The software architecture of a SAN storage control system

by J. S. Glider
C. F. Fuente
W. J. Scales

We describe an architecture of an enterprise-level storage control system that addresses the issues of storage management for storage area network (SAN) -attached block devices in a heterogeneous open systems environment. The storage control system, also referred to as the "storage virtualization engine," is built on a cluster of Linux®-based servers, which provides redundancy, modularity, and scalability. We discuss the software architecture of the storage control system and describe its major components: the cluster operating environment, the distributed I/O facilities, the buffer management component, and the hierarchical object pools for managing memory resources. We also describe some preliminary results that indicate the system will achieve its goals of improving the utilization of storage resources, providing a platform for advanced storage functions, using off-the-shelf hardware components and a standard operating system, and facilitating upgrades to new generations of hardware, different hardware platforms, and new storage functions.

Storage controllers have traditionally enabled mainframe computers to access disk drives and other storage devices.[1] To support expensive enterprise-level mainframes built for high performance and reliability, storage controllers were designed to move data in and out of mainframe memory as quickly as possible, with as little impact on mainframe resources as possible. Consequently, storage controllers were carefully crafted from custom-designed processing and communication components, and optimized to match the performance and reliability requirements of the mainframe.

In recent years, several trends in the information technology used in large commercial enterprises have affected the requirements that are placed on storage controllers. UNIX** and Windows** servers have gained significant market share in the enterprise. The requirements placed on storage controllers in a UNIX or Windows environment are less exacting in terms of response time. In addition, UNIX and Windows systems require fewer protocols and connectivity options. Enterprise systems have evolved from a single operating system environment to a heterogeneous open systems environment in which multiple operating systems must connect to storage devices from multiple vendors.

Storage area networks (SANs) have gained wide acceptance. Interoperability issues between components from different vendors connected by a SAN fabric have received attention and have mostly been resolved, but the problem of managing the data stored on a variety of devices from different vendors is still a major challenge to the industry. At the same time, various components for building storage systems have become commoditized and are available as inexpensive off-the-shelf items: high-performance processors (Pentium**-based or similar), commu-

0018-8670/03/$5.00 © 2003 IBM

nication components such as Fibre Channel switches and adapters, and RAID[2] (redundant array of independent disks) controllers.

In 1996, IBM embarked on a program that eventually led to the IBM TotalStorage* Enterprise Storage Server* (ESS). The ESS core components include such standard components as the PowerPC*-based pSeries* platform running AIX* (a UNIX operating system built by IBM) and the RAID adapter. ESS also includes custom-designed components such as nonvolatile memory, adapters for host connectivity through SCSI (Small Computer System Interface) buses, and adapters for Fibre Channel, ESCON* (Enterprise Systems Connection) and FICON* (Fiber Connection) fabrics. An ESS provides high-end storage control features such as very large unified caches, support for zSeries* FICON and ESCON attachment as well as open systems SCSI attachment, high availability through the use of RAID-5 arrays, failover pairs of access paths and fault-tolerant power supplies, and advanced storage functions such as point-in-time copy and peer-to-peer remote copy. An ESS controller, containing two access paths to data, can have varying amounts of back-end storage, front-end connections to hosts, and disk cache, thereby achieving a degree of scalability.

A project to build an enterprise-level storage control system, also referred to as a "storage virtualization engine," was initiated at the IBM Almaden Research Center in the second half of 1999. One of its goals was to build such a system almost exclusively from off-the-shelf standard parts. As any enterprise-level storage control system, it had to deliver high performance and availability, comparable to the highly optimized storage controllers of previous generations. It also had to address a major challenge for the heterogeneous open systems environment, namely to reduce the complexity of managing storage on block devices. The importance of dealing with the complexity of managing storage networks is brought to light by the total-cost-of-ownership (TCO) metric applied to storage networks. A Gartner report[4] indicates that the storage acquisition costs are only about 20 percent of the TCO. Most of the remaining costs are related, in one way or another, to managing the storage system.

Thus, the SAN storage control project targets one area of complexity through block aggregation, also known as block virtualization.[5] Block virtualization is an organizational approach to the SAN in which storage on the SAN is managed by aggregating it into a common pool, and by allocating storage to hosts from that common pool. Its chief benefits are efficient and flexible usage of storage capacity, centralized (and simplified) storage management, as well as providing a platform for advanced storage functions.

A Pentium-based server was chosen for the processing platform, in preference to a UNIX server, because of lower cost. However, the bandwidth and memory of a typical Pentium-based server are significantly lower than those of a typical UNIX server. Therefore, instead of a monolithic architecture of two nodes (for high availability) where each node has very high bandwidth and memory, the design is based on a cluster of lower-performance Pentium-based servers, an arrangement that also offers high availability (the cluster has at least two nodes).

The idea of building a storage control system based on a scalable cluster of such servers is a compelling one. A storage control system consisting only of a pair of servers would be comparable in its utility to a midrange storage controller. However, a scalable cluster of servers could not only support a wide range of configurations, but also enable the managing of all these configurations in almost the same way. The value of a scalable storage control system would be much more than simply building a storage controller with less cost and effort. It would drastically simplify the storage management of the enterprise storage by providing a single point of management, aggregated storage pools in which storage can easily be allocated to different hosts, scalability in growing the system by adding storage or storage control nodes, and a platform for implementing advanced functions such as fast-write cache, point-in-time copy, transparent data migration, and remote copy.

In contrast, current enterprise data centers are often organized as many islands, each island containing its own application servers and storage, where free space from one island cannot be used in another island. Compare this with a common storage pool from which all requests for storage, from various hosts, are allocated. Storage management tasks—such as allocation of storage to hosts, scheduling remote copies, point-in-time copies and backups, commissioning and decommissioning storage—are simplified when using a single set of tools and when all storage resources are pooled together.

The design of the virtualization engine follows an "in-band" approach, which means that all I/O re-

quests, as well as all management and configuration requests, are sent to it and are serviced by it. This approach migrates intelligence from individual devices to the network, and its first implementation is appliance-based (which means that the virtualization software runs on stand-alone units), although other variations, such as incorporating the virtualization application into a storage network switch, are possible.

There have been other efforts in the industry to build scalable virtualized storage. The Petal research project from Digital Equipment Corporation[6] and the DSM** product from LeftHand Networks, Inc.[7] both incorporate clusters of storage servers, each server privately attached to its own back-end storage. Our virtualization engine prototype differs from these designs in that the back-end storage is shared by all the servers in the cluster. VERITAS Software Corporation[8] markets Foundation Suite**, a clustered volume manager that provides virtualization and storage management. This design has the virtualization application running on hosts, thus requiring that the software be installed on all hosts and that all hosts run the same operating system. Compaq (now part of Hewlett-Packard Company) uses the Versastor** technology,[9] which provides a virtualization solution based on an out-of-band manager appliance controlling multiple in-band virtualization agents running on specialized Fibre Channel host bus adapters or other processing elements in the data path. This more complex structure amounts to a two-level hierarchical architecture in which a single manager appliance controls a set of slave host-resident or switch-resident agents.

The rest of the paper is structured as follows. In the next section, we present an overview of the virtualization engine, which includes the hardware configuration, a discussion of the main challenges facing the software designers, and an overview of the software architecture. In the following four sections we describe the major software infrastructure components: the cluster operating environment, the distributed I/O facilities, the buffer management component, and the hierarchical object pools. In the last section we describe the experience gained from implementing the virtualization engine and present our conclusions.

