. Although CS-80 commands appear longer, the difference in system performance is negligible. The **No-Op** in CS-80 is required because of the HP-UX C compiler. CS-80 uses a transfer length defined in bytes and SCSI uses the number of blocks.

been transferred, the status (**STATUS** phase) is sent to the host.

A requested data transfer may be broken into a series of shorter requests based on the device's requirements. The target device drives the bus phase and the host must be prepared for a phase change anywhere during a data transaction. Typically phase changes occur on logical block boundaries, but this is not guaranteed, and no assumptions can be made by the host when the phase change may occur. A typical transaction is shown in Fig. 3.

Conclusion

Our objective in implementing SCSI on the Series 300 was to provide an industry standard interface that added flexibility, expandability, and improved performance to our product family. Our customers wanted to use peripher-

als unavailable from HP, and in some cases, SCSI enables them to do this. As an example, CD ROM support came without any change in the driver. Other customers hoping for plug-and-play compatibility wanted to buy inexpensive peripherals to lower their system cost. Here caution is necessary. SCSI does not automatically imply plug-and-play compatibility. SCSI merely sets up some basic frameworks for hardware and software designers. Options and vendor-specific commands are plentiful. Within commands there are frequently vendor-specific fields or options. In hardware, SCSI allows two different types of transfer, single-ended for short distances and differential for longer distances, which are incompatible. SCSI allows a variety of connectors and cabling. With this type of variability, any two devices may not be mutually compatible.

Perhaps the greatest advantage afforded by SCSI is its simplicity in design. This goal is admirably achieved. It simplifies the design and development of software drivers, and most important, it expedites testing and system integration.

Acknowledgements

The SCSI product on the Series 300 was to some extent a "grass roots" effort. My thanks to John Byrnes as the key person in helping to get the project off the ground. Adding a second disc interface to the long tradition of HP-IB was not easy and John worked hard in achieving this success. Evan James came onto the SCSI project as product marketing manager and turned out to be a tremendous success in keeping the project moving and balancing the lab's requirements with those of marketing. Evan and John did an excellent job driving the team to get the job done.

Thanks to Shaw Moldauer for slipping an SCSI interface into the Model 319 almost undetected and thanks to Dave Kinsell for designing the Model 350 board.

The debugging of the software during critical moments in its life cycle was helped by Steve Wolf and Drew Anderson. The bugs that are found when hardware and software are being developed simultaneously are sometimes very difficult to contend with. When megabytes of data are flowing over the bus every second and only one or two bytes are occasionally wrong, it is a challenge beyond belief. Steve's help at some of these moments of despair was invaluable.

People who are able to straddle that magical wall that separates hardware from software are special people to HP indeed. I was fortunate to work with an excellent team.

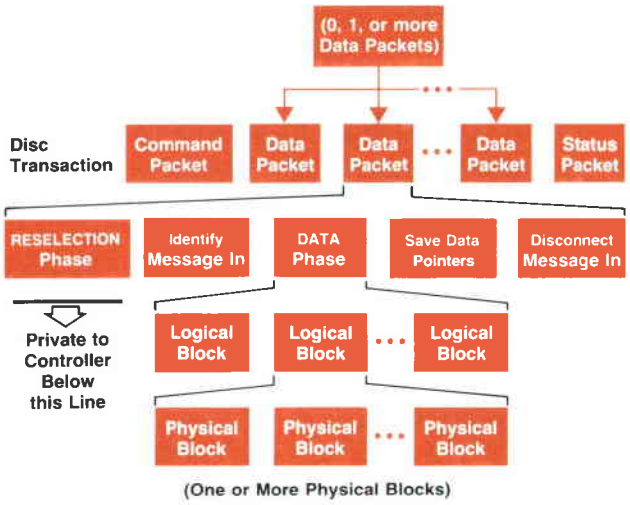


Fig. 3. *Data packets for a typical disc transaction on the SCSI bus. The Command packet contains the information for setting up the environment for data transfer. Information such as the* Identify *and the* Synchronous *messages are contained in this packet.*

X: A Window System Standard for Distributed Computing Environments

The X Window System allows applications running in different environments and on different machines to communicate high quality, graphical user interfaces over a network.

by Frank E. Hall and James B. Byers

The X WINDOW SYSTEM* has emerged as the industry standard for supporting windowed user interfaces across a computer network. It was originally developed at the Massachusetts Institute of Technology (MIT) as part of Project Athena, a large research project investigating networks of computing systems from multiple vendors. MIT has facilitated the acceptance of X as a standard by placing it in the public domain, distributing the standards definition documents and the source code of sample implementations for public use for a nominal fee.

The X Window System is network transparent, which means that an application running on one vendor's computer can display a high-quality, graphical user interface to a user sitting either at that same system or at another computer across the network, perhaps made by a different vendor. The location of the application's target display is not material to the application, and is determined by a parameter when the application is run.

X is virtually independent of the underlying hardware and operating system. The X software adjusts for differences in display or computer architecture automatically as the packets of interactive graphical information are exchanged between application and display according to a well-researched, efficient protocol. This protocol forms the heart of the X Window System standard. Since applications participate in this protocol through a standard programmatic interface library, applications written to the X library are highly portable to other computer systems that support X.

These features combine to make the X Window System a significant enabling technology that allows application developers, end users, and computer hardware vendors to explore the possibilities of the distributed computer environment relatively unencumbered by proprietary barriers that have prevented such seamless integration in the past. For application developers, X promises easier porting, which can allow them to reach a wider market while spending less time on porting and more time on writing better software. For end users, X promises more and better software, and more choice in hardware.

Accordingly, support for X has gained rapid momentum among hardware vendors. HP was among the very first computer manufacturers worldwide to sell X as a product when in March 1987 it began shipping the X Window System for HP-UX, HP's version of the UNIX® operating system. X is now publicly endorsed as a standard by over 40 computer vendors in the U.S.A., Europe, and Japan, including virtually every major manufacturer of UNIX work-stations.

The increasing power of the distributed computing environment, as demonstrated by the other articles in this issue, makes X a very timely technology. It has integrating implications for the areas of user interface, graphics, and networking. It also presents new challenges for addressing the emerging distributed computing market.

In this paper, we will compare the architecture of X to conventional window systems, and describe the industry efforts to support X as a standard.

*The X Window System is a trademark of the Massachusetts Institute of Technology.

The Basics of Window Systems

A window system is a low-level set of interactive graphics primitives that provide an application with efficient means to create, manipulate, and destroy communication regions or windows on the user's display.[1] The application uses the primitives to send simple graphics or multifont text in color or black and white to the window.

The basic unit out of which the window system builds both text and graphics is the pixel, which is the smallest directly accessible graphical element of the display, usually a very small squarish dot. On a monochrome display the pixel's value can be represented by a single bit. On a color display, the pixel contains an integer color value consisting of multiple bits, depending on the color depth of the display. For high performance, the pixels are typically accessed by memory mapped I/O techniques. Displays of this type are generally referred to as bit-mapped displays.

