Design of the TruCluster Multicomputer System for the Digital UNIX Environment

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

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

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

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

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

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

Brief Comparison of Computing Systems

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

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

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

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

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

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

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

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

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

Structure of the TruCluster System

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

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

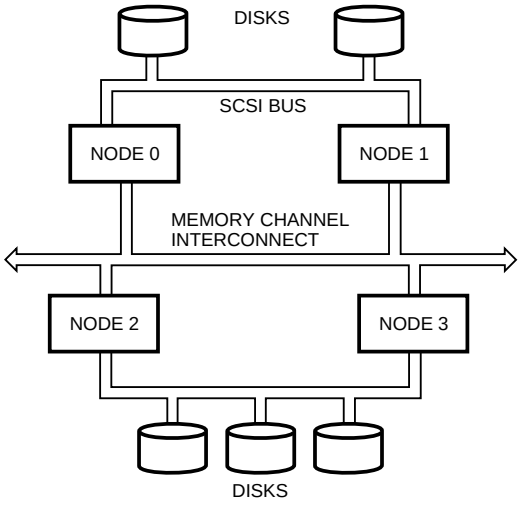


Figure 1
Configuration of a Four-node Cluster System

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

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

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

Distributed Remote Disk Subsystem

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

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

Cluster Storage Background

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

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

Overall Design of the DRD

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

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

Clusterwide Disk Access Model

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

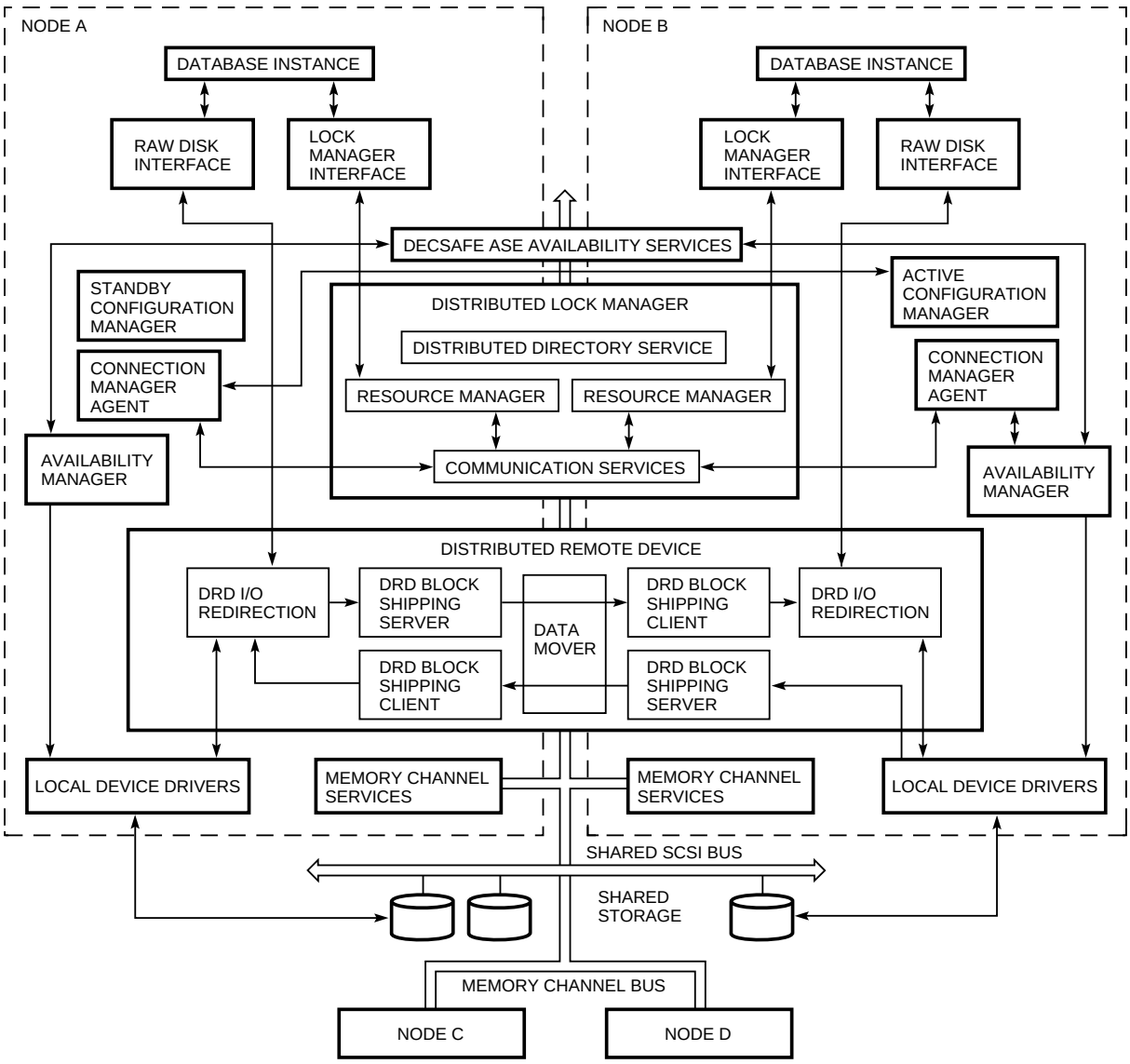


Figure 2
Software Components

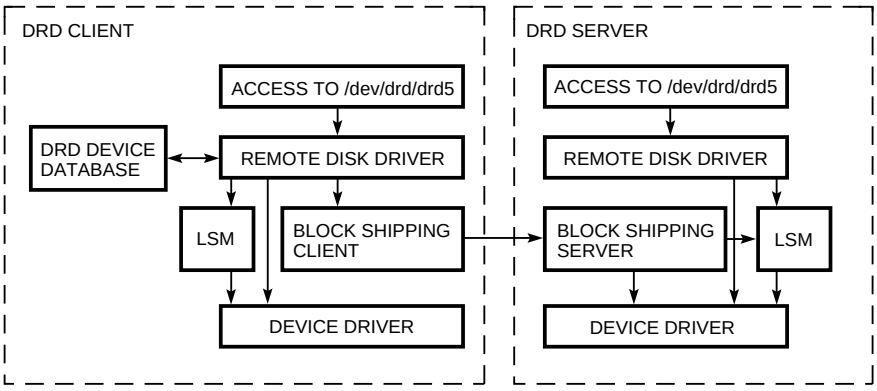


Figure 3
Distributed Remote Disk Subsystem

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

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

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

DRD Naming

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

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

DRD Relocation and Failover

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

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

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

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

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

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

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

Distributed Lock Manager

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

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

Usage of the DLM

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

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

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

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

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

Design Goals of the DLM

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

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

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

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

DLM Communication Services

