The Solaris Cluster Architecture

Yousef A. Khalidi

Version 0.51

Disclaimer: this is an evolving document. Not all features described in this document are committed at this time.

We refer to the architecture as the Solaris Cluster architecture. The first product of the Solaris Cluster architecture is the Sun Cluster (SC) 3.0 release.

See http://galileo.eng/docs/3.0_arch.ps for the latest version of this document. Send comments to yak@eng.

1.0 Executive Summary

The Solaris Cluster project is extending Solaris into a cluster operating system. A *cluster* is a collection of closely coupled computing nodes that provide a single view of a network service or services, such as a database or a file server. The key attributes of clusters are: support for highly-available access to data; scalable throughput and capacity; and support for modular system growth with a low entry price. Clustering commodity hardware has the potential to provide mainframe performance at a fraction of the cost. At the same time, building clusters out of large machines with high-speed backplanes, such as Wildcat, has the potential for providing even bigger and more scalable servers.

The project will deliver a platform that supports all existing Sun Cluster applications and APIs. These applications are in the area of on-line transaction processing, decision support, data warehousing, file service, and interactive network services. The clustering software will be portable over a variety of hardware platforms including SPARC and x86-based clusters.

SC 3.0 uses technology from the Solaris MC system, an advanced development prototype done at Sun Labs [1, 2]. The Solaris MC prototype serves as a starting point for the Sun Cluster product.

SC 3.0 delivers clustering extensions to the base Solaris system, along with several customer-visible features, including a continuously-available cluster file system, cluster-wide device access, global networking with load-balancing, browser-based system administration, and new Java and Wolfpack-like clustering APIs.

SC 3.0 provides seamless upgrade from previous Sun Cluster releases. and supports all Sun Cluster features including HA APIs, HA applications, and parallel databases.

2.0 Introduction

2.1 Why Customers Buy Clusters

It is important to understand why customer buy clusters. Based on several existing successful cluster products (VMScluster and Tandem NonStop) and numerous marketing reports, the perceived value of a cluster comes from:

- High availability configurations
- Scalable I/O, memory and processor capacity and performance
- Modular system growth
- Reduced system administration costs over equivalent collections of independent nodes
- Low entry level price

In current SMPs, a hardware or system software failure can bring down the entire system. A clustered system, on the other hand, can be configured to provide a highly available service by tolerating one or several hardware or software failures. If a hardware or software failure forces a node to stop, the remaining nodes are not affected by the failure and can quickly reconfigure the services to compensate for the failed node. In addition, most system administration tasks can be done while the service is on-line. The tasks include data backup/restore, installation of patches, software upgrades, and adding new hardware.

Scalable performance in the cluster is achieved by exploiting parallelism at the application level. For scalable throughput, cluster applications usually employ an inter-job parallelism. Many transactions can execute in parallel on the cluster nodes as long as the transactions do not access the same data in a conflicting mode. Inter-job parallelism applies to transaction processing, file service, and internet services. Intra-job parallelism is used by decision support systems to reduce response time for a query that scans a large database. Both types of parallelism are supported today by all major database vendors. Cluster nodes can be themselves SMPs, so traditional techniques for SMP scaling (e.g. multithreading and shared memory) still apply.

Mainframes and large SMP servers suffer from a high price for entry level configurations and a limited hardware life cycle. With a clustered solution, a customer can purchase an entry level system that is an inexpensive commodity server and then grow the system in a modular way by purchasing more nodes. As the cluster can mix hardware from several generations, the customer preserves his initial hardware investment while being able to use the latest hardware technology.

It is an important feature that clusters provide a “single system image;” that is, clusters will in most cases, appear to both users and system administrators as a single machine. Users have access to all cluster resources, such as files or devices, without having to explicitly reference the node to which they are attached, as shown in Figure 1. Similarly, administrators do not have to explicitly administer each node in the cluster. For example, software installation and backup are done as if the cluster was a single machine.

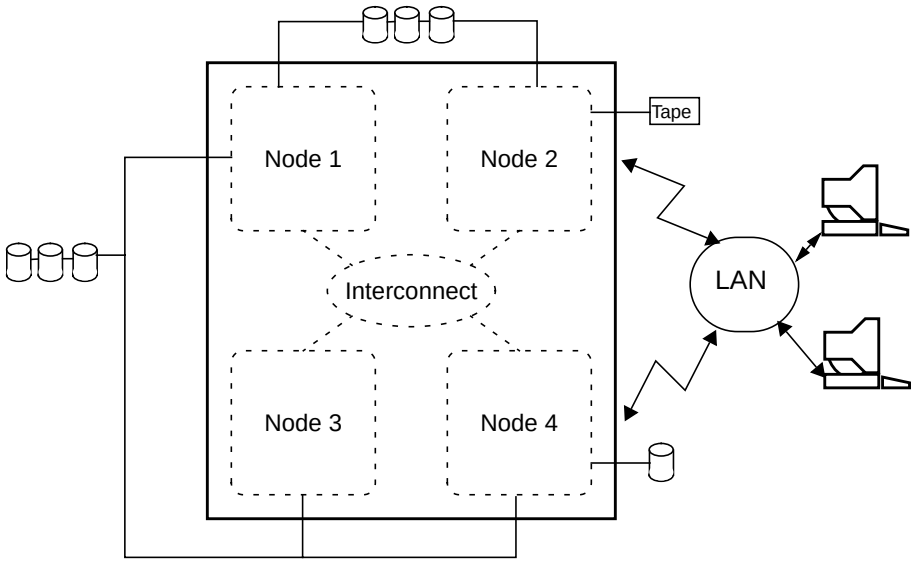


Figure 1. A Solaris Cluster

2.2 Cluster Hardware

The nodes within a cluster can be connected using a variety of technologies including LANs and ATM. Clusters that use LANs or ATM and traditional networking protocols as the node interconnection network typically require a few milliseconds for each inter-node communication. Newer high bandwidth links that use memory-based or other low-overhead protocols can communicate in tens of *microseconds*, or about two orders of magnitude faster than LAN or ATM-based clusters. In addition, the newer links can reach speeds of 100MBps to over 1GBps. In the latter case, a small number of links can achieve an aggregate bandwidth that surpasses a backplane. This makes clusters much more effective as a more scalable replacement for SMPs.

Examples of such high bandwidth hardware include Scalable Coherent Interconnect (SCI), Wildcat interconnect, P1394.2, ServerNet, Myrinet, and VIA.

2.3 Hardware Assumptions

SC 3.0 assumes that the cluster is composed of nodes (using any of the Solaris-supported processor architectures) that are connected using a low latency, high bandwidth interconnection network. Furthermore, SC 3.0 assumes that the interconnection network is private to the cluster and that all interconnected nodes run SC 3.0 software.[1] While these assumptions do not preclude the use of LAN or ATM technology, SC 3.0 may not

1. Mixing SC 3.0 and SC 3.x nodes will be supported for rolling upgrades.

run effectively if the latencies are high or the bandwidth is too low. In summary, SC 3.0 assumes the following properties of the interconnection network:

- Private interconnection network
- Low latency (10s of microseconds)
- High bandwidth (>100MBps)
- The ability to use more than one interconnect link for high availability and higher throughput
- A reset or failure of an interconnect adaptor/switch does not cause the whole interconnect fabric to fail

At a minimum, SC 3.0 assumes that most disk storage in the cluster is at least dual-ported with each port connected to a different node (e.g., dual-ported SCSI). SC 3.0 can handle disk storage that is multiported to more than two nodes (e.g., FC-AL). A configuration that allows direct node-to-disk I/O from any node to any disk is also supported (e.g., FC with switches). The cluster file system architecture allows concurrent direct data movement in such configurations (section 9.0).