The window system allows an application to own many windows on the bit-mapped display at the same time, and several applications to share access to the display simultaneously. The window system provides such basic output functions as clipping, drawing, text placement, color map management, and output multiplexing to multiple windows. It provides basic input functions by collecting and routing to the appropriate applications any events received from the user's input devices, which typically consist of at least a keyboard and a pointing device such as a mouse. This allows the applications to share the input devices

without having to engage in explicit arbitration among themselves.

Since user interface style is an evolving field, it is desirable that the window system remain free of a specific user interface policy so that it can efficiently implement alternatives. For example, the methods by which the user employs the input devices to direct the placement, movement, sizing, and shuffling of windows on the display, or to designate which window shall receive keyboard input, is a question of user interface policy that can be delegated to a higher level of software called a *window manager*. Similarly, the window system need not contain specific user interface components such as menus or scroll bars. These style building blocks can be delegated to a higher level of software called a *user interface toolkit*. Finally, the window system should not unduly restrict the type or number of simultaneous terminal emulators through which the user accesses window-dumb applications that were written to talk with a serial terminal. The ideal window system is able to support a wide variety of possible window managers, toolkits, and terminal emulators.

These items often accompany a window system and occasionally become entangled with its design. We will return to these items in more detail below with regard to window system architecture.

Convential Window Systems

Conventional window systems allow window applications to access the display device directly through calls to the operating system kernel, which is often extended to facilitate arbitration of display resource conflicts and window system communication. These applications must therefore reside on the local system.

A schematic of a conventional window system architecture is shown in Fig. 1. Here window applications A and B, linked with the window system library, share access to the display while terminal-based application C, which normally talks to a serial terminal, appears through a window provided by a terminal emulator module for backwards compatibility with the time-sharing environment. While the terminal emulator must reside locally, C may reside either locally or on a remote system that has been accessed through a network service that simulates a serial connection. In either case, C has no knowledge that it is talking to anything other than an ordinary serial terminal. User operations to shuffle and arrange the visible windows are provided by the window manager module, which is tightly coupled to the kernel and communicates with the window system library code linked with each window application.

The window manager and terminal emulator modules are often so closely integrated with the window system that alternatives cannot easily be substituted. User interface components such as scroll bars or menus, while they may be present in the window manager and terminal emulator, are often not available to the application developer.

Conventional window system architecture is more varied and complex than this simple diagram can indicate. For example, the window manager and terminal emulator may be completely implemented in the kernel, or window management may be supported by redundant code linked with the window system library into each application.

While conventional window systems were a great leap forward from the terminal-based time-sharing environment, their greatest problem in a distributed computing environment stems from their greatest strength, which is the direct display access that they provide for applications. Since this requires that window applications reside locally with the display, they cannot be accessed on a remote system. To access a remote application the user must go through a terminal emulator, thereby dropping back to the previous era's user interface paradigm.

Conventional window systems are therefore inherently limited in the distributed computing environment by their stand-alone, non-networked design.

The X Window System

The principal feature that distinguishes X from a conventional window system is its network transparency.[2] The X Window System allows window applications, or clients, to access the display only through the display server, which is a separate process that arbitrates resource conflicts and provides display, keyboard, and mouse services to all clients accessing the display. X can support a spectrum of hardware displays ranging from small monochrome units to advanced graphics systems with up to 32 bits of color per pixel.

The client and the display server exchange information only by means of the X Window System protocol which can be implemented via any reliable byte stream. In the HP-UX implementation of X, as in most others, this byte stream is implemented as a socket, which is a logical data connection between two processes on the network. Clients may reside locally with the display server, or on a remote system across the local area network (LAN). A performance optimization bypasses physical LAN access when the client and display server are local to each other.

Because the client program and the display server are two separate entities, the target display can be specified at the time an application is run. The client program is indifferent. It sends out X protocol commands, which the network services route to the target display server, which then executes the command.

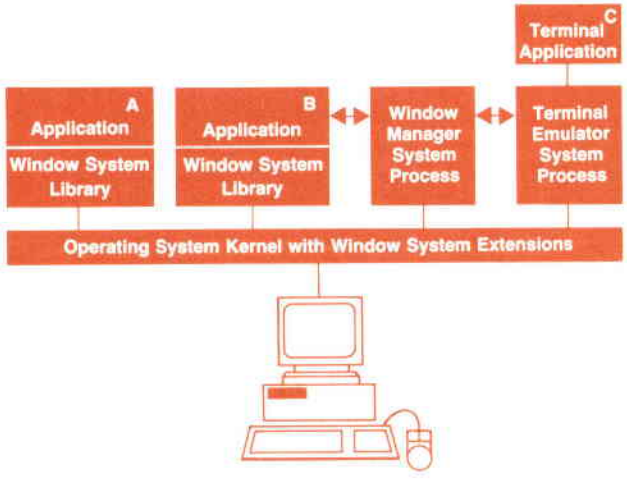


Fig. 1. *Conventional window system architecture.*

Note that the notion of a display server complements the notion of servers as commonly used in discussions of networks. Servers on the network provide applications with access to resources such as files, printers, or computational power. The display server rounds out that picture by making a given display on the network available to applications as a user interface resource.

In Fig. 2, window applications A and B, linked with the X Window System library, share access to the display while terminal-based application C again appears through a window provided by a terminal emulator client. User operations to shuffle and arrange the visible windows are provided by the window manager, which is an otherwise ordinary client whose function is to communicate with the display server to provide user interface policy for the user to manage the display layout. By definition the window manager does not specify how the user interacts within an application. In fact, applications can be written to execute properly with no window manager present. Sometimes this is desirable.

The window manager, user interface toolkit, and terminal emulator are cleanly separated from the X window system, so the user can substitute an alternative if desired, or even omit the item. Note that application B has chosen to use a user interface toolkit, while A has not.

Applications A, B, and C, the window manager, and the terminal emulator may be either local to the display server or on a remote system, or in any combination thereof. The only restriction is that the underlying operating system must support multitasking if the display server and a client are to reside on the same system.

For multitasking systems, it is customary for the window manager and display server to reside locally, and for the terminal emulator to reside on the system where its terminal-based application resides. A single-tasking system can execute only the display server or a single client at a time. In this case the display server can be used as a viewport onto the network. The window manager and all other clients can reside on computational servers elsewhere on the network.

The components of the X Window System standard itself are small in number. At the lowest level, it is simply a document that defines the X protocol.[3] At the programmatic level, it is a document that describes a standard programmatic interface, or window system library, by which an application participates in the protocol.[4] To facilitate ports of the X system, the MIT distribution contains source code of sample implementations of the X library and display server.

While the X library description distributed by MIT is defined for access from the C programming language, programmatic interfaces for other languages can be and have been developed. HP supplies a Fortran bindings package for the X library as part of its X Window System product.

Window Manager

At the outset, students of window systems sometimes confuse a window manager with the window system. The following scenario illustrates the role played by the window manager working in conjunction with the capabilities of the X Window System. Fig. 3 shows how the display might look after the activities described in the scenario.

Suppose that the user has just brought up the X Window System through a script that will later start some client programs. At this point, only the X Window System is running, so the display shows only blank background, which is referred to as the root window. The system cursor, controlled by the mouse, rests in the center of the display. When the script starts the window manager for that display, the appearance of the display does not change. The script also starts two other client programs at the beginning of the session. One is a simple clock program that displays the current time in a corner of the screen. The second is a terminal emulator that opens a window on the display and waits for the user to type a command.