Overview of the virtualization engine

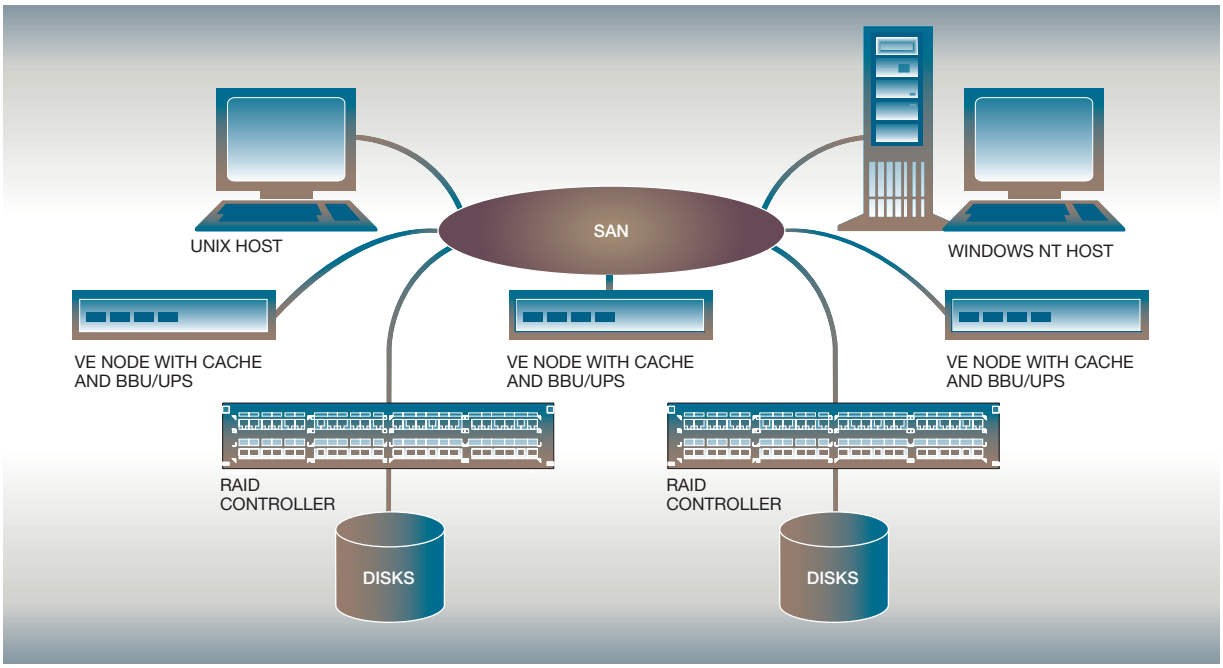
The virtualization engine is intended for customers requiring high performance and continuous availability in heterogeneous open systems environments. The key characteristics of this system are: (A) a single pool of storage resources, (B) block I/O service for logical disks created from the storage pool, (C) support for advanced functions on the logical disks such as fast-write cache, copy services, and quality-of-service metering and reporting, (D) the ability to flexibly expand the bandwidth, I/O performance, and capacity of the system, (E) support for any SAN fabric, and (F) use of off-the-shelf RAID controllers for back-end storage.

The virtualization engine is built on a standard Pentium-based processor planar running Linux**, with standard host bus adapters for interfacing to the SAN. Linux, a widely used, stable, and trusted operating system, provides an easy-to-learn programming environment that helps reduce the time-to-market for new features. The design allows migration from one SAN fabric to another simply by plugging in new off-the-shelf adapters. The design also allows the Pentium-based server to be upgraded as new microchip technologies are released, thereby improving the workload-handling capability of the engine.

The virtualization function of our system decouples the physical storage—as delivered by RAID controllers—from the storage management functions, and it migrates those functions to the network. Using such a system, which can perform storage management in a simplified way over any attached storage, is different from implementing storage management functions locally at the disk controller level, the approach used by EMC Corporation in their Symmetrix** or CLARiiON** products[10] or Hitachi Data Systems in their Lightning and Thunder products.[11]

Aside from the basic virtualization function, the virtualization engine supports a number of additional advanced functions. The high-availability *fast-write cache* allows the hosts to write data to storage without having to wait for the data to be written to physical disk. The *point-in-time copy* allows the capture and storing of a snapshot of the data stored on one or more logical volumes at a specific point in time. These logical volumes can then be used for testing, for analysis (e.g., data mining), or as backup. Note that if a point-in-time copy is initiated and then some part of the data involved in the copy is written to, then the current version of the data needs to be saved to an auxiliary location before it is overwritten by the new version. In this situation, the performance of the point-in-time copy will be greatly enhanced by the use of the fast-write cache.

Figure 1 A SAN storage control system with three VE nodes



The *remote copy* allows a secondary copy of a logical disk to be created and kept in sync with the primary copy (the secondary copy is updated any time the primary copy is updated), possibly at a remote site, in order to ensure availability of data in case the primary copy becomes unavailable. *Transparent data migration* is used when adding, removing, or rebalancing load to back-end storage. When preparing to add or remove back-end storage, data are migrated from one device to another while remaining available to hosts. The transparent data migration function can also be used to perform back-end storage load balancing by moving data from highly utilized disks to less active ones.

Hardware configuration. Figure 1 shows the configuration of a SAN storage system consisting of a SAN fabric, two hosts (a Windows NT** host and a UNIX host), three virtualization engine nodes, and two RAID controllers with attached arrays of disks.

Each *virtualization engine node* (abbreviated as VE node) is an independent Pentium-based server with multiple connections to the SAN (four in the first product release), and either a battery backup unit (BBU) or access to an uninterruptible power supply (UPS). The BBU, or the UPS, provides a nonvolatile memory capability that is required to support the fast-write cache and is also used to provide fast persistent storage for system configuration data. The VE node contains a *watchdog timer*, a hardware timer that ensures that a failing VE node that is not able (or takes a long time) to recover on its own will be restarted.

The SAN consists of a "fabric" through which hosts communicate with VE nodes, and VE nodes communicate with RAID controllers and each other. Fibre Channel and the Gigabit Ethernet are two of several possible fabric types. It is not required that all components share the same fabric. Hosts could communicate with VE nodes over one fabric, while VE nodes communicate with each other over a second fabric, and VE nodes communicate with RAID controllers over a third fabric.

The *RAID controllers* provide redundant paths to their storage along with hardware functions to support RAID-5 or similar availability features.[2] These controllers are not required to support extensive caching and would thus be expensive, but access to their storage should remain available following a single

failure. The storage managed by these RAID controllers forms the common storage pool used for the virtualization function of the system. The connection from the RAID controller to its storage can use various protocols, such as SCSI (Small Computer System Interface), ATA (Advanced Technology Attachment), or Fibre Channel.

Challenges for the software designers. Distributed programs are complex; designing and testing them is hard. Concurrency control, deadlock and starvation avoidance, and especially recovery after failure—these are all areas of great complexity.

Enterprise-level storage control systems must deliver high performance. Because the hardware platform can be easily assembled by any potential competitor and because high performance is a baseline requirement in the industry—I/O wait time is a major component of elapsed time for many host applications—it is not acceptable to trade processing power for additional layers of software. In addition, it is highly desirable that, as the workload increases, the system response time stays in the linear region of the performance curve (response time vs workload). In other words, we try to avoid the rapid deterioration of the performance when the workload increases beyond a critical point.

Deadlock and starvation, either at a single VE node or involving several VE nodes, are to be avoided. Host data written to the virtualization engine system must be preserved, even in case of failure. Specifically, following the completion of a write request from a host to the storage control system, the data must remain safe even after one failure. A well-designed storage control system tries to keep data safe wherever possible even when multiple failures occur.

The hardware platform for the system may change over time. Over the life of the storage control system, it is possible that other hardware platforms may become more cost effective, or for some other reason it may become advantageous to change the hardware platform. The software architecture for the storage control system should be able to accommodate such change. The architecture should also allow for changes in function: new functions are added, old functions are removed.

Software architecture. In order to address the challenges just discussed, we made several fundamental choices regarding the software environment, as follows.

Linux was chosen as the initial operating system (OS) platform and POSIX** (Portable Operating System Interface) was chosen as the main OS services layer for thread, interthread, and interprocess communication services. Linux is a well known and stable OS, and its environment and tools should present an easy learning curve for developers. POSIX is a standard interface that will allow changing the OS with a relatively small effort.

The software runs almost all the time in user mode in order to reduce kernel-mode/user-mode switching.

The software runs almost all the time in user mode. In the past, storage controller software has generally run in kernel mode or the equivalent, which meant that when a software bug was encountered, the entire system crashed. Entire critical performance code paths, including device driver code paths, should run in user mode because kernel-model user-mode switches are costly. In pursuing this execution model, debugging is faster because OS recovery from a software bug in user mode is much faster than rebooting the OS. Also, user-mode programs can be partitioned into multiple user-mode processes that are isolated from each other by the OS. Then failure of a "main" process can be quickly detected by another process, which can take actions such as saving program state—including modified host data—on the local disk. For performance reasons, the execution of I/O requests is handled in one process via a minimum number of thread context switches and with best use of L1 and L2 caches in the processor.

The code that runs in kernel mode is mainly the code that performs initialization of hardware, such as the Fibre Channel adapter card, and the code that performs the initial reservation of physical memory from the OS. Most of the Fibre Channel device driver runs in user mode in order to reduce kernel-mode/user-mode switching during normal operation.