2.4 What SC 3.0 is Not

A timely delivery is crucial to the success of SC 3.0. Therefore SC 3.0 is strictly focused on the functionality that we deem essential to competing in the commercial market segment. Other market segments such as high performance computing (HPC) are not explicitly addressed in early versions of the cluster product due to the limited market potential. Existing HPC toolkits such as PVM, Linda, LSF and MPI may be ported to the Solaris Cluster outside of the SC 3.0 project to provide inroads in this market.

SC 3.0 will also not directly address the needs of general workstations connected by LAN or WAN networks.

Support for NUMA architectures is orthogonal to SC 3.0. Any Solaris-supported node can be a member of the cluster, including NUMA nodes. However, SC 3.0 is not a replacement for NUMA OS support.[2]

3.0 Cluster Applications

This is a roughly prioritized list of market segments that SC 3.0 addresses. The intent of this section is not to solicit which market segments we are going to tackle, but rather to aid the determination of priorities for the Solaris clustering functional requirements:

1. Data base server
   - OLTP
   - DSS
2. Interactive Services
   - HTTP (WWW servers), Java

2. On the other hand, several technologies in SC 3.0 can be utilized for large NUMA machines, including global devices support with multiple path access.

3. Middleware
   - Transaction monitors
   - Enterprise JavaBeans
   - Messaging and CORBA-based middleware
4. File servers
5. Timesharing

Highly available databases are the most important applications for clusters. SC 3.0 will support all SC 3.0 HA DBMS applications, including Oracle, Informix and Sybase. IBM DB2 may be supported as will.

Parallel database servers are an important applications for clusters. All the major database vendors have recently re-architected their products to run on clusters. The following products are cluster ready: Informix XMP, Oracle Parallel Server and Parallel Query, Sybase MPP (formerly Navigation Server), and DB2/Parallel Edition. Clustering is supported for both OLTP (inter-job parallelism) and decision support (intra-job parallelism) environments. Several vendors have demonstrated near-linear scalability to a hundred of processors when using the parallel database servers on clustered hardware.

Transaction processing systems usually use a three-tier architecture consisting of an NC or PC client (or other terminal device), transaction monitor, and database server. It is important that the transaction monitors be cluster aware. Sun is working with the 3rd party vendors to help them port or extend their products to the cluster. The important products to Sun business include Tuxedo and TopEnd.

There are large suites of applications programs that are suitable for clusters. Examples of such suites are: Oracle Manufacturing or SAP. Using a cluster provides scalability and high availability to the organization. HA SAP will be supported by SC 3.0 timeframe, at least in a highly-available fashion.

Internet services are an example of applications that are naturally suited for clusters. The single system image feature of Solaris Cluster allows scalable throughput of the service while preserving the single network address for the service. Highly available, scalable Web servers will be enabled using SC 3.0 software. Many of the existing database services will be provided as interactive network services in conjunction with Web servers.

CORBA-based services are increasingly important as middleware and as part of three-tier and two-tier client-server computing. CORBA presents a new paradigm for building services. A service is constructed as a collection of co-operating server processes that communicate using object invocations. The CORBA programming paradigm is naturally suited for clusters.

Network file servers exhibit a natural parallelism; concurrent file request can be executed in parallel as long as they do not result in conflicting access to the same data. SC 3.0 makes a scalable file server because the capacity and throughput of the system grows as nodes are added to the system. The shared file system provides for automatic coherent replication of frequently accessed data across all the cluster nodes.

SC 3.0 will carry forward HA NFS support from SC 2.2. HA NFS in SC 3.0 is not scalable, in that two nodes cannot serve the same NFS file system concurrently. In subsequent 3.x releases, scalable HA NFS support will be provided. The SC 3.0 architecture, through the PXFS file system, can provide an excellent foundation for a scalable, highly-available NFS server, but schedule considerations preclude providing this scalable support in the initial release.

3.1 Cluster Programming

A cluster is a collection of computer nodes that collectively provide services to the users. The services are usually data (i.e. I/O) oriented: a database server, file server, or a WWW server. Each node runs one or several server processes which are usually internally multi-threaded. These processes may communicate between each other. For such services the cluster looks as if it were a single computer. For other services the cluster exposes the individual nodes and does not present a single view of the system.

An external user of a service that has a single system image usually cannot tell a cluster from an SMP system. However, there are important situations when it is desirable that a user connects to a particular node and the cluster should support such situations.

The programming model for a cluster is somewhere between that for an SMP and that for a LAN. While some API calls are the same as on SMP (for example file access), others are message based (for example message-based communication between the application's components). This is consistent with the needs of most cluster-ready applications that want to use single system image when appropriate, and take advantage of resource locality for performance or HA reasons, at the same time.

In addition to the existing HA APIs, SC 3.0 may provide:

- Wolfpack-like high-availability APIs
- Java system clustering API
- Transparent cluster support for Enterprise JavaBeans APIs
- APIs for constructing scalable system services
- VIA API for DBMS and HPC high-speed communication

It is important to emphasize that the existing Solaris ABI will be preserved in SC 3.0. Section 11.0 describes in more detail the APIs supported by SC 3.0.

4.0 SC 3.0 Architecture

Figure 2 shows a high-level view of SC 3.0: the cluster is a collection of nodes connected through a high-speed interconnect, where the nodes share a global cluster file system, with global scalable access to LAN network interfaces. The cluster is managed centrally from a browser-based interface. From outside the cluster, network clients view the cluster as a set of highly-available services. Applications running in the cluster can still view the cluster as a distinct set of nodes, yet at the same time have seamless access to global file system, devices, and network resources.

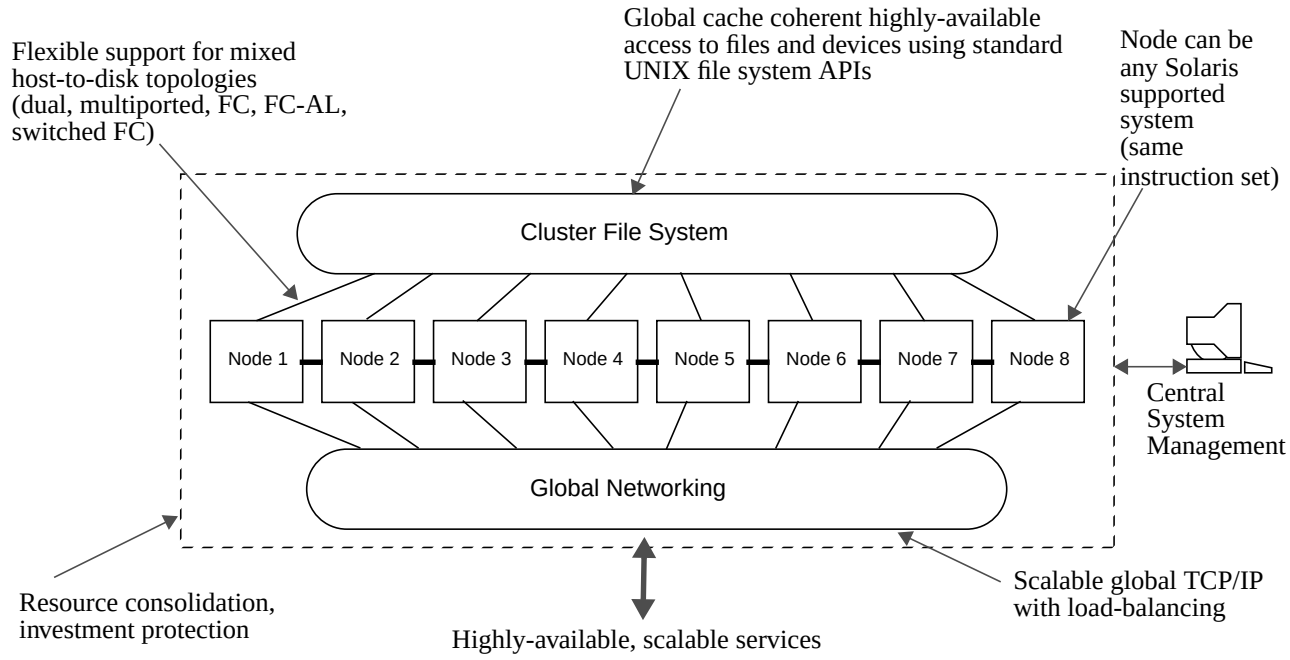


Figure 2. SC 3.0 high-level view

SC 3.0 has an availability goal between 99.99% to 99.999% of uptime (which translates to an annual downtime of 55 minutes to 5 minutes).