When the user presses the right mouse button, the window manager, which has requested the display server to notify it of all mouse events that occur when the cursor is directly over the root window, receives a notification of the event in its input queue. Let us assume that the user interface policy of this window manager honors a right button press over the root as a command to present a menu, the contents of which the user has specified in a start-up file. In actuality the meaning of this button event could be changed through the start-up file, which would allow a left-handed person, for example, to reverse the window manager's meaning for the left and right buttons.

After notification, the window manager prepares the menu contents and presents it on the display using a component from a popular user interface toolkit. Since this toolkit is also used by many applications, the user is familiar with the operation of this type of menu. The title of the menu is Launch, and it contains the names of the programs the user most often wants to start up. The user selects a program name from the menu. The window manager executes the instructions that the user specified in the start-up file as corresponding to that selection, which in this case is to start up a program directed to the local display that provides the user with a control panel that is used to

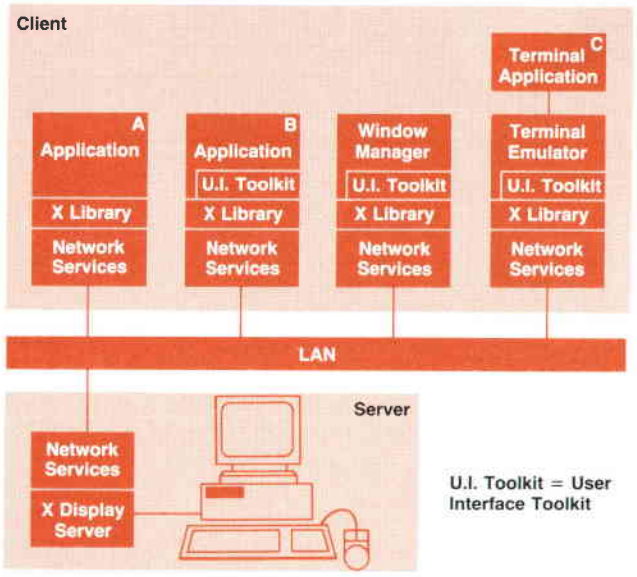


Fig. 2. *X Window System client-server architecture.*

monitor and direct various processes across the network.

The new program establishes contact with the display server, opens a window on the screen, and draws the control panel. When the user clicks the mouse over the appropriate buttons on the control panel, a simulation program begins on a large mainframe computer. The simulation program sends its graphical output across the network to a new window that it opens on the user's display.

While watching the simulation, the user uses the Launch menu to select a program that enables the user to browse through the contents of a remote file system, and then later the user brings up an application to read electronic mail. At this point the screen is too cluttered so the user iconifies one window by bringing up a menu specific to the window manager, and selecting the iconify option. This changes the appearance of the system cursor to indicate that the user needs to pick the window to iconify. The user does this by moving over to the terminal emulator window and clicking on it. The window manager immediately unmaps (removes without destroying) the terminal emulator window from the display, and as a placeholder puts in a convenient spot on the display a small, named, meaningful symbol of the application, called an *icon*. In this way the user also iconifies the electronic mail program. The user can restore these windows later and continue interacting with these applications.

User Interface

User interface toolkits offer the programmer higher-level tools than the X library with which to program. As an example, in the X Window System itself there are primitives to draw lines, move rectangles of pixels, and so forth. Toolkits, on the other hand, provide the programmer with easy ways to create and manipulate useful interactive gadgets such as menus, buttons, scroll bars, text entry fields, and so on. These tools greatly reduce the effort that the programmer must put into creating the user interface portion of a program.

To promote acceptance of the X Window System as a standard, HP developed a user interface toolkit based on X, called Xrlib, and contributed it to the MIT public distribution of X Version 10.4 in December of 1986. HP subsequently enhanced this toolkit and ported it to the next revision, X Version 11. It provides a useful set of 13 interactive components called field editors, including pop-up walking menus, panels, scroll bars, title bars, a variety of buttons, and fields for entering, editing, and displaying text or graphical data.

Several user interface toolkits have now been contributed to the MIT public X distribution, offering a wide range of capability. Perhaps the most significant is Xt which is a set of low-level toolkit procedures called *intrinsics*. The Xt intrinsics were developed in a collaborative effort by Digital Equipment Corporation, Hewlett-Packard, Massachusetts Institute of Technology, and others. The intrinsics provide a flexible, powerful foundation upon which to construct interactive components, such as buttons, scroll bars, menus, and other items which are collectively called *widgets*. The X Consortium has voted to accept Xt as part of the X standard. HP is developing a useful set of widgets, based partly on the Xrlib functionality, that has been contributed to the MIT public distribution to promote acceptance of the Xt intrinsics.

Terminal Emulator

The HP X Window System product includes hpterm, a terminal emulator that approximates an HP terminal, complete with softkeys. This allows the HP workstation user to access a broad range of terminal applications that are compatible with this generation of HP terminals.

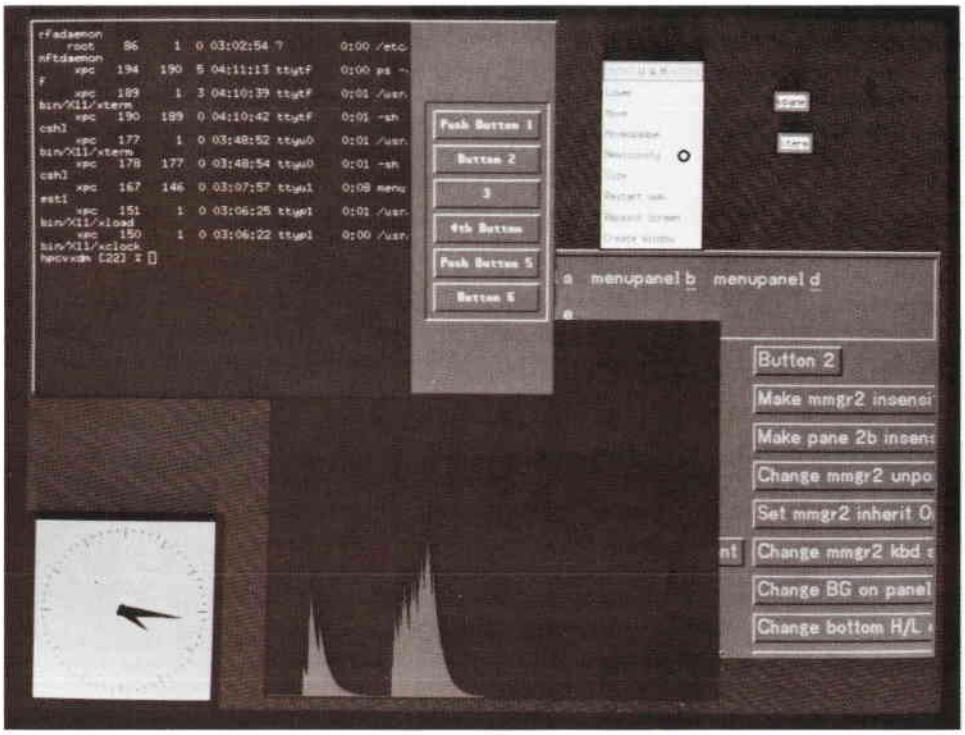


Fig. 3. *Final screen for the scenario presented in the text. The* Launch *menu and the menu containing the* iconify *command are pop-up menus; they are called up by clicking the proper mouse button and go away after a selection is made. The two icons for the terminal emulator and electronic mail applications are in the upper right corner.*

The MIT X distribution contains source code for a Digital Equipment Corporation VT100 and a Tek 4010 terminal emulator called xterm, which is also included in HP's X Window System product. It allows applications written for these older Digital Equipment Corporation and Tektronix Inc. terminals to be accessed directly from the HP workstation display.