Figure 2 illustrates the main components of the virtualization engine software. The software runs primarily in two user-mode processes: the External

Configuration Process (on the left in Figure 2) and the I/O Process (on the right in Figure 2). During normal operation, the I/O Process could occasionally run in kernel mode, mainly when some part of a physical page needs to be copied to another physical page.

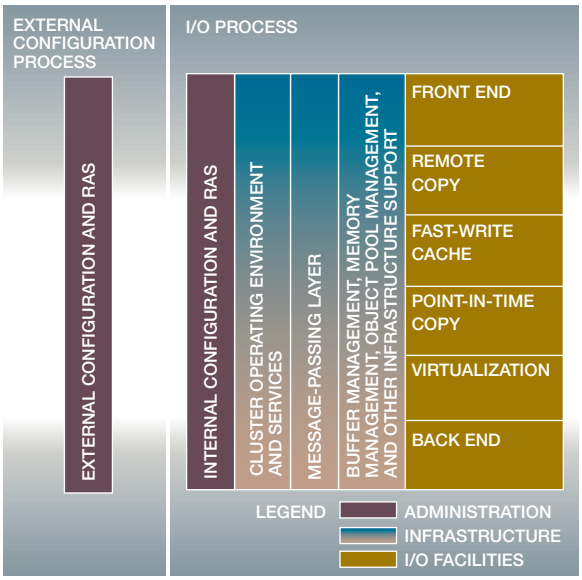
On a so-chosen VE node within a cluster, the *External Configuration Process* supports an administrator interface over an Ethernet LAN (local-area network). This interface receives configuration requests from the administrator and the software coordinates the processing of those requests with the I/O process on all VE nodes. The External Configuration Process is responsible for error reporting and for coordinating repair actions. In addition, the External Configuration Process on each VE node monitors the health of the I/O Process at the VE node and is responsible for saving nonvolatile state and restarting the I/O Process as needed.

The *I/O Process* is responsible for performing internal configuration actions, for serving I/O requests from hosts, and for recovering after failures. The software running in the I/O Process consists of *Internal Configuration and RAS* (reliability, availability, and serviceability) and the *distributed I/O facilities* (shown on the right in Figure 2); the rest of the modules are referred to as the *infrastructure* (shown in the middle of Figure 2). The use of the "distributed" attribute with an I/O facility means that the I/O facility components in different nodes cooperate and back each other up in case of failure.

The infrastructure, which provides support for the distributed I/O facilities, consists of the *cluster operating environment and services*, the *message-passing layer*, and other infrastructure support, which includes buffer management, hierarchical object pool management, tracing, state saving, and so on. The message-passing layer at each VE node enables cluster services as well as I/O facilities to send messages to other VE nodes. Embedded in the messaging layer is a heartbeat mechanism, which forms connections with other VE nodes and monitors the health of those connections.

The *virtualization* I/O facility presents to hosts a view of storage consisting of a number of logical disks, also referred to as *front-end* disks, by mapping these logical disks to back-end storage. It also supports the physical migration of back-end storage. The fast-write cache I/O facility caches the host data; write requests from hosts are signaled as complete when the write data have been stored in two VE nodes. The

Figure 2 The software architecture of a VE node



back-end I/O facility manages the RAID controllers and services requests sent to back-end storage. The *front-end* I/O facility manages disks and services SCSI requests sent by hosts.

The major elements of the architecture that will be covered in detail in succeeding pages are: the cluster operating environment, which provides configuration and recovery support for the cluster, the I/O facility stack, which provides a framework for the distributed operation of the I/O facilities, buffer management, which provides a means for the distributed facilities to share buffers without physical copying, and hierarchical object pools, which provide deadlock-free and fair access to shared-memory resources for I/O facilities.

The cluster operating environment

In this section we describe the cluster operating environment, the infrastructure component that manages the way VE nodes join or leave the cluster, as well as the configuration and recovery actions for the cluster.

Basic services. The cluster operating environment provides a set of basic services similar to other high availability cluster services such as IBM's RSCT (Reliable Scalable Cluster Technology) Group Ser-

vices.[12,13] These services are designed to keep the cluster operating as long as a majority of VE nodes can all exchange messages.[14]

Each VE node communicates its *local view* (the set of other VE nodes with which it can exchange messages) to its neighbors using a protocol known as a *link state protocol*.[15] Thus, each VE node maintains a database containing the local views of all VE nodes it knows about. When a connection is discovered or lost, the local view is updated.

At times, the database of views changes in such a way that views on VE nodes are not all consistent. Then the cluster undergoes a partitioning so that all VE nodes in a partition have consistent views. The algorithm used for partitioning guarantees that as soon as a set of VE nodes have a consistent database of views—which will happen once the network stabilizes—all these nodes arrive at the same conclusion concerning membership in a partition.

Next each node tries to determine whether the partition that includes itself is the new cluster. There are three possible cases.

1. The set of VE nodes in the partition is not a majority of the previous set. Then the VE node stalls in order to allow the cluster to operate correctly.
2. The set of VE nodes in the partition as viewed by our VE node is a majority, but other VE nodes in the partition have a different idea of what the partition is. This might be the case during network transient conditions, and the cluster will stall until the network has become stable.
3. The set of VE nodes in the partition as viewed by our VE node is a majority,[12] and all other VE nodes in the partition agree. The partition becomes the new cluster.

Communication among the VE nodes of the cluster involves the cluster *boss*, the lowest numbered VE node in the cluster, who controls the broadcast of messages called *events*, which are distributed throughout the cluster in a two-phase protocol. Events, which can originate at any VE node, are first sent to the boss. The boss collects and orders the events, and then distributes them to all VE nodes in the cluster.

VE nodes provide I/O service under a *lease*, which must be renewed if the VE node is to continue service. The receipt of any event by a VE node implies the renewal of its lease to provide service on behalf of the cluster (e.g., service I/O requests from host) for a defined amount of time. The cluster boss ensures that all operating VE nodes of the cluster receive events in time for lease renewal. The boss also monitors VE nodes for lease expiration, when VE nodes stop participating in the voting protocol for new events, and in order to inject appropriate VE node events into the system to inform the I/O facilities about VE nodes that have changed availability state.

Replicated state machines. The cluster operating environment component has the task of recovery in case of failure. Recovery algorithms for distributed facilities are typically complex and difficult to design and debug. Storage controllers add another layer of complexity as they often manage extensive nonvolatile (persistent) state within each VE node.

System designers find it convenient to view a distributed system as a master and a set of slaves. The master sets up the slaves to perform their tasks and monitors them for correct operation. In the event of an error, the master assesses the damage and controls recovery by cooperating with the functioning slaves. The master is distinguished by its holding complete information for configuring the system, also referred to as the *state*. The slaves, in contrast, hold only a part of that information, which each slave receives from the master.

Therefore, in designing recovery algorithms for a distributed facility, one can think of the job as designing a set of algorithms to operate in the master when slaves fail, with another set of algorithms to operate in the slaves when the master fails. Of these two, the second set is more difficult. Generally the slaves are not privy to the entire master state, although each slave might have a portion of that state. Often the algorithms entail the nomination of a new master and must deal with failures that occur during the transition period where a new master is taking control.

If the master cannot fail, then the job is much easier. For that reason, we decided to create a "virtual" master that cannot fail using a method derived from the work of Leslie Lamport[16] and involving a concept known as a *replicated state machine* (RSM). An RSM can be viewed as a state machine that operates identically on every VE node in the cluster. As long as a majority of VE nodes continues to operate in the cluster, the RSM algorithms—which run on every VE node in the cluster—also continue to oper-

ate. If, following a failure, a majority of VE nodes is no longer available, then the RSMs do not get any new stimulus and the VE nodes are suspended until a majority of VE nodes is once again available.

For the RSMs to operate correctly, the initial state must be the same on all VE nodes in the cluster, all VE nodes must process the same stimuli in the same order, and all VE nodes must process the stimuli in exactly the same way. The cluster operating environment ensures that the cluster state is correctly initialized on a VE node when it joins the cluster, that stimuli are collected, ordered, and distributed to all VE nodes in the cluster, and that stimuli that have been committed to the cluster are guaranteed to be processed on each VE node in the cluster—even through a power failure—for all VE nodes that remain in the cluster. The cluster operating environment also ensures the consistency of the persistent cluster state on all operating VE nodes. A distributed I/O facility can leverage this function by storing as much of its persistent meta-data (e.g., system configuration information) in cluster state as is possible without impacting performance goals (updating that information is not fast). Meta-data stored in this way are automatically maintained through error and power failure scenarios without the distributed I/O facility being required to supply any additional algorithms to assure the maintenance of the meta-data.