As shown in Figure 3, the software in SC 3.0 is composed of several components connected through interfaces. These components work together to provide the illusion that a collection of loosely coupled nodes form a coherent single system. Each of the components provide a set of features that are explained in the following sections.

4.1 User-visible Components

The major end user-visible components are:

- Global device access, with alternate path discovery in case of failures (section 8.0).
- Cluster-wide distributed UNIX file system with near-continuous access to data (section 9.0).
- Scalable global networking, with load-balancing as well as IP failover in case of failures (section 10.0).
- A rich set of clustering, communication, load-balancing and management APIs (section 11.0).
- Support for all SC 2.2 features, data services, and APIs (section 12.0).
- Browser-based system monitoring and control.
- Webstart browser-based system installation.

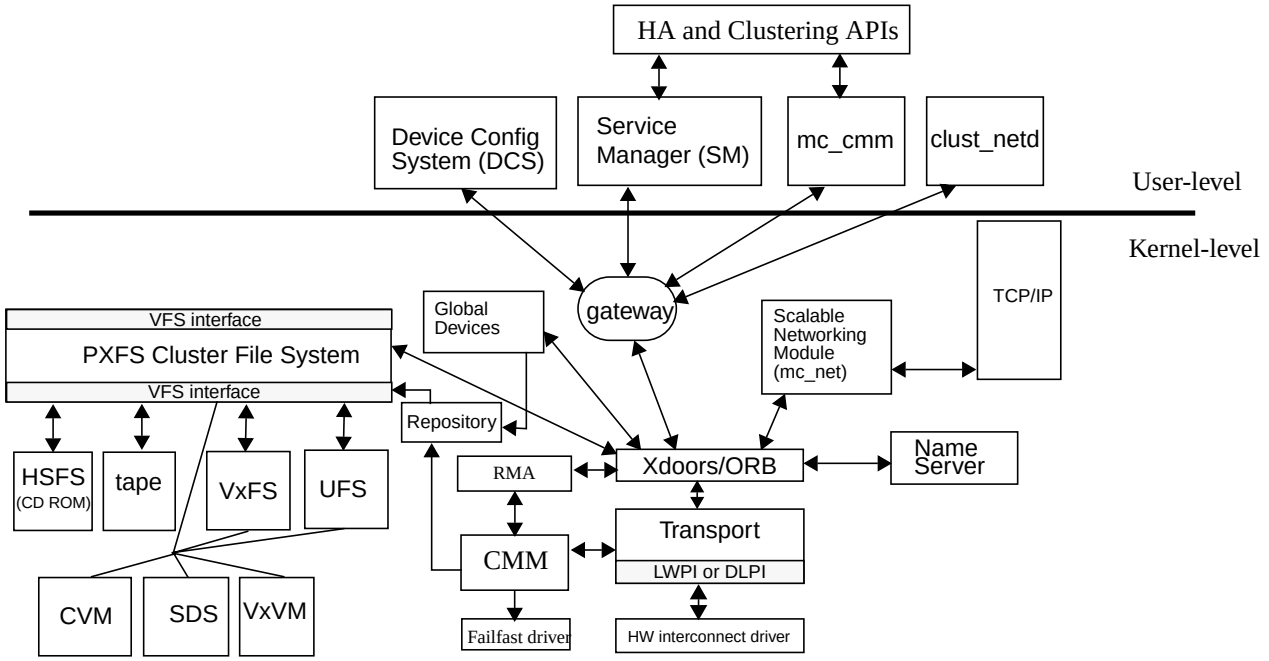


Figure 3. SC 3.0 per-node software components

4.2 Clustering Infrastructure

Clustering infrastructure not directly seen by the user includes (Figure 3):

- A high-speed cluster *transport* mechanism (section 5.2).
- A *communication* mechanism, with cluster-wide resource management and reclamation in case of failures (section 5.3).
- A *high-availability* and *replica* management system (section 6.0).
- A cluster *membership* monitor and a *failfast* driver (section 6.3).
- A *service manager* that coordinates the various services in the system (section 7.1).
- A low-level data *repository* that is used to store system configuration data before the file system is mounted (section 7.2).
- Fully-versioned internal interfaces and protocols to allow *rolling upgrades* to post 3.0 releases.

Nodes may be added to the cluster dynamically (hotplug) or nodes be cleanly shut down without affecting applications so that they might be maintained or deleted from the cluster. SC 3.0 allows support of a “rolling” cluster software upgrade to post SC3.0 releases where new versions of the operating system may be installed and tested *without bringing down the entire cluster.*[3] This allows an administrator to bring down nodes, install

new versions of SC 3.x on them and then let them rejoin the cluster instead of statically partitioning the cluster.

4.3 Security and Trust Among Cluster Codes

The SC 3.0 system assumes a trusted computing base among the nodes of the cluster. Nodes are connected to a private interconnect, and kernels are trusted entities. Security issues concerning authentication and authorization between kernels can thus be avoided.

SC 3.0 also does not provide any mechanism that allows a node to run independent of the cluster[4], nor can a foreign (non-SC 3.0 based) kernel join the cluster.

Although the system does not guard against rouge kernels or Byzantine failures, the software has a number of measures to safeguard against the failure of a node affecting the rest of the system:

- The system never shares any kernel data in shared memory between the nodes. Above the transport, all communication is message-based with well-defined failure boundaries.
- The transport software uses (incoming and/or outgoing) MMU mappings, if provided by the interconnect transport hardware, to guard against a kernel inadvertently clobbering another kernel's memory.
- Each node has a failfast driver that must be periodically contacted by the CMM or it causes the node to shutdown.
- The CMM algorithms and timeouts are carefully designed to insure that a node will actually be down (either because its CMM or failfast driver shut it) when it is declared down by the rest of the cluster.
- The software uses disk fencing, if provided by the hardware, to insure that a node declared down cannot access any data.
- The CMM uses a quorum device, if provided by the hardware, to guard against a cluster splitting into two split-brain clusters that do not know of the existence of the other.

5.0 Cluster Communication Infrastructure

5.1 Overview

Figure 4 provides an overview of the communication architecture. The following subsections provide a bottom-up description of this architecture.

3. Such a feature requires a built-in interface evolution strategy (section 5.3).
4. A node can be booted independently from the cluster in a restricted "maintenance" mode.