The HP workstation user can activate any number of these terminal emulators in any combination, choosing the right one for the application to be accessed.

Support

The 1986 release of X, Version 10.4, was the first version with multivendor support. While this has allowed solution creators to begin development of X Window System solutions, developers soon recognized that to make X a solid standard, there was a need for increased capability and backwards-compatible extensibility of the X protocol to incorporate new functionality that might arise. This protocol extensibility would allow X to improve without breaking applications written earlier. Such enhancements to the design of the protocol, X library, and display server resulted in Version 11 of X, the sample implementation of which MIT formally released in March, 1988.

While the technical innovations in X are quite impressive, what really sets X apart from other window systems is its openness as a standard and its broad base of support.[5,6,7,8] in coordination with MIT, a consortium of companies has been formed to define enhancements and to divide the engineering effort needed to develop standard descriptions and sample implementations of those enhancements. MIT chairs the consortium but the consortium defines the extensions to X. This should ensure the stability and broad support of X for the foreseeable future.

MIT and the X Consortium administer the release of the public-domain X Window System code as well as the contributions of the various supporting vendors. In this way the software and enhancements are available to all interested parties at the same time.

X Consortium membership includes HP, Apollo Computer Inc., Apple Computer Inc., American Telephone and Telegraph Co., Control Data Corp., Digital Equipment Corp., International Business Machines Corp., Sun Microsystems Inc., Tektronix Inc., and Xerox Corp. This list represents a large segment of computer vendors in the technical market. Software vendors formally supporting the X Window standard include Adobe Systems Inc., Applix Inc., and Cognition Inc.

X has been chosen by the X/Open committee, a group that adopts UNIX operating system standards for a consortium of U.S.A. and European computer vendors. Following this lead, the recently formed Open Software Foundation has adopted X as its window system standard. Work is also occurring for formal acceptance of X by other standards groups.

X is the beginning of a new generation of software and systems design that takes a significant step forward in the era of distributed computing environments. A seamless integration of services in these multivendor environments now appears possible, allowing the scaling of computers to their appropriate tasks while maintaining open, productive access to their functions. A standard user interface style, which would allow the easy porting of users between computing systems and between applications, may not be far behind.

References

1. D.S Rosenthal, "Toward a More Focused View," *UNIX Review*, June 1986, pp. 54-63.
2. R.W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, Vol. 5, No.2, 1986, pp. 79-109.
3. R.W. Scheifler, *X Window System Protocol, X Version 11, Release 2*, Massachusetts Institute of Technology, September 1987.
4. J. Gettys, R. Newman and R.W. Scheifler, *Xlib - C Language X Interface, X Version 11, Release 2*, Massachusetts Institute of Technology, September 1987.
5. "11 Companies Back Windowing Standard," *Electronic Engineering Times*, January 19, 1987, pp. 1,16.
6. "The Advantages of X," *Computer Graphics World*, August 1987, pp. 57-60.
7. "DEC, HP, Nine Others Adopt MIT X Window as Standard," *Electronic News*, January 19, 1987, pp.1,6.
8. E. Lee, "Window of Opportunity," *UNIX Review*, June 1988, pp. 47-61.

Authors

October 1988

6 Discless HP-UX

Scott W. Wang

Scott Wang served as project manager for a variety of calculator software projects before he joined HP's UNIX development team. Successively, he was a project manager and R&D section manager involved in HP-UX development for the HP 9000 Series 300. Scott now is R&D manager at the Information Software Division of HP and continues to be responsible for HP-UX software. He came to HP in 1972, when he joined the Calculator Products Division in Loveland, Colorado. His BSEE degree is from the Massachusetts Institute of Technology (1971) and his MSEE degree is from the University of Michigan (1972). Scott is a member of the IEEE. He has contributed two previous articles to the HP Journal. He was born in Taipei, is married and has two children. He is an afficionado of high-fidelity audio, video, and photography.

10 Discless File System

Debra S. Bartlett

Debbie Bartlett was responsible for the HP-UX 6.0 file system, particularly the I/O and FIFO scheme. She has since become a project manager for the file system and discless testing. Debbie attended Purdue University, where she obtained a BS degree in mathematics in 1977. Her MS degree in computer science is from Colorado State University (1982). She was born in Indianapolis, Indiana, is married, and has two small daughters. Debbie's husband is a project manager at HP's Colorado IC Division. She resides in Ft. Collins, Colorado, and enjoys outdoor activities with her family.

Joel D. Tesler

As the lead engineer of the team that conceived and built the original distributed HP-UX environment, Joel Tesler designed most of the initial file system code and the message interface. In a previous project, he designed a softkey package and other environments for the HP 64000-UX system. Joel came to HP in 1980, when he joined the Logic Systems Division. He is the coauthor of a paper on HP-UX operating systems presented at Uniforum 87, and he has previously contributed to the HP Journal. He attended the University of California at Davis, where he received his BS degree in computer science in 1980. Joel was born in Los Angeles and lives in Cupertino, California. His favorite pastime is orienteering, a cross-country race in which contestants navigate through unfamiliar territory using only a compass and a map.

15 Discless Program Execution

Ching-Fa Hwang

Ching Hwang initiated and managed the DUX project at HP Laboratories. He continued to manage kernel and integrated systems at HP's Information Software Division and led their integration into HP-UX products. He coauthored a paper on the subject at Uniforum 87, and a patent application describing the DUX network protocol includes his ideas. Previously, Ching led two projects aimed at developing distributed data bases. Before joining HP Laboratories in 1979, Ching's professional activities included real-time process control, computer architecture and processors, and multiple-processor operating systems. His BSEE degree is from the National Taiwan University (1971), and his MS degree in computer science is from the University of Utah (1974). Ching and his wife, who also works for HP, have two sons and live in Cupertino, California. He is building a koi pond with waterfalls and enjoys playing the piano.

William T. McMahon

The remote swapping scheme for the discless workstations was Bill McMahon's primary project. His previous development assignments include the graphics ROM for the HP 9826 BASIC Release 1.0, and the linker and assembler for Release 2.0 of the HP 9000 Series 200 HP-UX system. He holds a BA degree in philosophy from Ohio University (1971) and an MS degree in computer science from Colorado State University (1979). He came to HP in 1979. Bill is married and has two children. Among his favorite recreational activities are Tai-Chi, cross-country skiing, hiking, and backpacking.