The main uses of RSMs are for configuration and recovery. Therefore, their operation does not impact system performance during normal servicing of I/O requests. A stimulus presented to an RSM is referred to as an event. An event, the only type of input to RSMs, can be associated with a configuration task or with a recovery task. Normal processing of I/O requests from hosts does not generally involve processing of events.

RSMs and I/O facilities. Figure 3 illustrates an I/O facility in a three-VE-node cluster. Each distributed I/O facility has two distinct sets of modules.

First, the I/O modules (shown at the bottom of Figure 3) process I/O requests. This is the code that runs virtually all the time in normal operation.

Second, the modules in the cluster control path (in the middle of Figure 3) comprise the RSM for the distributed I/O facility. They listen for events, change cluster state, and take action as required by communicating with the I/O modules.

RSMs and I/O modules do not share data structures. They only communicate through *responses* and *events*. When an I/O module wishes to communicate with an RSM, it does so by sending an event (labeled "proposed event" in Figure 3). When an RSM wishes to communicate with an I/O module, it does so by means of a *response* (blue arrows in Figure 3) sent through a *call gate* (gray in Figure 3).

Recalling the model of a replicated state machine, an RSM operates on each VE node as if it is a master commanding the entire set of VE nodes. An RSM may be programmed, for example, to command VE node X to take action Q in some circumstance. In reality a state machine response is implemented as a function call that is conditionally invoked according to the decision of a call-gate macro, with the effect in this example that the response Q only reaches the I/O modules on VE node X. Although the RSM at each VE node attempts to make the response, the function of the call gate is to ensure that only the I/O modules on specified VE nodes receive the response from the RSM. The decision is determined at a VE node answering the following questions.

Is the response being sent to this VE node? Remember that the same RSM code operates on all VE nodes. If only node X is to perform an action, the call gate must block the function from being called on all nodes other than node X.

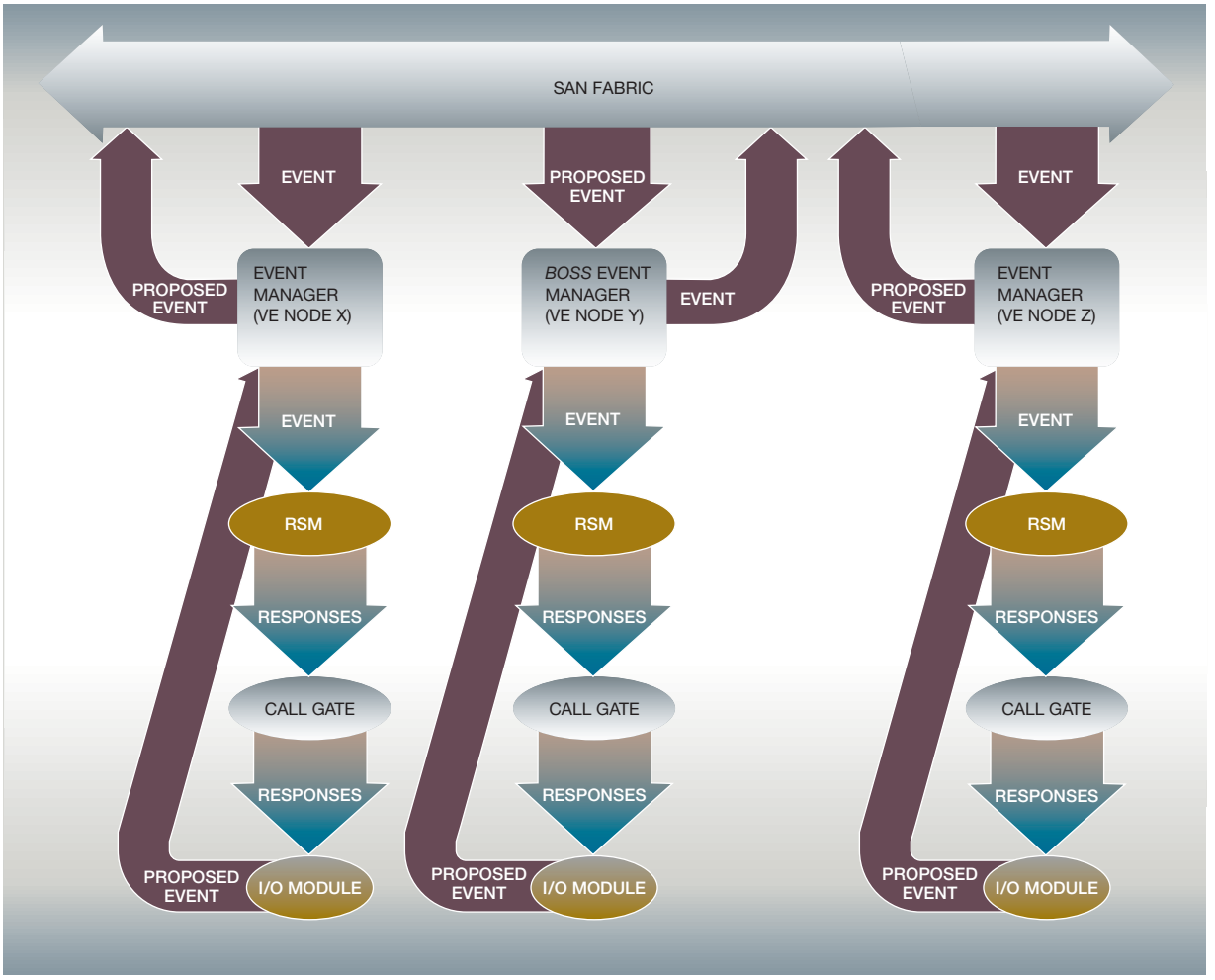
Is this node on line? The I/O modules on a node may not act on events that occurred while it was not an on-line cluster member and therefore not holding a lease.

Is the RSM code executing in the second execution phase? Each event is dispatched two times to each RSM, each time pointing to a different copy of cluster state. This algorithm, by maintaining two copies of the cluster state, only one of which is modified at a time, allows rollback, or roll-forward, to an event-processing boundary in the event of power failure. The call gate ensures that responses are seen at most once by I/O modules even though each event is processed twice by each RSM.

The I/O facility stack

A VE node accepts I/O requests from hosts, i.e., requests for access to front-end disks. The I/O facilities in the I/O Process are responsible for servicing these requests. Figure 4 shows the I/O facilities at a VE node arranged in a linear configuration (we re-