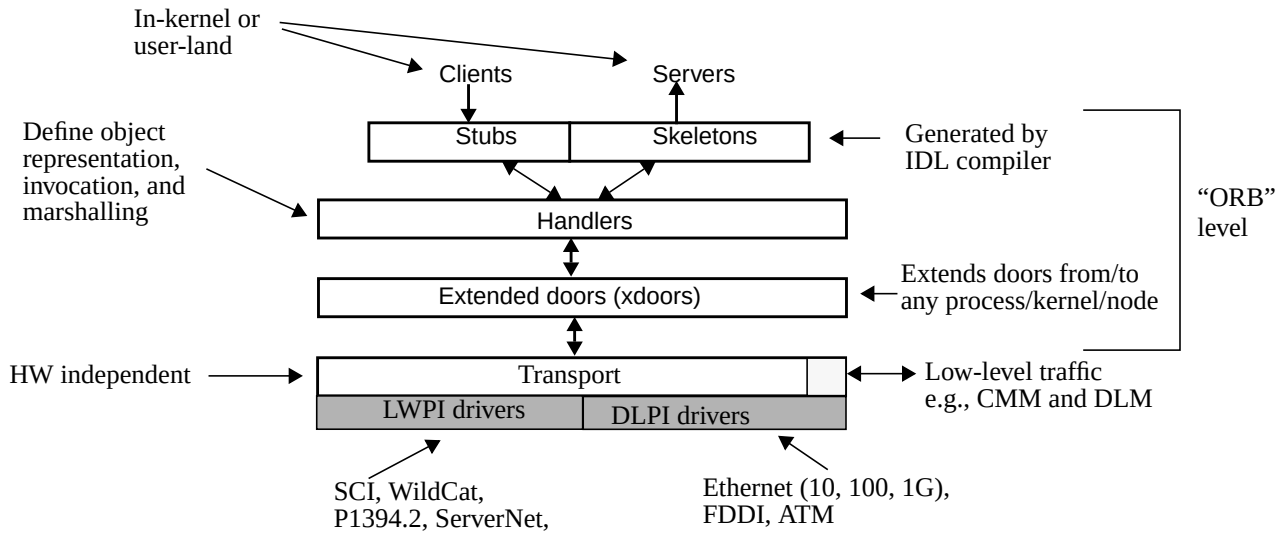


Figure 4. SC 3.0 Communication Architecture

5.2 Transport

5.2.1 Interconnect independent portion

The bulk of the transport is hardware and interconnect independent. The transport layer defines the interface that a transport has to satisfy to be used by the xdoor layer. The transport implements reliable *sends* of arbitrary length messages. It is also possible to register receive functions with a transport. A transport has a set of buffer pools with lists of buffers, waiting to be filled by incoming messages. Arriving messages are stored in buffers from the buffer pools specified in the message's headers. The transport's interface is both simple and sufficiently powerful to support highly efficient object invocation, providing message delivery with scatter and gather capabilities through the buffer pools.

In summary, the transport provides:

- A simple one-way send/receive interface that is use by the xdoor layer.
- Support for data marshalling that can use the features of LWPI drivers (see below), if any, to provide hardware optimized data movement between the nodes (e.g., zero-copy or fast PIO for small writes).
- A low-level interface for specialized clients. The CMM in particular has a separate transport channel that is higher priority than the rest of the transport traffic.

5.2.2 Interconnect dependent drivers

The transport relies on interconnect-specific drivers to manage the actual interconnect hardware. The transport supports two varieties of such drivers:

- LWPI (Light weight Programming Interface) drivers. LWPI is designed for remote store/reflective memory type of interconnects. Examples of such interconnects include SCI, ServerNet, P1394.2, and Wildcat cluster interconnects.

  LWPI drivers offer the most performance for SC 3.0 as they are optimized for cluster communication. LWPI allow the transport to exploit the underlying hardware for zero-copy, for example.
- DLPI. DLPI is the standard interface used for STREAMS-based drivers. There are many DLPI drivers, including drivers for Ethernet, FDDI, and ATM. SC 3.0 supports DLPI drivers for low-end clusters that use Ethernet or FDDI.

5.2.3 Multiple Adaptor Support

The transport architecture allows multiple interconnect connections/adaptors to be used per node. There are two reasons for this support:

- **Redundancy**

  As with other parts of SC 3.0, high availability is a major goal. With multiple interconnect adaptors, the transport hides the failure of interconnect adaptors from the rest of the system.
- **Scalability**

  During normal operation, the transport uses all available adaptors for increased throughput. This is especially important for large cluster nodes, where each node is a large SMP with a high-bandwidth backplane.

5.3 Distributed Object Technology

The Object Request Broker (ORB) underlies much of SC 3.0 communication infrastructure. As shown in Figure 4, the ORB refers to the stubs and skeletons generated by the IDL compiler, the handler level, and the xdoors levels.

The ORB provides a distributed object invocation service similar to CORBA. In fact, its interfaces are defined using the CORBA-defined Interface Definition Language. Distributed object technology provides several advantages:

- Strongly specified distributed interfaces
- Transactional RPC semantics
- Level of indirection for transport and object-specific optimizations
- Built-in evolution strategy
- Little additional overhead in local case (same address space)
- Strongly typed specialization
- Built-in support for heterogeneity

The inter-node distributed interfaces in a cluster must be strongly specified since the clients and servers of an interface may potentially be upgraded at different times. Strong interfaces are useful even in single node systems as they allow clients and implementations of a function to continue to work over multiple releases. This would not be a benefit if the object model added significant overhead to the local case. However, the distributed object model is, in general, a superset of standard local objects and so local invocations don't add significant overhead. In fact a local object invocation is executed using a native C++ virtual function call (9 SPARC instructions). There is some additional space overhead associated with CORBA objects.

SC 3.0 provides:

- An IDL compiler with CORBA-like C++ mapping
- Ultra-fast local object invocation
- Allows optimizations for hardware support of high-speed remote object invocation
- Doors-based object representation calling
- Distributed object reference (xdoor) collection
- Exception and resource fencing support on failure
- Separate object handlers
- Failover object handling and transaction support (sections 6.5 and 6.6)

5.3.1 IDL

Interfaces are defined by using CORBA's *Interface Definition Language* (IDL). IDL allows the definition of interfaces by specifying the set of operations the interface accepts (similar to C function declarations), as well as the set of exceptions any given operation may raise. Interfaces can be composed using *interface inheritance* mechanisms, including multiple inheritance. Client and server object implementation code can be written using any programming language for which a mapping from IDL has been established (in theory C, C++, and Java).

Every major component of the system is defined by one or more IDL specified object types. All interactions among the components are carried out by issuing requests for the operations defined in each component's interface. Such requests are carried out independently of the location of the object instance. When the invocation is local (within the same address space), it proceeds like a C++ virtual function call. When the invocation crosses domains (across address spaces or nodes), the invocation proceeds essentially as an RPC, where the client code uses stubs, and the server (implementation) code uses skeleton code to handle the call.

The stubs used by the client code, as well as the skeletons used by the server code are generated automatically from the IDL interface definition by the CORBA IDL to C++ compiler.