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

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

Connection Manager

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

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

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

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

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

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

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

Memory Channel Interconnect

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

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

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

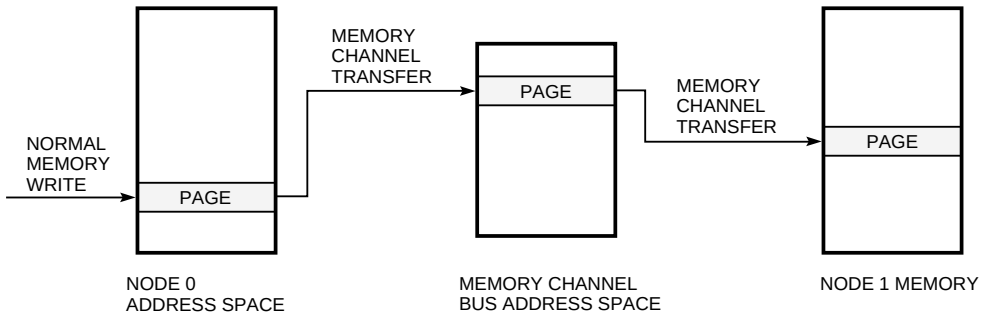


Figure 4
Transfers Performed by the MEMORY CHANNEL Interconnect

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

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

Low-level MEMORY CHANNEL Software

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

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

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

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

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

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

Synchronization

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

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

Error Recovery and Node Failures

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

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

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

Future Enhancements to MEMORY CHANNEL Software

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

Application Failover

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

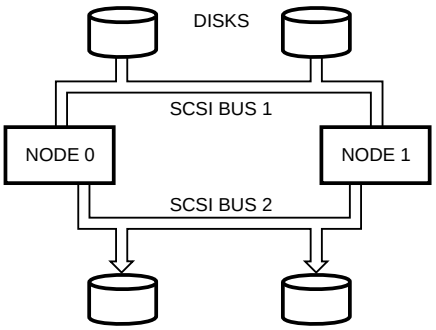


Figure 5
Typical ASE Configuration

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

Hardware Configurations

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

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

Conclusions

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

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

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

Future Developments

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

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

Acknowledgments

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

References and Notes

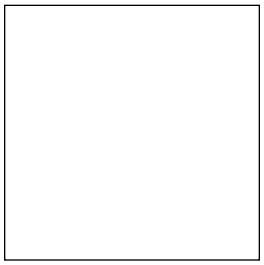
1. "Introduction to DCE," *OSF DCE Documentation Set* (Cambridge, Mass.: Open Software Foundation, 1991).
2. Internet RFCs 1014, 1057, and 1094 describe ONC XDR, RPC, and NFS protocols, respectively.
3. G. Pfister, *In Search of Clusters* (Upper Saddle River, N.J.: Prentice-Hall, Inc., 1995): 19–26.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

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

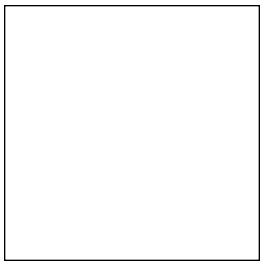
17. T. Rengarajan, P. Spiro, and W. Wright, "High Availability Mechanisms of VAX DBMS Software," *Digital Technical Journal,* vol. 1, no. 8 (February 1989): 88–98.

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

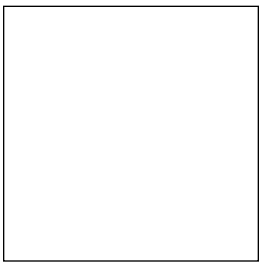
Biographies



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



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



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