20 Discless Network Functions

David O. Gutierrez

As a member of the team developing the HP-UX 6.0 software, David Gutierrez' responsibilities included the network functions and transport, protocol and buffer management, and configurability. Before joining HP in 1985, he worked for Digital Equipment Corporation, Western Electric Company, and Bell Laboratories. David's main professional interests are the UNIX operating system, special-purpose networking protocols, and distributed operating systems. He attended the University of New Mexico, where he received his BS degree in 1980, and did graduate work in computer engineering at the Illinois Institute of Technology. He has taught a project business class in a middle school and serves as treasurer of the Parent/Child Education Center in Windsor, Colorado, where he lives. He was born in Pueblo, Colorado, is married, and has two daughters. He has designed his own house and enjoys woodworking, golf, skiing, camping, and "anything that doesn't deal with computers."

Chyuan-Shiun Lin

Distributed operating systems, data base systems, and communications are Chyuan-Shiun Lin's focal professional interests. Before working on the distributed HP-UX system, his responsibilities included data base research at HP Laboratories and work on the HP-UX Release 2.0 for the HP 9000 Series 800. Before coming to HP in 1981, he worked in the data processing field. He has published a number of papers on a variety of computer subjects and has contributed to a protocol design for which a patent is pending. Chyuan-Shiun is a member of ACM. His BSEE degree is from the National Taiwan University (1970), and his master's degree in computer science (1976) is from the University of Utah. Born in Taipei, he is married and has three children. He lives in Cupertino, California.

27 Discless Crash Recovery

Annette Randel

Crash detection and recovery, system reboot, self-test, and the file system cnode maps were among Anny Randel's projects for the HP-UX 6.0 system. Previous responsibilities include work on the boot ROMs for the HP 9000 Series 200/300 Computers and the assembler and commands for the Series 200. She first joined HP in a summer-student position in 1981, and two years later joined full-time.

Anny's BS degree in computer science and computer engineering is from Graceland College (1981) and her MS degree in computer science is from Colorado State University (1983). Born in Roseville, California, she is married and lives in Ft. Collins, Colorado. She sings for a pop/jazz group and plays both clarinet and baritone saxophone. Her other hobbies include bicycling, triathlons, running, skiing, softball, and water skiing.

33 Discless Boot Mechanism

John S. Marvin

A software engineer on the HP-UX Release 6.0 project, John Marvin developed a number of commands for discless operation. Before he came to HP in 1984, John worked as a programmer analyst for his alma mater, the University of Virginia. Both his BS and his MS degrees are in computer science (1981 and 1983, respectively). He was born in Brooklyn, New York, is married, and lives in Ft. Collins, Colorado. Among his favorite off-hours activities are skiing and ballroom dancing.

Perry E. Scott

Perry Scott's primary responsibilities for HP-UX Release 6.0 were the secondary loader, discless kernel initialization, discless context for CDF, and system clock synchronization. He had previously worked on Releases 5.0, 5.1, and 5.2. He has been with HP since he earned his bachelor's degree in electrical engineering from North Dakota State University in 1980. Perry has served in the Air National Guard for six years. He was born in Fargo, North Dakota, is married, and lives in Ft. Collins, Colorado. Gardening and bicycling are among his favorite leisure activities.

Robert D. Quist

Among the HP projects Robert Quist has worked on since he joined HP in 1971 are a Lisp workstation, third-party support, a Pascal workstation and a boot ROM for the HP 9000 Series 200 system. To the HP-UX Release 6.0 system he contributed boot ROM revisions B and C. Specialty boot ROMs, Pascal workstations, low-level drivers, and human interfaces are Robert's special interests. His BE degree in computer science is from Brigham Young University (1971). Born in Lethbridge, Alberta, Robert is married and has eight children. He lives with his family in Loveland, Colorado. He is active in the Cub Scouts and teaches Pascal programming. Robert enjoys birdwatching, camping, and reading science fiction.

37 Discless System Configuration

Kimberly S. Wagner

Kim Wagner's responsibilities on the HP-UX Release 6.0 project included the discless administration tools and system software integration. She came to HP in 1986. She holds a BS degree in computer science and mathematics from the University of California at Davis (1983) and an MS degree from Colorado State University at Ft. Collins (1986). Kim is a member of ACM and SIGGRAPH. She was born in Redwood City, California, and now lives in Ft. Collins, Colorado.

39 SCSI

Paul Q. Perlmutter

Paul Perlmutter's responsibilities in the HP-UX Release 6.0 project included the mass storage software, the driver support, and backup strategies. High-performance mass storage devices for UNIX systems were the main focus of the positions Paul held before joining HP in 1985. His PhD and MS degrees in mathematics are from the University of Colorado (1975 and 1971). He has held a position as a college professor of mathematics and has published several articles on the subject. Paul serves as president in his synagogue in Ft. Collins, Colorado, where he lives. He was born in New York and has two daughters. His favorite pastimes are bicycling, photography, and hiking, but he spends most of his spare time with his children.

46 X Window System

Frank E. Hall

Frank Hall is a project manager for user interface productivity tools and has held a similar position in the development of the Xrlib and HP X Widget user interface libraries for HP-UX. Before coming to HP in 1979, he worked as a system analyst for the Computer Sciences Corporation, where he helped develop a worldwide computer network for communications with the space shuttle and geosynchronous satellites. At HP, Frank has worked as a software engineer on operating system firmware for the HP 71B Handheld Computer and application software for the HP Portable PLUS laptop computer. He has published three articles in the proceedings of the HP Software Engineering Productivity Conference. Frank holds a BA degree in mathematics from Florida State University (1972) and an MA degree in anthropology from the University of Texas at Austin (1979). He is a member of the ACM. He was born in Ft. Myers, Florida, is married, and lives in Corvallis, Oregon. His leisure interests include bicycle touring, birdwatching, folk music, skiing, white water rafting, and fishing.

James B. Byers

With marketing user interface technology his focal interest, Jim Byers is the product marketing engineer involved with HP's release of the X Window System Version 11. He joined the Indianapolis sales office of HP in 1982 and has served as the marketing representative for the HP 9000 and HP 1000 Computers. Jim's BSEE degree is from Purdue University (1980) and his MBA degree in marketing is from Indiana University (1982). He was born in South Bend, Indiana. He's married, has a small daughter, and lives in Corvallis, Oregon. In his off-hours, he likes skiing, camping, and exploring the outdoors with his family.

51 Managing DeskJet Development

John D. Rhodes

When he joined the Microwave Division of HP in 1966, John Rhodes had just received his MBA degree from Stanford University. He also holds a bachelor's degree in industrial technology from Long Beach State College (1964). In his varied career at HP, John served in many different engineering and managerial positions. As project manager of the mechanical team working on the DeskJet project, he originated several patents. John was born in San Jose, California. He is married, has a daughter and a son, and lives in Vancouver, Washington. For seven years, he served on the board of directors for a puppet theater which tours extensively throughout the United States. His hobbies include astronomy, photography, and piloting light aircraft.