5.3.2 ORB Architecture

The different components of the system communicate using the services of the ORB. The main functions of the ORB are reference counting, marshalling/unmarshalling sup-

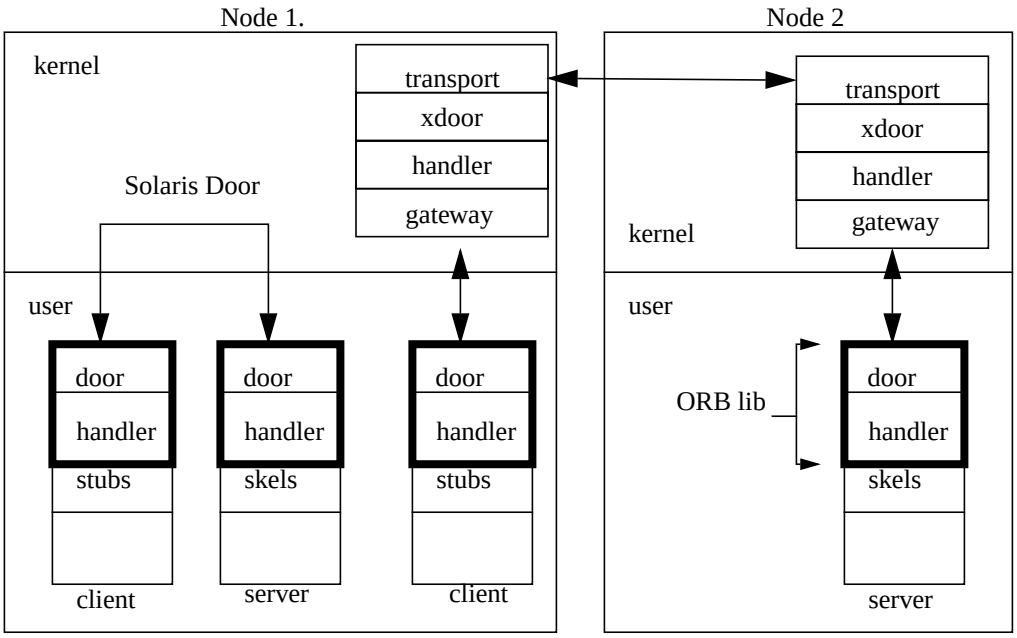


Figure 5. User-level communication through doors

port, remote request support (RPC), and communication fault recovery. The three main goals of our ORB architecture are to provide an efficient object invocation mechanism, easy configuration of clusters, and support for high availability.

The ORB is composed of three layers: the stub/skeleton, the handler, and the xdoor layers (see Figure 4). Each object reference is associated with a handler. The handler is responsible for preparing inter-domain requests to the object whose reference it handles. A handler is also in charge of performing marshalling of its associated references, as well as of local (to an address space) reference counting. The handler layer implements the subcontract paradigm, providing flexible means of introducing multiple object manipulation mechanisms, such as zero-copy.

In order to perform an invocation on its object, a handler is associated with one or more *xdoors*. The xdoor layer implements an RPC-like inter process communication mechanism. This layer extends and builds on the functionality of the Solaris doors mechanism. With xdoors, it is possible to perform arbitrary inter-domain (intra- or inter-node) object invocations following an RPC scheme. References to xdoors are carried out with invocations and replies, permitting the ORB to locate particular instances of implementation objects. The xdoor layer implements a reference counting mechanism for ORB and application structures.

Together, the three layers and the transport, provide support for efficient parameter marshalling and passing, making it possible to implement zero-copy schemes for inter-node communications when the communications hardware supports it.

To support high availability, the xdoor layer never aborts an outstanding invocation unless a failure in the cluster is detected. In that case, outstanding invocations to failed nodes are aborted, and an exception is raised which can be caught by the handler layer, or by the component code itself. Services needing high availability make use of the information in the exception either directly or by means of special handlers (see Section 6.5). In addition to this failure reporting, the xdoor layer implements a reference counting algorithm that can recover after node failures. For efficiency, the algorithm is optimistic in nature, performing most of its work when an actual node failure is detected.

The ORB permits both kernel- and user-level communication. The ORB is implemented as a loadable kernel module to be used by the code residing in the kernel, and as a library, to be used by code executing in a user-level process. Most of the code used in both cases is identical, differing mostly in the xdoor and transport layers.

The user level ORB uses Solaris doors to provide a secure and efficient transport mechanism between address spaces. The gateway, which is an additional layer of software in the SC 3.0 kernel provides the mapping between Solaris doors, and xdoors. This allows a secure means by which user processes on different nodes can communicate. Xdoors are only passed between nodes by trusted entities (kernels). Clients/Servers that reside on the same node use the Solaris doors transport directly, bypassing the gateway.

It is important to note that, in SC 3.0, all IDL/ORB interfaces are private to the system, and are used to build the rest of the clustered system. They are not exposed nor documented to end users.

6.0 High Availability & Replica Infrastructure

6.1 Overview

SC 3.0 delivers true high availability. The functionality was chosen to meet the requirements of DBMS, HA-NFS, and interactive services. The HA capabilities of SC 3.0 exceed those of VMScluster, SC 3.0, HA-NFS, and other “HA products” that are on the market today.

SC 3.0 makes all the components on the path between the user and data on disk highly available for the target applications listed above. Figure 6 illustrates the typical path from the user to the data.

In general, a resource is highly available if it survives any single (software or hardware) failure in the system. A node failure is detected very quickly. After a failure is detected, a new equivalent server for the resource is created on one of the remaining nodes. Our goal is to recover (i.e. make again available) all HA resources of a crashed node in less than 10 seconds, excluding volume manager, database, or application recovery time.

Note that there is no period of time when all resource are unavailable (as it is on the VAXcluster systems during DLM recovery). Resources unaffected by the crash (for example file systems served by the remaining nodes) are fully available during the recovery. Furthermore, resources of the failed node become available as soon as they are

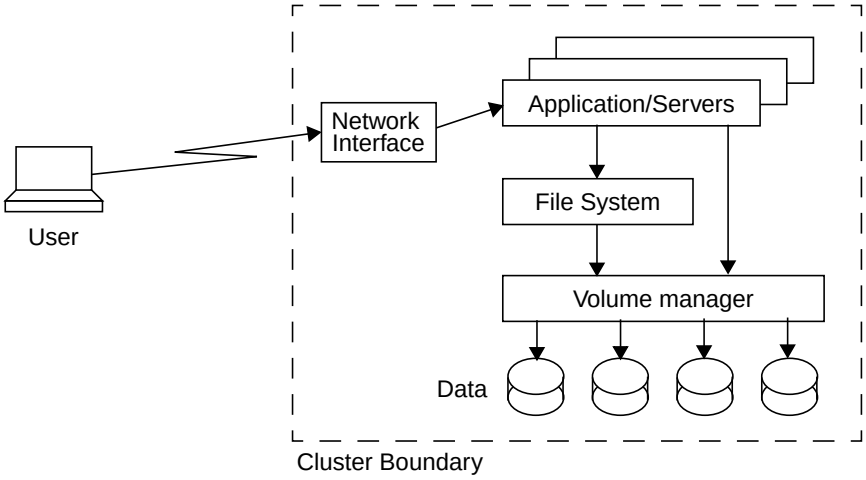


Figure 6. Path between client and data is highly-available

recovered; a recovered resource does not have to wait for all other resources to complete their recovery. For example, a read-only file system that was served by the crashed node becomes available very quickly while a heavily used read-write file system may take several seconds before it becomes available.[5] This high degree of availability is one significant improvement over the existing HA products.

Most of the HA resources are recovered transparently to the applications using the resource. The semantics of resource access is fully preserved across a node failure; the applications simply cannot tell that the resource server has been moved to another node. This is a second significant improvement over the existing HA products which do not fully mask the node failure to the applications (e.g., if HA NFS, or the proposed Veritas

Component	Software Techniques	Hardware Techniques
Application	HA/Clustering APIs, HA framework	
Network (LAN)	Inter- and intra-node adaptor failover	Multiple LAN adaptors
File system	Primary/secondaries, mini-transactions	Dual/multi-hosted disks
Volume	Software RAID1/5	Hardware RAID1/5
Device	HA devices, *did* driver	Multiple paths, switches
Interconnect	HA software transport driver	Multiple adaptors, switches
Node	CMM, *failfast* driver	Multiple nodes

TABLE 1. Levels of failure detection and recovery

5. This example will not be true if both file systems are in the same volume manager disk group.

cluster file system, were used as a cluster file system, creating a file can result in the `EEXIST` error when the request is retried on the secondary server).