Figure 3 An I/O facility in a three-VE-node cluster

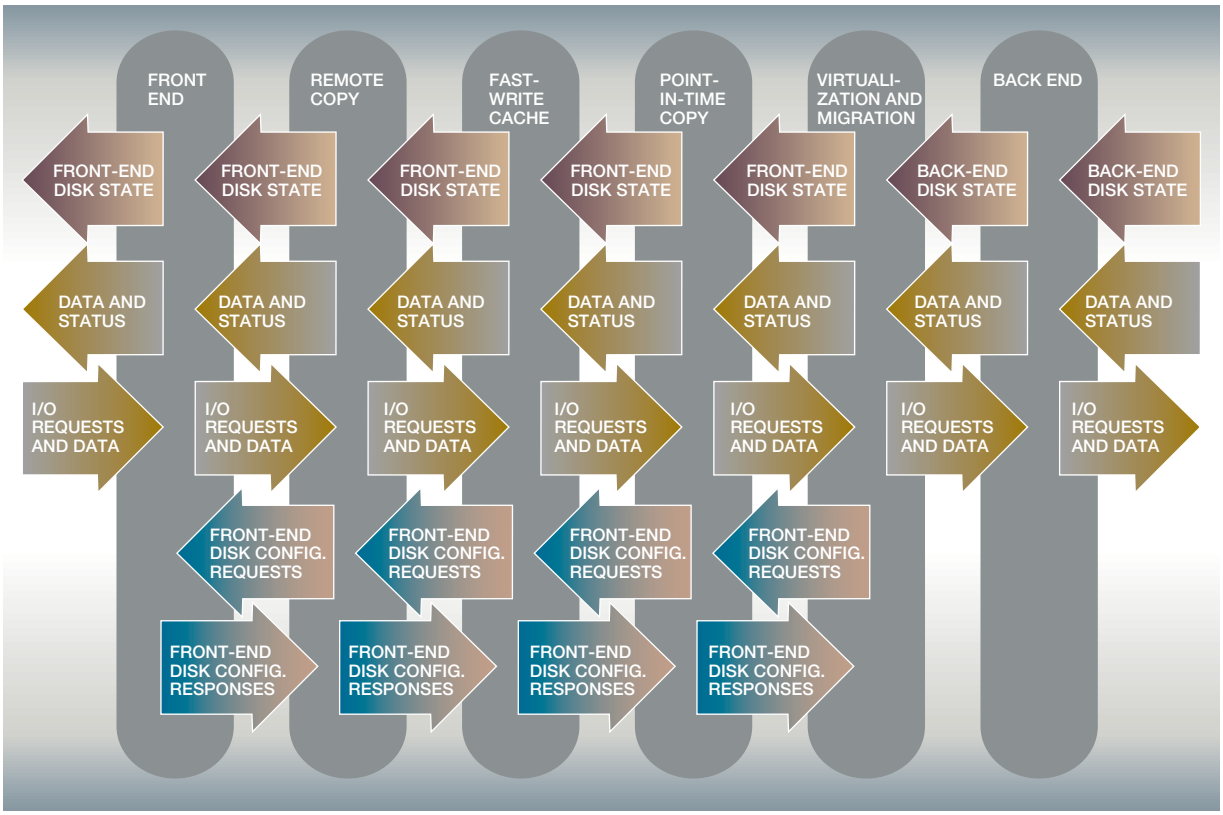


fer to it as the *stack*). The I/O requests arrive at the leftmost I/O facility and are then handed over from facility to facility, and each facility processes the I/O request as required according to the particulars of the request, usually involving a command (e.g., read, write) and a front-end disk. Each I/O facility in the stack has the same API (application programming interface) to the facility on either side of it except for the end points of the stack. In fact an I/O facility is not aware of the type of its adjacent facilities. Relative to an I/O facility, its neighbor on the left is its IOClient, while its neighbor on the right is its IOServer.

The stack configuration offers a number of advantages. The interfacing between facilities is simplified because each facility interfaces with at most two other facilities. Some testing can be performed even before all the facilities are implemented, while the symmetry of the stack configuration makes debugging easier. Additional facilities can be added simply by insertion in the stack. Testing the implementation is simplified when using scaffolding consisting of a dummy front end that generates requests and a dummy back end that uses local files rather than SAN-attached disks.

The I/O facilities are bound into the stack configuration at compile time. The symmetry of the stack configuration does not imply that the placement of the various facilities is arbitrary. In fact each I/O facility is assigned a specific place in the stack. The front-

Figure 4 The I/O facility stack and the interactions between stack elements



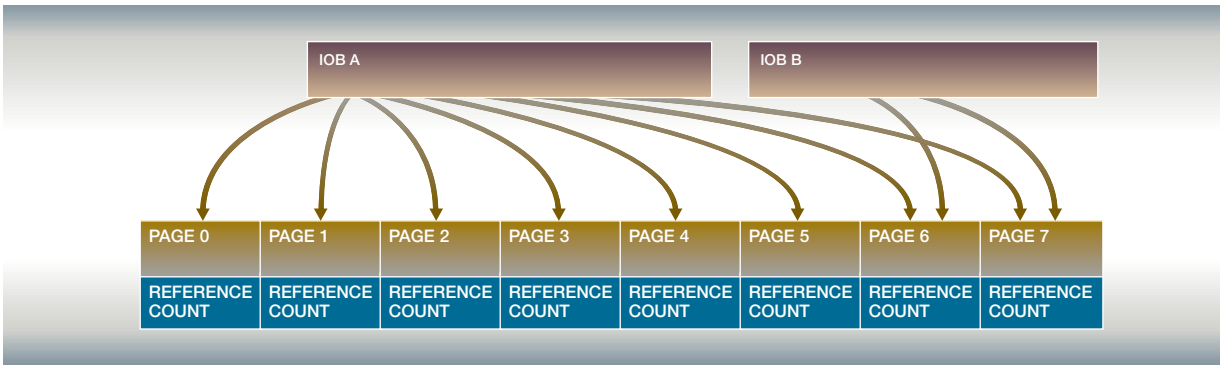
end facility serves specifically as a SCSI target to hosts acting as SCSI initiators, while the back-end facility serves as a SCSI initiator to back-end storage devices, that is, the SCSI targets. The point-in-time copy facility is placed to the right of the fast-write cache facility, so that host write requests would be serviced by the fast-write cache without needing extra I/O that might be required by the point-in-time-copy of the source to target.

There are three types of interfaces between I/O facilities. The first is for use by I/O modules and involves submission of I/O requests and transfer of data (shown in the second and third rows of arrows in Figure 4). A SCSI request received by the front-end facility is handed over to its IOServer, and so on down the stack. For example, a read request for noncached data is handed over by each I/O facility to its IOServer all the way to the back-end facility, which sends a SCSI request over the SAN fabric to a SCSI target.

A second interface, involving events as input to RSMs, is for communicating the availability of front-end or back-end resources (shown in the top row of arrows in Figure 4). An I/O facility's RSM receives a disk-path event (an event signaling the availability of a disk on a specific VE node) from its IOServer, and sends to its IOClient a disk-path event that signals its ability to service I/O requests targeted to the specified disk on the specified VE node. For example, if a back-end disk is removed from the system, then the virtualization I/O facility must identify all front-end disks that make use of that back-end disk, and, for each of these front-end disks, it must inform its IOClient that all the corresponding disk paths are unavailable. It does so by issuing an appropriate disk-path event to its IOClient.

The third type of interface is for servicing front-end disk configuration requests (bottom two rows of arrows in Figure 4). Front-end disk configuration re-

Figure 5 An example of buffer allocation involving two IOB's and eight-page data structures with associated reference counts



quests come from the Internal Configuration and RAS component, but it is up to the I/O facilities to properly sequence the servicing of these requests. Some requests, such as front-end disk creation and termination, are handled by the virtualization facility. It processes the request by issuing an appropriate front-end disk event to its IOClient, which performs the action required and passes the same event to its IOClient, and so on until the left edge of the stack is reached.

Other configuration requests, such as the initiation of a point-in-time copy, are given to the point-in-time copy facility by Internal Configuration and RAS. The point-in-time copy facility issues a sequence of front-end disk configuration events to its IOClient, which are passed on until the top of the stack is reached. For example, triggering a point-in-time copy includes flushing modified data for a point-in-time source and discarding all data for a point-in-time target. These activities are asynchronous in nature and therefore a set of events are passed "down" (right) the I/O stack to signal asynchronous completion of a front-end disk configuration request.

Thus, the use of the stack structure to configure I/O facilities results in a simplified API between neighboring facilities and the flexibility to insert new function into the stack, or to remove function from the stack.

Buffer management

The virtualization engine has the capability to move host data (data read or written by the host) through without performing any memory-to-memory copy operations within the VE node. Prior to starting the I/O Process, the operating system loads the memory manager kernel module and instructs it to allocate real pages from memory for the use of the I/O Process. Then the I/O Process starts. Its memory manager allocates the appropriate number of real pages, and takes over the management of this resource. From that point on, the memory manager kernel module is idle, unless the I/O Process must perform a move or a copy that involves a physical page (a relatively rare occurrence under normal workload).

The host data handled by the VE nodes, as well as the meta-data, are held in buffers. These buffers are managed using IOB (I/O buffer) data structures and page data structures. The IOB contains pointers to up to eight-page data structures, as well as pointers to be used for chaining of IOBs when longer buffers are needed. The page data structure includes, in addition to a 4096-byte payload, a reference count whose use is described below. When I/O facilities in the stack communicate, only IOB pointers to data pages are passed (instead of the actual data) from I/O facility to I/O facility. This avoids copying of host data within the VE node.

There are times when two or more I/O facilities need to simultaneously operate on the same host data. Using Figure 5 as an example, when a host writes data to a VE node, both the remote copy and the fast-write cache I/O facilities may need to operate upon those data. The remote copy facility allocates an IOB A and starts the data transfer from the host. Once the data are in the buffer (pages 0 through 7), the remote copy starts forwarding the data to the secondary location in another VE node, and in parallel submits an I/O

request for writing pages 0 through 7 to the fast-write cache. The fast-write cache allocates IOB B, which now needs to point to the data to be handled. Because the data are already in memory, the pointers of B are simply *cloned*—set so that both A and B point to the same pages—and the reference counts of page structures 0 through 7 are incremented. The two I/O facilities now operate independently on shared buffer pages as long as neither one writes again into any of the pages. If, however, a write to a page that has a reference count greater than one occurs, then the buffer management allocates the I/O facility its own page by physically copying the contents of the page to a new page (the page was reserved when the IOB was cloned) and setting its IOB to point to the new page. Reference counts are appropriately adjusted for the pages involved in the operation.