55 High-Resolution Printhead

Kenneth E. Trueba

The printhead firing chamber, the ink feed process, and a method for bubble observation were among Ken Trueba's design assignments for the DeskJet pen. Earlier design work included the thermal printheads for the HP 2621P Terminal, the HP 2671 Thermal Printer, and the HP 85A/B Personal Computers. On the DeskJet development team, Ken was responsible for process, assembly, and architectural design of the printhead. He came to HP after receiving his BS (1974) and MSEE (1976) degrees from the South Dakota School of Mines. Ken has presented papers and published a journal article on the subjects of thin-film techniques and thermal inkjet devices. Three patents based on his designs are pending. Ken was born in Boise, Idaho, is married, and lives with his wife and two daughters in Corvallis, Oregon. He enjoys photography, skiing, playing the guitar, and psychology.

Richard R. Van de Poll

A process engineer in the team developing the DeskJet printer, Rich Van de Poll worked on printhead-related assignments. He is now a project leader at the Inkjet Components Operation. His BS degree in chemical engineering is from the University of Colorado. Before he joined HP's Logic Systems Division in 1986, he had been a process engineer at Eastman Kodak, where he participated in developing photographic emulsions. Born in Surabaja, Indonesia, Rich served three years in the U.S. Army. He is married, has two children, and serves as a cubmaster in the Boy Scouts. He lives in North Albany, Oregon, and spends his leisure time with photography and scuba diving.

Paula H. Kanarek

A statistician and R&D project manager for the DeskJet product, Paula Kanarek came to HP in 1982. Her bachelor's degree is from the University of Michigan (1967), and her ScM (1969) and ScD (1973) degrees in biostatistics are from Harvard University. Before joining HP, Paula held positions as an assistant professor in statistics and biostatistics, respectively, at the University of Washington and at Oregon State University. Statistical applications and reliability in engineering are her focal interests and provide the subjects of some 20 of her publications. She is a member of the American Statistical Association, the American Society of Quality Control, and the Society of Women Engineers. Paula is active in local school advisory committees at Salem, Oregon, where she lives. She is married and has two children. For recreation, she likes hiking, bicycling, and cross-country skiing.

Robert N. Low

Starting as a temporary hire during summer recess, Bob joined HP permanently in 1977, when he received his BS degree in metallurgical engineering from Purdue University. His first assignment involved the hybrid development for the HP-41C Calculator. He went on to other projects, participated in development of the ThinkJet printer, and eventually became project manager for pen assembly and manufacturing technology for the DeskJet printer. Bob's work has resulted in four patents for DeskJet-related devices. His professional interests focus on high-precision, high-volume manufacturing technology, and he has previously contributed to the HP Journal. He was born in South Bend, Indiana. Bob is married, has two small daughters, and lives in Corvallis, Oregon. His leisure interests include sailing, scuba diving, skiing, hiking, and sports cars.

William A. Buskirk

Bill Buskirk came to HP in 1977 with a BSEE degree from the University of Colorado at Boulder. Before he became a project manager for the DeskJet printhead, he handled a variety of assignments as a production engineer and R&D engineer, including work on the HP 82161A Digital Tape Drive. The DeskJet development provided the subject of two papers Bill published in conference proceedings. He recently was appointed R&D section manager at the Inkjet Components Operation. Bill was born in Bloomington, Indiana, is married, and now lives in Albany, Oregon. He has three young children. His favorite recreational activities are hiking, windsurfing, alpine skiing, and sailing.

Stanley T. Hall

After joining HP in 1976 at the Corvallis Component Operation, Stan Hall spent over five years as a manufacturing engineer working on tooling for HP 33, 10, and 75 Series calculators and an HP-IL printer. He moved to InkJet development in 1983 and became a project manager in production engineering. Stan's BS degree in industrial technology is from California Polytechnic University. His previous professional experience includes positions as a supplier quality engineer for ISS-SperryUnivac and Intel. Stan is a member of the Society of Manufacturing Engineers. He was born in Oakland, California, and now resides in Corvallis, Oregon. He is married and has two daughters. His leisure activities include woodworking in the winter, sailing and gardening in the summer.

David E. Hackleman

As R&D project manager at HP's Inkjet Components Operation, the inks and media used in ThinkJet, PaintJet, and DeskJet printers have been the focus of David Hackleman's interests in recent years. He came to HP with a BSEE degree from Oregon State University (1979) and a PhD in analytical electrochemistry from the University of North Carolina at Chapel Hill (1978). His previous design work at HP includes a variety of integrated circuit processes at the InkJet Component Operation. His work in IC processing, thermal inkjet inks, and multiplexing has resulted in six patents, with several more in application. David is a member of the American Chemical Society and the Electrochemical Society. He is a National Youth Science Camp lecturer and serves as a control station operator of the amateur radio emergency service. David was born in Coos Bay, Oregon, is married, and lives in Monmoth, Oregon, where in his off-hours he operates a tree farm "on forty acres way out of town."

62 — Printer/Printhead Integration

John A. Widder

Author's biography appears elsewhere in this section.

J. Paul Harmon

As a development engineer on the DeskJet project, Paul Harmon was responsible for carriage, service station, and interconnect design. A patent is pending for the DeskJet interconnect support structure Paul developed. He came to HP after receiving his BMSE degree from the University of Washington in 1981 and has since earned his MSME degree from Stanford University (1988). Paul has previously coauthored an article for the HP Journal (May 1987). Born in Hermiston, Oregon, Paul is married and has a child. He now lives in Washougal, Washington. Paul's spare time is taken up by motorcycles, sports cars, and church activities.

67 — Chassis and Mechanism Design

David W. Pinkernell

Mechanical design of the servo control for paper feed and carriage drive was Dave Pinkernell's focal contribution to the DeskJet development. With the project since he joined HP in 1981, he worked as a design engineer until the conclusion of the design phase, when he joined a team of manufacturing engineers in the task of setting up production facilities. He is coinventor of a pending patent for the printer mechanism. Dave attended the California Polytechnic State University at San Luis Obispo, where he received his BSME degree in 1981. His MSME degree is from Stanford University. He was born in Santa Barbara, California, and makes his home in Pullman, Washington. His recreational interests include photography and traveling, both of which he recently combined in a trip to East Africa.

John A. Widder

The design of the carriage servo system, linefeed motor control, and printhead driver printed circuit boards are among John Widder's contributions to the DeskJet printer. As a development engineer at the Vancouver Division, he also worked with the suppliers of the power supply. Now a manufacturing engineer, John has shifted his attention to the

DeskJet production line. In the past, he has worked on the design of thermal printers such as the HP 2675A, HP 2671/3A, and HP 2674A. John came to HP's Boise Division in 1978, after receiving his BSEE degree from the University of Portland. He is a member of the IEEE and the American Association for the Advancement of Science. He's also active in the City Club of Portland and the World Affairs Council of Oregon. John was born in Bethesda, Maryland, and now makes his home in Brush Prairie, Washington. His favorite pastimes include backpacking and cross-country skiing.