Transparent recovery of the file system and disk volumes is a key feature of SC 3.0. A failure of any single node is completely transparent to programs on remaining nodes using the files, devices, and disk volumes attached to this node as long as there is an alternative hardware path to the disks from another node (e.g., dual ported disks or Fibrechannel Arbitrated Loop).

SC 3.0 carries forward support for IP address failover from SC 2.2. If a network interface fails, a backup interface on the same node or another node is automatically configured to bind the IP address. For the new SC 3.0 scalable services (section 10.1), an adaptor failover is transparent for most live TCP connections.

SC 3.0 is a good HA database platform. SC 3.0 provides a fast transport for the Oracle DLM (note that we do not provide a DLM for Oracle). The Oracle DLM and highly available shared volumes are sufficient to make a good scalable and available platform for Oracle Parallel Server. The non-parallel version of Oracle, Sybase and Informix do not require additional support to be HA on SC 3.0 beyond SC 3.0 support.

The fact that the HA support is integrated in the operating system rather than unbundled decreases the potential for administrator's mistakes and lowers the overall platform cost.

To summarize, the above HA features make SC 3.0 cluster an excellent platform for HA applications of DBMS, NFS, and interactive services.

6.2 Failure Detection and Recovery

There are several levels where failures are detected and recovered. Table 1 summarizes the software and hardware techniques used at each level.

6.3 Cluster membership monitor (CMM)

The cluster membership monitor (CMM) maintains distributed agreement on the set of nodes that are the current members of the cluster. It is important to note that reaching a consistent distributed agreement is crucial for closely-coupled systems such as SC 3.0. Since "split-brain" or inconsistent membership cannot be tolerated, the only entity that can decide on the membership is the CMM. For example, the CMM is the sole entity in SC 3.0 that uses timeouts to detect potential failures.

The CMM is a distributed set of agents, one per node. The agents periodically exchange node heartbeat messages. At the core of the CMM is a state automaton. In case of any failures, the CMM enters into a state where a new cluster membership is achieved.

The CMM uses a quorum device, if provided by the hardware/platform, to guard against a cluster splitting into two split-brain clusters that do not know of the existence of the other.

Each node has a failfast driver that must be periodically contacted by the CMM or it causes the node to shutdown. The CMM algorithms and timeouts are carefully designed to insure that a node will actually be down (either because its CMM or failfast driver shut it) when it is declared down by the rest of the cluster.

SC 3.0 modifies the CMM used in SC 2.2. This includes moving the CMM into the kernel, replacing call-backs that are implemented as process fork/exec with IDL call-backs, using interrupt threads to drive the CMM automaton, and defining the interface between CMM and the ORB.

To support the SC 3.0 framework and HA-APIs, a user-level daemon (mc_cmm) is used to execute the SC 2.2 reconfiguration scripts (see section 12.0). The reconfiguration scripts are however expected to be modified to run on SC 3.0.

After every new membership change, a series of steps are executed by the CMM in a global, lock-step fashion. The CMM steps include HA xdoors cleanup, ORB global reference count reclamation, and replica manager call-backs to failover from failed primary servers to new primary servers. The following two sections explain some of the call processing that results from CMM reconfiguration.

6.4 Replica Management

Service high availability is achieved in SC 3.0 by server replication. Replicated servers are called replicas and are usually started on different nodes in the cluster. At any point of time for a given service, one replica of the service is designated as a *primary*, and the rest as *secondaries* or *standbys*.

The primary replica is the current server contacted by clients. A secondary replica is an active standby that is ready at any point of time to become a primary. A standby is a passive secondary that can become an active secondary. A highly available service must have a primary and at least one secondary. For example, a disk array that is connected to four nodes may have a primary file system server replica, two secondaries, and a standby. If the primary fails, one of the two secondaries takes over. If a primary or a secondary fail, the standby is promoted to a secondary to maintain at least two active secondaries.

The replica manager (RM), and a per-node replica manager agent (RMA) coordinate server replication. The replica management system is responsible for:

- Keeping track of the current primary, secondaries, and standbys for each service.
- Promoting one of the secondaries to become primary when the original primary replica crashes.
- Switching back to the preferred primary (if any) after it recovers.
- Activating a standby to become a secondary (and vice versa).
- Blocking calls at the xdoor level when a primary fails, and coordinating the switchover at the xdoor level from the failed primary to the new primary (see section 6.5).

The primary/secondary approach is used for all highly-available servers in SC 3.0. An exception is the RM itself, as it is used to coordinate primary and secondary relation-

ships. The RM is made highly-available by saving its state in a distributed manner in each node as part of the RMAs. In case of the failure of the RM, the remaining RMAs elect a leader and the new RM reconstitutes itself out of the state saved in each RMA.

6.5 Replica Object Handlers and HA xdoor Support

A mechanism is needed to “hide" the presence of replicas to the client code. In an object oriented system, the appropriate level for implementing this transparency is the object handler. The object handler is responsible for marshaling the input arguments, sending them to the object implementation, waiting for a reply, and unmarshalling the output arguments.

Figure 7 shows an example of a replica object handler that encapsulates two xdoors: one to the current primary, and the other to a secondary. The client of the object does not know that the object represented by the object handler is actually replicated.

The replica object handler communicates only with the current primary replica. The primary replica is responsible for updating the remaining replicas. The primary replica is usually responsible for synchronization of requests (e.g. by locking) and for ensuring that all replicas see conflicting operations in the same order.

The protocol between the object handler and the replicas achieves the exactly once semantics of object invocation even in the presence of a failover of the primary replica during the invocation. A service is available as long as at least one replica remains online.

When a primary fails, the RMA components of the replica management system coordinates with xdoors to block any calls that are in progress at the time of the failure. After electing a new primary, the RMA releases the blocked calls which are then retried to the newly elected primary. The operation is guaranteed to produce exactly-once semantics because it is executed as a transaction.

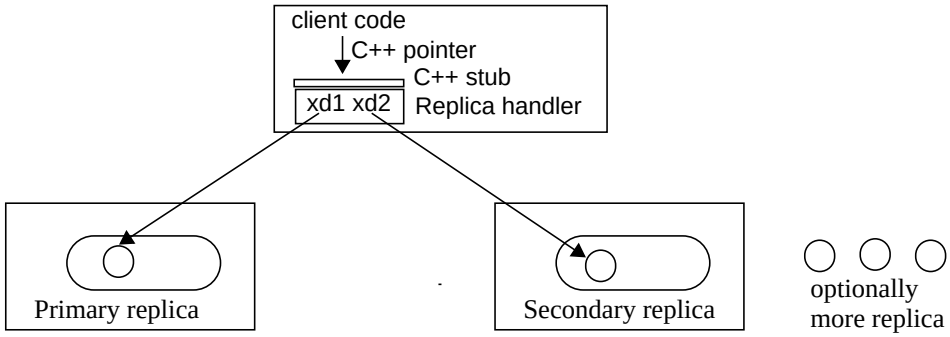


Figure 7. Replica object handler

6.6 Mini-transactions

To achieve exactly-once semantics, most servers in SC 3.0 are structured as primary/secondary replicas that use mini-transactions, including various objects in PXFS, device management, the name server, and scalable networking support. This is a significant differentiator of our system: the system is not only highly-available, but also provides continuous operation, masking the failures from the applications.