When an I/O facility has no further use for a buffer (e.g., the remote copy facility finishes the writing of data to a secondary location), it frees it. The reference count for each page in the freed buffer is decremented. When the reference count for a page becomes zero, that page can be returned to the pool of available pages. It is therefore possible, for example, that the fast-write cache might free pages 0 through 5 in Figure 5, such that among the pages pointed to by an IOB some may be shared (pages 6 and 7 in Figure 5) by other IOBs, while other pages are not shared (pages 0 through 5), leaving the state of the two IOBs and page data structures as shown.

The power of this buffer management technique lies in allowing multiple I/O facilities to share host data—for the most part without performing physical copies—without requiring special protocols between I/O facilities to coordinate that sharing.

Hierarchical object pools

Within a VE node, there are a number of memory-related entities that I/O facilities need for performing their tasks. Host data handled by I/O facilities are stored in pages linked into buffers and managed with IOBs. Also needed are control blocks for work in progress, cache directory control blocks, and hardening rights (see below).

When a power failure occurs, nonvolatile memory in a VE node is implemented by spooling pages of memory to a local disk. Given that battery backup systems that provide the power supply for the backup have enough power for spooling only a subset of the entire memory, I/O facilities must ensure that only the data that need to be preserved through power failures (e.g., modified host data or persistent meta-data), get spooled to the local disk. To accomplish this, an I/O facility must reserve a *hardening right* for each page that needs to be preserved through a power failure.

All these memory-related resources are known as *memory objects*, or simply objects. It would be possible to statically allocate each I/O facility a set of objects for its exclusive use. However, it would then be possible that in some I/O facilities many objects might be idle, while in others some objects would be in short supply. It is better to use objects where they are most needed, although this must be balanced against the effort to prevent deadlock and starvation. We discuss here the technique we developed, and empirically validated, to enable the sharing of these objects.

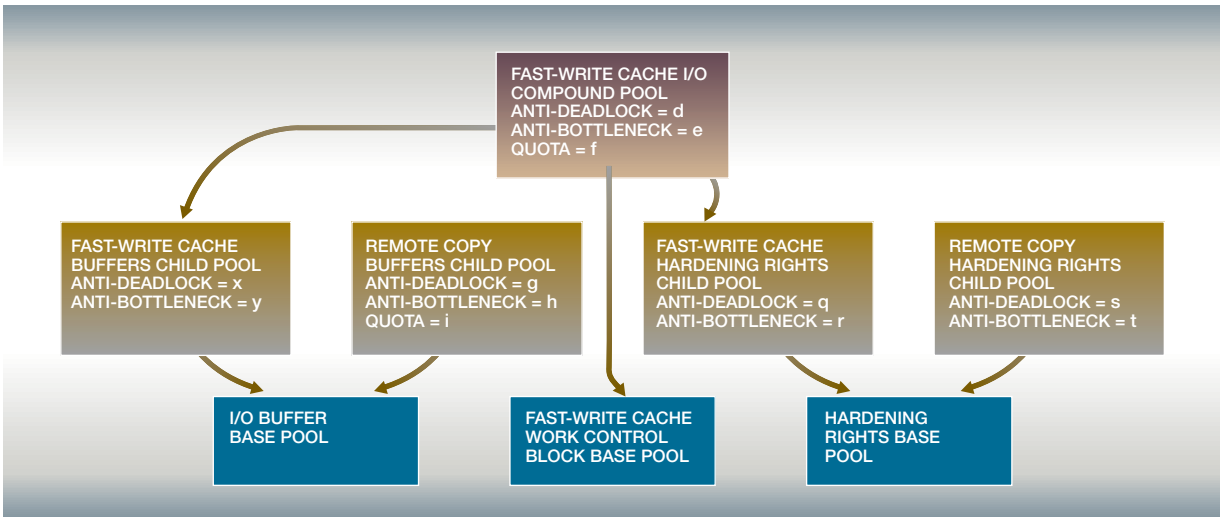
To ensure the spooling of a page of memory to disk in case of power failure, an I/O facility must reserve a hardening right for the page.

In our system, different streams of work arrive at a VE node from various places. Host requests arrive from a set of hosts, work requests also arrive from partner VE nodes. Deadlock may occur if concurrent work processes successfully reserve some, but not all, of the objects they require, and if work item A is waiting for an object held by B, while A holds an object B is waiting for. Deadlock prevention can be difficult to achieve when multiple distributed facilities are competing for objects, and when the many possible execution paths exhibit a wide range of possible reservation sequences for those objects.

Starvation occurs when some streams of work have access to objects to the detriment of other streams whose progress is impeded by lack of objects, usually due to unfairness in the allocation policies in the system.

The virtualization engine hierarchical object pool was designed to allow the I/O facilities to share objects while satisfying a set of goals. First, each I/O facility

Figure 6 A hierarchy of object pools



should be able to have its execution paths—and in particular the sequence by which objects are reserved and unreserved—to be deadlock free, independent of execution paths in other I/O facilities. (This allows the design of each facility to be independent of other I/O facilities.) Second, objects should be quickly made available to more "active" streams while ensuring that processing of all streams in the VE node is "making progress" (and thus avoid starvation). Last, objects should be distributed throughout the system such that balanced performance is experienced by all active streams of work, i.e., prevent starvation.

Object pools, regardless of type, support the following operations: (1) *reserve* guarantees the right to use an object from the specified pool, (2) *allocate* allocates an object from the specified pool, (3) *free* frees an object and returns it to the pool from which it was allocated, and (4) *unreserve* returns the guaranteed right to use an object in the specified pool.

This resource allocation scheme splits object management into two primitive pairs: reserve/unreserve and allocate/free. The first pair deals with object accounting without any allocation or deallocation of objects. The advantage of this scheme is that objects can be allocated but unreserved, e.g., pages of memory can hold data in the hope of a cache hit but can be reclaimed if they are required somewhere else.

Objects of a specific type are placed into hierarchies of object pools as shown in a simplified example in Figure 6. For each object type (e.g., buffers) there is a *base* pool (such as the I/O buffer base pool in Figure 6) into which all objects of that type are initially placed. For each object type there is only one base pool. Figure 6 shows three base pools (bottom layer).

From each base pool a hierarchy of *child* pools can be created, as demonstrated in the middle and upper layers in Figure 6. A child pool, with the exception of a compound pool to be described below, has only one parent: either a base pool or another child pool. In order to service reservation and allocation requests, a child pool reserves and allocates objects from its parent, who in turn may have to reserve and allocate from its parent. As a base pool gets low on objects used to satisfy reservations, it calls on its child pools and so on up the hierarchy to reclaim (i.e., free and unreserve) objects so as to make them available in the pools that are low. This way objects can dynamically migrate to work streams in need of memory resources.

A *compound* pool is a special type of child pool that aggregates objects from different parents—base pools or child pools—so that a request involving multiple object types can be satisfied from a compound pool in a single step. Thus, the compound pool enables faster servicing of object requests. It is also the best way to aggregate objects in order to feed lines of credit (see below). Figure 6 shows five child pools, one of which is a compound pool (top layer).

An I/O facility may create a child object pool from a base pool or from another child pool. It may also create multiple child pools from the same base pool. This might be indicated, for example, when the facility needs a different pool for each VE node that might be feeding it work. For the discussion below, we point out that a child pool reserves objects from its parent in order to satisfy reservation requests from clients.

A child pool has three important attributes. The *antideadlock threshold*, specified by the I/O facility when creating a child pool, is the minimum number of objects that a child pool must have reserved to it. When creating a child pool, if that number of objects cannot be reserved from its parent, then the pool creation fails. The value of the antideadlock threshold should be picked to be at least the maximum number of objects that might be required for any one work item. It is expected that an object reservation necessary for processing of a work item will be performed "atomically" from each pool, and if the work item requires reservations of objects from multiple pools, the object reservation requests will be performed in a fixed order; this fixed order will apply to object reservation requests made by all work items. As long as this sequence is followed, no work item will wait indefinitely to reserve a set of objects from a child pool. Since each facility creates its own set of child pools—each pool with a specified antideadlock threshold—no facility can completely starve another facility.