Kieran B. Kelly

The DeskJet printer was not the first product to use a paper path design by Kieran Kelly. With mechanical design his main professional interest, he has previously worked on the QuietJet printer, the tilt/swivel base for the HP 150 Computer, and the HP 9826 and HP 9836 Controller/Computers. His work on the QuietJet paper drive resulted in a patent application. Kieran joined HP in 1979, the year he received his BS degree from the University of Virginia. He also holds an MS degree from Stanford University (1984). He lives in Vancouver, Washington, where he is presently renovating a turn-of-the-century Victorian house. Other hobbies include sailing, skiing, bicycling, and hiking.

Steve O. Rasmussen

The paper path, carriage, and transmission of the DeskJet printer were the focal points of Steve Rasmussen's design work. He has contributed to designs for the DeskJet printer that resulted in a patent and five patent applications. He came to the Vancouver Division in 1982, after receiving his BSME degree from Iowa State University. Steve also holds an MSME degree from Stanford University (1987). He was born in Fort Dodge, Iowa. He is married, has an infant son, and lives in Vancouver, Washington. Steve likes to spend his off-hours with church activities, working on his house, bicycling, and woodworking.

Larry A. Jackson

Larry Jackson coordinated the design of DeskJet paper handling parts, such as the chassis, carriage guide, pinch rollers, pinch springs, and gear trains. An R&D engineer at the HP Vancouver Division, his past product involvement includes the ThinkJet printer, the HP-01 Watch, the HP-41C Calculator, an IC tester, a laser interferometer, and a multichannel analyzer. He has managed the hybrid laboratory at Santa Clara Division. A patent and four patent applications are based on Larry's designs. He attended Utah State University from which he received BS (1965) and MS (1966) degrees in mechanical engineering. He was born in Ogden, Utah, is married and has four children. He resides in Vancouver, Washington. Larry's spare time interests include camping, boardsailing, downhill skiing, and church activities.

76 Data to Dots

Donna J. May

As a development engineer at the Vancouver Division, Donna May worked on the firmware for the DeskJet printer. On other assignments, she has worked on the HP 2934A Business Printer and a landscape upgrade cartridge for the DeskJet printer. Donna came to the Vancouver Division in 1983, after receiving her BS degree in computer engineering from Iowa State University. Born in Cedar Rapids, Iowa, she is married and resides in Vancouver, Washington. She plays piano and bassoon and likes backpacking and bicycling.

Claude W. Nichols

After joining HP in 1980 as a development engineer, Claude Nichols worked on a variety of printers, including the HP 2675A, the HP 2671A/G, the HP 2674A, and the HP 2932A. In the design of the DeskJet printer, he was involved in various aspects of the firmware. Claude's BS degree in computer science is from Brigham Young University, earned in 1979. He was born in Reno, Nevada, but grew up in Cheney, Washington. He is married and has three children. Claude is active in the Boy Scouts and in his church in Vancouver, Washington, where he lives. He enjoys cross-country skiing, bicycling, and hiking.

Mark D. Lund

One of the patents pending for the DeskJet printer resulted from Mark Lund's design work. An R&D engineer at the Vancouver Division, he joined HP in 1977 after receiving his BSEE degree from the University of California at Irvine. Among the projects Mark has worked on are the electronics architectures of the HP 2621A Terminal and the HP 2671A and HP 2673A Thermal Printers. He was born in Santa Monica, California, and now lives in Vancouver, Washington. He is married and has a daughter and triplet sons. He enjoys camping, backpacking, scuba diving, and woodworking. He also likes fishing, especially salmon and steelhead.

Thomas B. Pritchard

As a development engineer for the DeskJet printer, Tom Pritchard's responsibilities included design of a custom IC and testing and correction of electrostatic discharge conditions. His past assignments include both firmware and electronics design on HP 2934A, HP 2932A, and HP 2671A Printers and work as a production engineer for the HP 3000 Business Computer System. Tom is the primary author of an article describing a microprocessor-based signal processing system for biomedical measurements and also has previously contributed to the HP Journal. A patent is pending for a character generator he developed. Born in Ann Abor, Michigan, Tom is married and has a three-year-old child. He now lives in Vancouver, Washington. He enjoys hiking and playing tennis.

81 DeskJet Firmware

Mark J. DiVittorio

Mark DiVittorio is a project manager at the Vancouver Division, where his responsibilities included development of the DeskJet firmware. In the past, he has served as development engineer, production engineer, and R&D engineer. His EE and MS degrees in computer science (1974 and 1978, respectively) are from the University of Santa Clara. Born in Chicago, Illinois, Mark is married, has a seven-year-old son, and lives in Vancouver, Washington. In his off-hours, he likes to go fishing.

Claude W. Nichols

Author's biography appears elsewhere in this section.

Michael S. Ard

As a project manager at the Vancouver Division, Mike Ard directed the development of the Epson FX-80 printer emulation firmware for the DeskJet. His past responsibilities as a development engineer include work on the HP 300 Business Computer System and the HP 2675A, HP 2673A, and HP 2934A printer systems. He also managed the printer systems group, focusing on printer solutions to system and application support. He holds a BS degree (1975) and an MS degree (1978) in computer science, both from Brigham Young University. Mike is active in his church and in youth sports. Born in St. Anthony, Idaho, he is married and has six children. He lives in Vancouver, Washington. Mike enjoys outdoor activities and has been busy designing, building, and landscaping his new home.

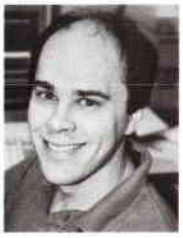
Kevin R. Hudson

Development of the Epson FX-80 printer emulation firmware, specifically the graphics and parser, was Hud Hudson's focal interest on the DeskJet project. In previous years, he has worked on printhead and character set development for the ThinkJet printer and hardware for the QuietJet. Hud earned his BS degree at Iowa State University in 1981 and shortly thereafter joined HP, where he now is a development engineer at the Vancouver Division. He was born in Vinton, Iowa, is married, and lives in Vancouver, Washington. His recreational interests include softball, golf, and science fiction.

Brian Cripe

Development of the DeskJet formatter was among Brian Cripe's most recent projects. Presently, he is working on the X Window System at the Corvallis Workstation Operation, and past assignments include the mechanism controller code for the ThinkJet printer. Brian has originated a text scaling system for which a patent is pending. His BSCE and BACS degrees are from Rice University (1982). Brian was born in Anapolis, Brazil, is married, and lives in Corvallis, Oregon. His favorite pastimes are bicycling, telemark skiing, and tending his prize-winning roses.

David J. Neff



David Neff's responsibilities for the DeskJet printer included development of the Epson FX-80 emulation firmware. In the past, he has worked on the RTE-L and RTE-XL operating systems and landscape cartridge firmware. He also developed CAD/CAM software links used internally in HP's Vancouver Division. David attended Harvey Mudd College, where in 1979 he earned his BS degree in mathematics. He was born in Portland, Oregon, is married, and has two children. He now resides in Vancouver, Washington.