A replica object handler communicates with the replicas to achieve atomic execution of client requests. The primary replica for a given service sends checkpoint messages to

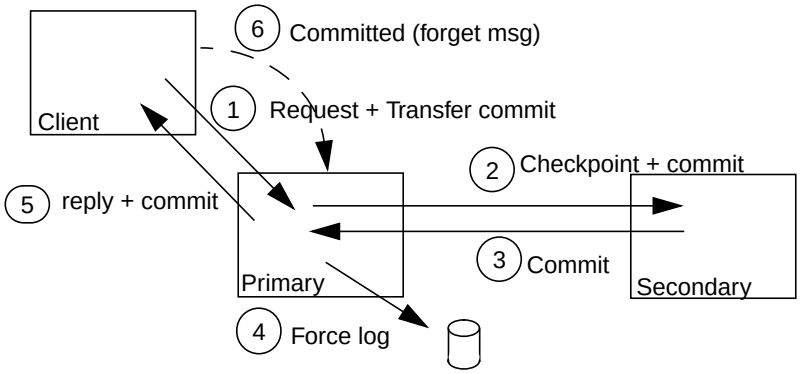


Figure 8.

A mini-transaction. The client request (1) implicitly transfers the coordinator responsibility to the primary. The primary obtains the necessary locks and effectively performs the operation. A checkpoint message (2) is sent to the secondary. The secondary replies with a commit message (3). The primary forces the log (4) and replies to client (5). Using the asynchronous object unreference mechanism, the client eventually replies with a committed message (6), and the primary and secondaries forget the transaction.

each of its secondaries. The exact state transferred in these messages is service-dependent. Figure 8 shows the sequence of operations on a typical replicated object

Since the UNIX kernel does not support transaction semantics, the use of mini-transactions is hidden from the rest of the kernel. For example, in PXFS all failover and transaction handling is done below the vnode layer. An error, such as EIO, is propagated through the vnode layer to the UNIX kernel only after PXFS fails in containing the error, either because the primary and all its secondaries failed, or because the operation was not conducted on a highly available file system.

Mini-transactions are based on traditional database transaction concepts, with some restrictions. In particular, a replica (primary or secondary) may not call other highly-available objects, and no nesting of mini-transactions is possible. We sacrifice generality for performance, as mini-transactions are highly-optimized for the basic call sequence depicted in Figure 8.

7.0 Other system services

7.1 Service Manager

The service manager provides global coordination and group membership support for system services. A *service* is any entity in the cluster that wants to be coordinated by the service manager. A service consists of one or more *servers* (a server can be part of the kernel or a user-level daemon).

The service manager provides means for registering services and servers, notification of service membership changes, means for electing service leaders, and a mechanism for global synchronization among servers.

7.2 Repository

A low-level cluster-wide repository is used to store system configuration information. There are two types of information stored in the repository: global device information, and CMM-related configuration information. This information is accessed very early on during the kernel boot sequence, and before membership is established. Therefore, the repository is a very specialized data store, and a not a general-purpose data repository. The repository is needed to boot-strap the system and to mount the global file system. Once the global file system is mounted, the file system can be used to store general configuration information as regular files.

8.0 Global Devices

8.1 Overview

SC 3.0 maintains the independence of the individual nodes, but provides at the same time global highly-available device access for physical devices such as disks, tapes, and cd-rom drives.

8.2 Device Name Space

Each node in the cluster maintains its own local device name space on its individual root file system. In addition, the global file system maintains the global device name space. Highly-available devices are accessible from more than node, and are therefore also represented by a logical name that always provides access to the device, as long as there is a hardware path to the device.

The device name space is configured used the following rules:

- Local names are accessible from the local node only. They maintain their familiar form, but add a node id in the /devices path name. For example,
  /dev/dsk/c0t3d0s7 -> ../../devices/nodeid/iommu/.../sd@3,0

- Highly available devices can be reached from more than one node (e.g., dual or multi-pathed disks). Such devices are represented by global, highly-available logical names that includes the device id (*did)* driver name, for example:[6]

  /dev/did/dsk/...

  These highly-available global names are the ones normally accessed by applications, and are used with volume managers and global mounts.
- Each device that is accessible globally can also have global names that describe each possible physical path to the device, for example:

  /global/nodeid/<local name of device on node> -> ../../devices/nodeid/.../sd@3,0

  Such global physical names are used to access single-ported global devices, and for diagnostics and maintenance purposes.

SC 3.0 assigns global minor numbers to all physical storage devices in the cluster (disks, tapes, cd-rom). This results in a consistent view of all device accesses in the cluster. For example, issuing a stat call on a device returns consistent information, regardless of the location of the device or the issuing application.

The advantages of this naming scheme include:

- Each node remains fairly independent, with little change in the device administration model.
- Devices can be selectively made global.
- Third-party link generators continue to work.
- Given a local device name, an easy mapping is provided to obtain its global name.

8.3 Device High Availability

Access and I/O operations to global devices are highly available as long there is a hardware path to the device:

- The device is always accessible under the same global name from every node.
- For disk-like devices, raw reads and writes to the devices are guaranteed to succeed even when a node on the path to the device fails. I/O to other types of devices, such as tapes, will return an appropriate error (e.g., EIO) if an error is encountered while accessing the device (e.g., if a node on the path fails). The device, however, can always be opened again and accessed using the same global name.

8.4 Global Device Management

8.4.1 Cluster formation and alternate path discovery

Following the Solaris model, a node may boot using the -r flag. When this flag is used, new devices are recognized and configured into the system. Hot-plugging of new devices is also supported (see below).

6. These are only examples. The exact format of the names are TBD.

Whether or not the -r flag is used, SC 3.0 discovers all multiple paths to devices as part of the boot process. Nodes can join the cluster incrementally and alternate paths can be added dynamically.

8.4.2 Hot-plugging

SC 3.0 supports the hot-plugging of new global devices. When a new devices is added, the Solaris kernel recognizes the new device and a new path name is configured.

The act of adding a new device may result in an alternate path to a previously non-highly-available device. Therefore, devices that *can* become dual/multi-ported should always be configured and accessed through their logical names (e.g., /dev/did/dsk/..).

8.5 Restrictions

Memory-mapped (mmap) access to remote devices is not supported.

SC 3.0 supports global access only to physical devices, such as disks, tapes, and cd-roms. This restriction will be relaxed in the future.

9.0 Cluster File System

9.1 Overview

SC 3.0 provide a cluster file system based on the Solaris MC proxy file system (PXFS) file system. PXFS has the following features:

- It makes file accesses location transparent—a process can open a file located anywhere in the system and processes on all nodes can use the same pathname to locate a file.
- The cluster file system uses coherency protocols to preserve the UNIX file access semantics even if the file is accessed concurrently from multiple nodes.
- PXFS provides extensive caching for high performance using the caching approach from Spring, and provides zero-copy bulk I/O movement to move large data objects efficiently.
- PXFS provides continuous access to data, even when failures occur. All file system operations are executed as mini-transactions (Section 9.4), and are retried atomically in case of failures. Applications do not detect failures as long as there is a path to disks that is still operational. This guarantee is maintained for raw disk access, normal file system operations, and even for complex file system operations, such as file renames.
- PXFS is independent of underlying file system and volume manager. PXFS makes any on-disk file system global.[7] See Figure 9.