The *antibottleneck threshold*, also specified by the I/O facility when creating the pool, is the number of objects that, if allocated to a stream of work items, should sustain reasonable throughput. Once a child pool starts receiving object reservation requests from the I/O facility that created it, it over-reserves from its parent with the intention of quickly increasing up to the antibottleneck threshold the number of objects reserved to that pool. After the number of objects reserved to the pool reaches that threshold, the child pool will allow more reservations to accumulate in that pool if requested by the I/O facility, but it will not continue to over-reserve. In the reverse direction, a child pool will readily allow the number of objects reserved to it to bleed down to the antibottleneck threshold but will generally require periods of inactivity to occur before allowing reservations to a pool to diminish below the antibottleneck threshold limit.

The *quota*, an optional attribute assigned by the I/O facility during creation, is the maximum number of objects that can be reserved to a pool by any one client. Setting a quota on child pools ensures that no stream of work can accumulate objects such that other streams of work suffer too much.

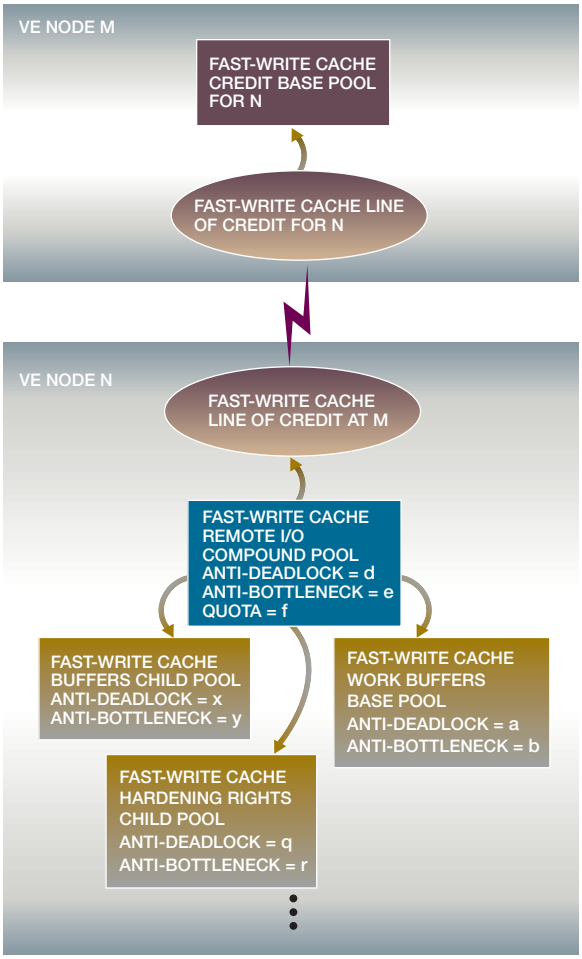
An I/O facility uses the antideadlock threshold attribute of the child pools it creates to guarantee that each work item it services will eventually get access to the objects it needs for the work item to make progress. The I/O facility also uses the antibottleneck attribute of the child pools it creates to enable its various stream of work items (also referred to as work streams) to progress at a satisfactory rate when all areas of the VE node are in heavy usage, and it uses the quota attribute to ensure that its work stream does not starve work streams occurring in other I/O facilities. Last, the adaptive reservation and reclaiming algorithms that operate along an object hierarchy in a VE node promote the redistribution of shared object types when the activity level of various work streams changes such that the more active work streams gain access to more of the shared objects.

The object pool mechanism governs use of objects within a VE node. An extension of this mechanism involving lines of credit is used for object management involving partner VE nodes in the cluster, as described below.

Lines of credit. It is often the case that an I/O facility on one VE node requires its peer on another VE node to perform some work on its behalf. For example, suppose the fast-write cache facility on one VE node (such as VE node N in Figure 7) needs its peer on another VE node (such as VE node M in Figure 7) to receive and hold newly written data from a host. In such a case, starvation or deadlock can occur not only within a VE node, but also between VE nodes. In fact designing distributed I/O facilities to be deadlock- and starvation-free is one of the more difficult challenges that confronts the designer.

Some distributed systems are designed in such a way that they can occasionally become deadlocked during normal operation. Such systems often deal with deadlocks through detection and restarting of deadlocked threads. Those systems perform satisfactorily if the deadlocks occur infrequently. Object deadlocks in the virtualization engine could happen more frequently because the probability of lock collision is higher. Consequently, we designed the virtualization engine software to be deadlock-free. Toward this

Figure 7 A compound child pool connected to a line of credit for use by another VE node



end the object pool facility was extended to service other VE nodes through lines of credit as described below. In addition, we made sure that chains of work requests for other VE nodes are never circular (e.g., VE node A sends a type of work request to VE node B, causing B to send a work request to C, causing C to send a work request to A). Circular work requests would violate the object pool rule that a work item reserves atomically from any given pool.

A virtualization engine line of credit is based on the concept of a *credit provider* such as VE node M in Figure 7 (an I/O facility's role of servicing remote work requests from its peers at other VE nodes) and a *credit consumer* such as VE node N in Figure 7 (an I/O facility's role of requesting work by peers at other VE nodes). The credit provider "exports" objects from object pools to peers on one or more other VE nodes in the form of credits, which the consumer can then reserve in order to guarantee access to the required object (often a compound object) on the other VE node.

Typically a credit provider sets up a compound pool (such as the pool shaded blue in Figure 7) containing objects that might be needed to serve work items from another VE node. That pool is then bound to a line of credit (such as the bottom line of credit in VE node M in Figure 7) to be used by the consumer (such as VE node N in Figure 7). The line of credit is established between two nodes as follows.

The consumer sets up a base pool to hold credits and binds that pool to a specific line of credit for objects provided at a specified other VE node. When a consumer VE node (such as VE node M in Figure 7) joins the cluster, the provider VE node (such as VE node N in Figure 7) and the consumer activate the line of credit via a messaging protocol between the two VE nodes, and thus the line of credit is established. Thereafter, when the consumer wishes the provider to perform work on its behalf, it first reserves a credit from its corresponding base pool (such as the pool in VE node M at the top of Figure 7). When that work item is completed on the provider VE node (using a compound object such as one taken from the blue shaded object pool in VE node N in Figure 7) and a response is sent to the consumer VE node, the consumer VE node unreserves the credit back to the appropriate base pool (such as the pool in VE node M at the top of Figure 7).

The objects in the object pool that is bound to a provider credit pool are assigned values for the antibottleneck, antideadlock, and quota attributes to regulate usage of those pools in the same way as the other object pools in the VE node. Therefore the number of objects reserved at any point in time to perform work between any provider/consumer pair of VE nodes varies adaptively, depending on the flow of work between that provider/consumer pair on the two VE nodes, the flow of work between other provider/consumer pairs on the two VE nodes, and the antideadlock thresholds, antibottleneck thresholds, and quotas set on the object pools for each.

In our system, lines of credit are currently used primarily by three I/O facilities: fast-write cache, remote copy, and point-in-time copy. These three are the

distributed I/O facilities that have internode protocols that operate actively during processing of I/O requests. For example, the fast-write cache implements distributed cache coherency via its internode protocols, and the lines of credit are used to regulate object usage resulting from those protocols. The other distributed I/O facilities have only infrequent need for internode protocols, and in those cases the cluster event system was chosen instead as an easy-to-use mechanism for implementing those protocols.

Preliminary results and conclusions

The virtualization engine system has been implemented and has gone through extensive function validation tests for over a year. Testing was conducted in a product test lab on dozens of test stands on configurations ranging from two through eight VE nodes, multiple hosts, and multiple RAID controllers. Functional testing confirmed that the system does improve the utilization of storage resources and that it provides a platform for advanced storage functions.

The hardware components assembled for the virtualization engine product are, as intended, almost exclusively standard parts. There was more work than initially expected in selecting a processing platform with the right mixture of I/O bandwidth, processing power, footprint, and power requirements. However, the right platform was identified after careful balancing of product goals against components available in the market.

We required our Fibre Channel driver to run in user mode, and the Fibre Channel ports to be able to both send and receive packets from hosts (which act as SCSI initiators that originate SCSI requests), RAID controllers (which act as SCSI targets that service SCSI requests), and other VE nodes (which act as SCSI targets when they receive messages and as SCSI initiators when they send messages). Because these two requirements could not be satisfied by the typical Fibre Channel driver, we needed to adapt the source code provided by a Fibre Channel adapter manufacturer and build additional software around it. This means that we will not be able to simply install an already-available driver if and when we move to another type of SAN adapter, and instead we will need to plan for a minor software development effort in order to make the transition. Although not ideal, we think the performance benefits justify this approach.

During the course of the project we have upgraded to new generations of hardware (e.g., server, Fibre Channel adapter) with little effort. We consider this evidence that product release cycles will be short, because we will be able to leverage the newest hardware available with minimal effort.