87 Robotic Assembly

P. David Gast

As a manufacturing engineer at the Vancouver Division, Dave Gast developed an automated high-volume assembly line for mixed-mode production of DeskJet and Rugged-Writer 480 printed circuit boards. He also designed the mechanical hardware for the robotic workcell that builds the boards. In a previous position, Dave worked for the Research Center of Weyerhaeuser Company. He holds a BSME degree from Texas A&M University (1982) and an MBA degree from Oregon State University (1984). He was born in Minneapolis, Minnesota, and now lives in Vancouver, Washington. Windsurfing, bicycling, telemark skiing, and photography are Dave's favorite leisure activities.

91 CIM and Machine Vision

Robert F. Aman

As a production engineer and later as a procurement engineer, Bob Aman has shared responsibility for production, procurement, and materials selection for the HP-85 Series personal computers and other portable computers. More recently, at HP's Inkjet Components Operation, he served as a project leader for design and fabrication of the equipment used for wafer assembly and its integration in the manufacturing process of the DeskJet pen. Bob came to HP in 1980, after working for some three years as a manufacturing engineer at Boeing Commercial Airplane Company. He earned a BSME degree from Oregon State University in 1977 and was awarded a professional engineering license in 1981. Bob was born in Silverton, Oregon. He is married, has two sons, and lives in Albany, Oregon. His hobbies include radio-controlled model airplanes, canoeing, fishing, and bow hunting.

Brian L. Helterline

Just after receiving his BSEE degree from Montana State University in 1987, Brian Helterline joined the Inkjet Components Operation of HP. As a product engineer, he was responsible for print quality testing of DeskJet print cartridges, ownership considerations for the print quality tester, and machine vision algorithms used for testing. Vision software and computer-control applications are his main professional interests. Born in Plains, Montana, Brian now makes his home in Salem, Oregon. He is married and enjoys playing basketball and tennis.

Gregg P. Ferry

An engineer at HP's Inkjet Components Operation, Gregg Ferry participated in the design of vision applications for the DeskJet printer. The project continues to be central to his design activities as he concentrates on electronic tools and computer-integrated manufacturing for the product. Both his BSEE (1973) and his master's (1976) degrees are from the California Polytechnic Institute at San Luis Obispo. Gregg was born in Minneapolis, Minnesota, and now lives in Corvallis, Oregon. He serves as a volunteer instructor for the Saturday Academy, an organization offering extracurricular instruction for high school students. Gregg likes traveling and bicycling, two avocations he once combined in a two-year biycycle trip around the world.

Timothy S. Hubley

A vision applications engineer and project leader at the HP Inkjet Components Operation, Tim Hubley has focused on evaluation of DeskJet print quality. In previous projects, he has worked as a product engineer on the chips for the HP 71B Handheld Computer and as an electrical tooling engineer for test equipment. He earned BS and ECE degrees from the University of Massachusetts in 1981 and joined HP the same year. He is a member of the Society of Mechanical Engineers and the Machine Vision Association. Tim was born in St. Charles, Illinois. He is married, has two children, and lives in Corvallis, Oregon. Among his favorite activities, Tim lists volleyball, tennis, and fatherhood.

Mark C. Huth

In the over seven years since he joined HP, Mark Huth has worked on manufacturing development and tool design for the HP 85 Computer, the HP 75C and HP 71B Handheld Computers, and inkjet print cartridges. On the DeskJet team, Mark helped design the optics system and develop the machine vision software. He received his BS degree in mechanical engineering from Virginia Tech in 1981. He was born in Boston, Massachusetts, and now lives with his wife and two sons in Corvallis, Oregon. Bicycling, volleyball, and rock climbing are Mark's favorite sporting activities, but he also likes potluck parties and enjoys the "small-town atmosphere of Corvallis."

Robert A. Conder

Vision applications and computer-integrated manufacturing are Bob Conder's focal professional interests. Since he joined HP in 1975, his product involvements have included a wide variety of HP calculators, handheld computers, and the ThinkJet and DeskJet printers. Bob is now a project manager and has been responsible for the coordination of vision projects and control systems associated with the DeskJet product. He is the author of an article about inkjet cartridges, and three patents are

pending on optics and alignment procedures he developed. Bob's BSEE degree is from the University of Utah (1975). He was born in Salt Lake City, Utah, and served four years as a sergeant in the U.S. Air Force. He is married, has a son and a daughter and lives in Corvallis, Oregon. His recreational interests include dirt-bike riding, sailing, and fishing.

99 Optical Encoders

Robert Nicol

As a manufacturing engineer, Rob Nicol developed a number of production methods for the HEDS-9000 encoder. Prior to that, he contributed to the development of the HEDS-9000 as an R&D engineer. He is now working on new optical-encoder applications. Before joining HP as an R&D engineer in 1983, Rob worked for the Santa Barbara Research Center, a subsidiary of GM/Hughes. His BSME degree is from the University of California at Santa Barbara (1983). He was born in Walnut Creek, California, is married and has two sons. He resides in Fremont, California. Rob is a member of the Society for Automotive Engineers and, in his spare time, has converted a conventional automobile to electric operation. He also enjoys serious music, sailing, and making beer.

Mark G. Leonard

Mark Leonard was one of the designers of the original HP encoder, and his collaboration with other development engineers led to product definition and design of the HEDS-9000. His professional interests are interdisciplinary and encompass electronics, computer science, and reliability physics. In the past, Mark has been involved in the design of a fiber optic receiver and high-voltage optocouplers. Before coming to HP in 1975, his responsibilities included failure analysis and computer programming. He is a senior member of the IEEE and a registered professional engineer and has published articles about optical encoders. Four U.S. patents are based on his designs. Mark attended Macalester College at St. Paul, Minnesota, where he received his BA degree in 1965. He was born in San Jose, Costa Rica, is married, and has three sons. He lives in Los Altos, California. His hobbies include woodworking and photography.

Howard C. Epstein

Since coming to HP in 1976, Howard Epstein has been responsible for the development of shaft encoder technologies. On the HEDS-9000 encoder, he led the definition and prototype stages and was the architect of the emitter/detector/lens system. Howard has since concentrated on extensions of the HEDS-9000 product. Before joining HP, he developed piezoelectric and piezoresistive transducers. Sensors and transducers are the focus of his professional interests, and he has published four papers about electromechanical and optical sensors. He is a named inventor on five patents. A registered professional engineer, Howard holds BA and MS degrees in physics (1965 and 1975) from the California State University at Los Angeles. Born in Los Angeles, he is married, has three teenage daughters, and lives in Los Altos, California. Howard serves on the board of the American Association for Ethiopian Jewry, an organization dedicated to the rescue of the ancient Ethiopian Jewish community. He enjoys playing handball, roller-skating, and surf fishing.

Hewlett-Packard Company, 3200 Hillview Avenue, Palo Alto, California 94304

HEWLETT-PACKARD JOURNAL

October 1988 Volume 39 • Number 5

Technical Information from the Laboratories of Hewlett-Packard Company

Hewlett-Packard Company, 3200 Hillview Avenue
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Central Mailing Department
P.O. Box 529, Startbaan 16
1180 AM Amstelveen, The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

Bulk Rate
U.S. Postage
Paid
Hewlett-Packard
Company

CHANGE OF ADDRESS: To subscribe, change your address, or delete your name from our mailing list, send your request to Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304 U.S.A. Include your old address label, if any. Allow 60 days.

5953-8571