The only hardware component developed for the project was a small unit that included a hardware watchdog timer and interface logic to a front panel. The development of this hardware was kept out of the critical path of the project and was only a small part of the development effort.

There have been relatively few surprises concerning the hardware platform. Storage controller development projects commonly experience large setbacks due to problems with the hardware designed for the product or even problems with the standard ASICs (application-specific integrated circuits). Although there have been instances when unexpected hardware behavior required some work-around solution, these have been relatively minor efforts.

The performance of the system has met or exceeded the goals set for it at the beginning of the development effort, with less effort than expected for this type of project. In part this is because the use of off-the-shelf components has eliminated possible problems associated with the development of new hardware. In addition, the continuous advances in technology toward better and faster components has, at times, compensated for suboptimal choices made during the design process.

Linux has been a good choice for an OS. There have been some problems due to the fact that Linux is evolving quickly. Bugs have been uncovered in a few areas, but these have been worked around. The decision to run most of the critical software in user mode meant that no modifications to the Linux kernel were needed, which has helped with testing and the legal process preceding the launch of the product. In addition, the virtualization engine's use of Linux and POSIX was limited to the thread and shared lock libraries, starting up the VE node and other applications, the Ethernet driver, and the TCP/IP (Transmission Control Protocol/Internet Protocol) stack. This has lowered risk of encountering a Linux problem and has also enhanced the adaptability of the code to other platforms.

User-mode programming has been much easier to debug than kernel-mode programming. During testing, the user program has often terminated abruptly, but the operating system rarely hung or failed, and

gathering system state data and traces has been easier.

The isolation between user-mode processes in a VE node has been particularly beneficial. For example, a bug in the I/O Process does not affect the External Configuration Process. The External Configuration Process is thus able to successfully harden modified host data and system state and then reinvoke the I/O Process. In addition, during recovery from error, by having one user-mode process monitor another, the probability of losing both copies of modified host data is minimized.

The replicated state machine model has been a very useful tool for implementing the distributed recovery function, particularly for formulating recovery strategies and algorithms and for storing state data. One of the more challenging tasks of implementing the recovery function was to get right the sequence in which the I/O modules are updated with the most current RSM data during the many different recovery scenarios.

The design of the I/O facilities based on a stack configuration is beneficial for several reasons. First, it allowed the development and testing of I/O facilities to be done in phases. Second, it allowed the design and development work of the I/O facilities to be partitioned to multiple teams, some at remote locations. Third, it will facilitate the incorporation of additional functionality and the reuse of software components in other projects.

The buffer management techniques and the object pool architecture have also aided in the development of the I/O facilities. The design of each I/O facility could proceed independently due to the transparency of the object management scheme.

There is ongoing work to deal with recovery in the face of programming errors inside a replicated state machine. When this happens, all VE nodes succumb to the same error and may end up attempting to restart endlessly. We are focusing our efforts on creating separate domains of cluster state and, through a human-guided process, allowing the less essential domains to become quiescent so that the cluster may recover.

The software architecture could be further improved by an API between Internal Configuration and RAS and the I/O facilities. Currently specialized interfaces are defined between specific I/O facilities and Internal Configuration and RAS. However, system flexibility would be enhanced if, based on such an API, choices could be made at compilation, or at run time, about which I/O facilities interact with which configuration actions.

Further work would also be desirable in creating common mechanisms to aid synchronization between I/O facility peers on separate VE nodes. We have realized that when I/O facility peers need to switch operating mode, the switching must be accomplished in two phases: in the first phase to inform all the VE nodes that the other VE nodes are also ready to switch, and in the second phase to perform the switch. This synchronization mechanism should be built into the infrastructure in order to facilitate the implementation of a distributed algorithm for I/O facilities.

During testing, some problems came up concerning the design of the I/O facility stack, specifically concerning how the handling of I/O requests by an I/O facility has to change when availability of back-end or front-end disks change (for example, when an I/O facility may forward I/O requests and when it must hold back those requests). Some subtle behavioral requirements were not well understood at first, which led to some integration problems and required that the API specification be clarified.

We have described the architecture of an enterprise-level storage control system that addresses the issues of storage management for SAN-attached block devices in a heterogeneous open systems environment. Our approach, which uses open standards and is consistent with the SNIA (Storage Networking Industry Association) storage model,[17] involves an appliance-based in-band block virtualization process in which intelligence is migrated from individual devices to the storage network. We expect the use of our storage control system will improve the utilization of storage resources while providing a platform for advanced storage functions. We have built our storage control system from off-the-shelf components and, by using a cluster of Linux-based servers, we have endowed the system with redundancy, modularity, and scalability.

Cited references and notes

1. See for example C. J. Conti, D. H. Gibson, and S. H. Pitkowsky, "Structural aspects of the System/360 Model 85," *IBM Systems Journal* **7,** No. 1, 2–14 (1968).
2. D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *International Conference on Management of Data*, Chicago, IL, June 1988, ACM, New York (1988), pp. 109–116.
3. G. A. Castets, D. Leplaideur, J. Alcino Bras, and J. Galang, *IBM Enterprise Storage Server*, SG24-5465-01, IBM Corporation (October 2001). Available on line at http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245465.pdf.
4. N. Allen, *Don't Waste Your Storage Dollars: What You Need to Know*, Gartner Group (March 2001).
5. *IBM TotalStorage Software Roadmap*, IBM Corporation (2002), http://www.storage.ibm.com/software/roadmap/index.html.
6. E. Lee, C. A. Thekkath, "Petal: Distributed Virtual Disks," *The Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, October 1996, ACM, New York (1996), pp. 84–92.
7. LeftHand Networks, http://www.lefthandnetworks.com/index.php.
8. VERITAS Software Corporation, http://www.veritas.com.
9. COMPAQ Storage, Hewlett-Packard Company, http://h18000.www1.hp.com/.
10. Symmetrix Networked Storage Systems, CLARiiON Networked Storage Systems, EMC Corporation, http://www.emc.com/products/platforms.jsp.
11. Global Storage, Hitachi Data Systems, http://www.hds.com/products/systems/.
12. *RSCT Group Services: Programming Cluster Applications*, SG24-5523, IBM Corporation (May 2000).
13. See section "Topology Services Subsystem" in "Parallel System Support Programs for AIX: Administration Guide," document number SA22-7348-04, IBM Corporation (2002), http://publibfp.boulder.ibm.com/epubs/pdf/a2273484.pdf.
14. When failure causes a cluster to be partitioned in two sets of VE nodes, the partition consisting of the larger number of VE nodes becomes the new cluster. When the two partitions have the same number of VE nodes, the partition containing the "quorum" VE node is selected—where the quorum VE node is a so-designated VE node in the original cluster for the purpose of breaking the tie.
15. J. Moy, "OSPF Version 2," RFC-2328, Network Working Group, Internet Engineering Task Force (April 1998), http://www.ietf.org.
16. L. Lamport, "The Part Time Parliament," *ACM Transactions on Computer Systems* **16**, No. 2, 133–169 (May 1998).
17. Storage Networking Industry Association, http://www.snia.org.

Accepted for publication January 2, 2003.

Joseph S. Glider *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (gliderj@almaden.ibm.com).* Mr. Glider is a senior programming manager in the Storage Systems and Software department at the IBM Almaden Research Center in San Jose, California. He has worked on enterprise storage systems for most of his career, including a stint as principal designer for FailSafe, one of the first commercially available fault-tolerant RAID-5 controllers. He has also been involved in supercomputer development, concentrating on the I/O subsystems. Mr. Glider received his B.S. degree in electrical engineering in 1979 from Rensselaer Polytechnic Institute, Troy, NY.

Carlos F. Fuente *IBM United Kingdom Laboratories, Hursley Park, Winchester HANTS SO21 2JN, United Kingdom (carlos_fuente@uk.ibm.com).* Mr. Fuente is a software architect with the storage systems development group at the IBM Hursley development laboratory near Winchester, UK. He has worked on high-performance, high-reliability storage systems since earning his M.A. degree from St. John's College, Cambridge University, in 1990 in engineering and electrical and information sciences.

William J. Scales *IBM United Kingdom Laboratories, Hursley Park, Winchester HANTS SO21 2JN, United Kingdom (bill_scales@uk.ibm.com).* Dr. Scales is the leader of the development team working on the IBM virtualization engine. He received his Ph.D. degree in 1995 at Warwick University, working with John Cullyer on developing a safety kernel for high-integrity systems. His research interests include RAID, high availability systems, and microprocessor architecture.