7. This is true for any *reasonable* file system, where reads return the last written value.

SC 3.0 supports the UFS, VxFS, and HSFS file systems, as well as SDS, VxVM and CVM volume managers. The exact combinations of file system-volume manager, and whether multiple volume manager types will be supported in the same cluster is TBD.

- PXFS is built on top of the existing Solaris file system at the *vnode* interface. This interface allows PXFS to be implemented without extensive kernel modifications.[8]

The rest of this section discusses these features of PXFS in more detail.

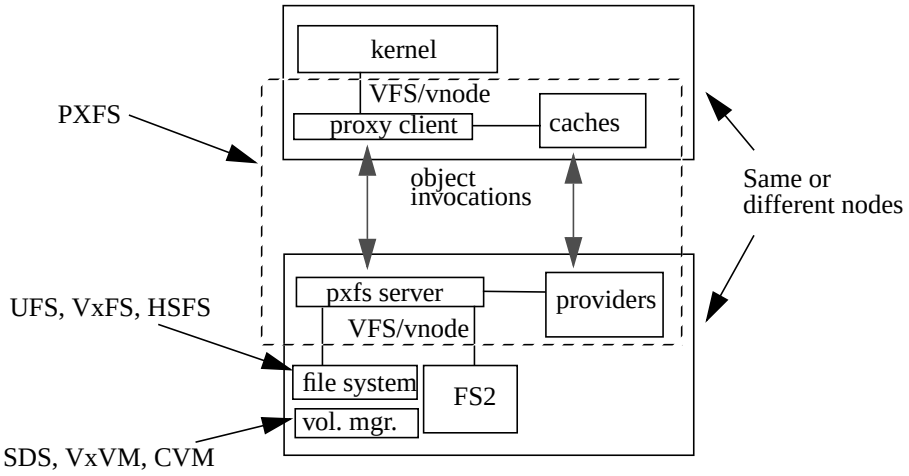


Figure 9. PXFS is a VFS-based proxy file system between the kernel and underlying file system. PXFS is invisible to user-applications, which see the underlying file system only.

9.2 File System Name Space

The Solaris Cluster is centered around a shared view of the file system. In general, all nodes should see a consistent image of *global* files, directories, devices, and other operating system resources that are accessible through the file system. A process can open a global file or a global device anywhere in the system and use the file with the familiar semantics of local file access.

A deliberate design decision is to hide the notion of PXFS as a distinct file system type. Instead, clients see the underlying file system (e.g., UFS) and not PXFS. When mounting a global file system, the traditional mount command is used, with a new global mount flag, *but with the file system type and file system specific flags of the underlying file system* (not PXFS). In essence, PXFS is a transparent proxy between the distributed kernels that provides a global view of the mounted file system.[9]

8. For performance reasons, three new vnode operations will be introduced into Solaris to allow efficient data movement between PXFS and on-disk file systems.

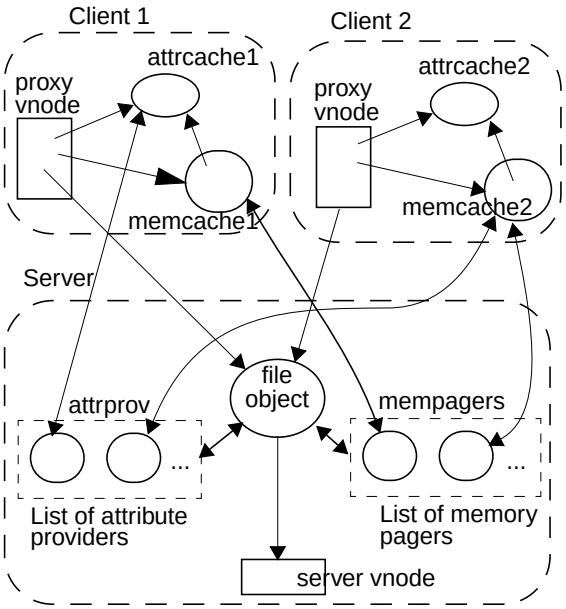
SC 3.0 architecture does not dictate a particular policy as for the names of the global file systems, or as to the exact devices that are exported globally.

Each node in the cluster still maintains its own root and other file systems. This becomes increasingly important if a cluster must support different kernel binaries, different versions of the operating system, or even heterogeneous machine architectures. In these cases flexible manipulation of node names spaces is required.

9.3 Cache Coherent Cluster-wide File System

PXFS interposes on file operations at the vnode/VFS interface and forwards them to the vnode layer where the file resides, as shown in Figure 10. Besides files, PXFS also provides access to other types of vnodes, such as directories, symbolic links, and devices. In SC 3.0, PXFS provides global access to a subset of all possible vnode types. Because PXFS is built on top of the existing file system, it can leverage off the existing file system code. This is an important difference from distributed file systems such as Sprite or Spring that rewrite the entire file system, or other systems that depend on adding distributed lock management to an existing file system (see section 9.6 for comparisons to other approaches).

Figure 10. **A simplified picture of file Paging and Attribute Caching**



Each client file has a memcache object to cache data and an attrcache object to cache attributes. The server has corresponding mempager and attrprov objects to provide the data and attributes. The file object is an IDL object implementing the file protocol. The server vnode provides the underlying file storage.

PXFS uses extensive caching on the clients to reduce the number of remote object invocations. Figure 10 shows a simplified picture of some of the objects used in the file paging and attribute caching protocols. The design of PXFS was influenced by the Spring

9. File system specific ioctl calls pose an interesting challenge. Normally, such ioctl calls are passed directly to the underlying file system. Some ioctl calls, however, may require special handling by PXFS to provide global semantics.

file system and its caching architecture. A client cache is implemented through a *cached* object on the client to manage the cached data and a *cacher* object on the server to maintain consistency. For data, the client has a *memcache* object and the server has a *mempager* object. For attributes, the client has a *attrcache* object and the server has a *attrprov* object.

As an example, suppose a process on Client 1 wishes to page in a page from a file. A *memcache* is a vnode in addition to being an IDL object, so it can accept `GETPAGE` and `PUTPAGE` operations from the Solaris virtual memory system. The memcache vnode is used as the paged vnode for the `VOP_MAP` operations on the proxy vnode. Memcache searches the local cache for the page. If it is not available, memcache requests the page from the associated mempager. The mempager checks the other mempagers to see if another client has the page, to maintain consistency. Finally, the page is obtained from the backing server vnode. Thus, PXFS has control over global page coherence.

The PXFS coherency protocol is token-based and allows a page to be cached read-only by multiple caches or read-write by a single cache. If a dirty page is transferred from one node to another, it is first written to the stable storage on the server to avoid losing updates due to crashes of unrelated nodes. Similarly, an attribute cache is also protected by a reader-writer token. The token is also used to enforce atomicity of read/write system calls on regular files. Token management is integrated with data transfer for better performance.

Directory caching and caching of ACLs (if provided by the underlying file system) is done in a fashion similar to attribute caching. Directory operations that create or remove objects are implemented as write-through to be reflected synchronously in stable storage on the server.

PXFS has a “bulkio” object handler to perform zero-copy transfers between nodes of large data (file pages, uioread/uiowrite data) if the hardware interconnect has sufficient support. For example, if a process takes a page fault, it allocates a page in the local cache and invokes the *page_in* method on the mempager. The server then allocates a kernel buffer and reads the data from the disk into the buffer. The data is then transferred using the bulkio handler directly into the page on the client. If the underlying hardware supports shared memory, the server can map the client page and read data from the disk directly into the page without the need for an intermediate buffer on the server. By using a separate handler for bulk I/O, no changes to the PXFS client or server code are necessary to port PXFS to a different interconnect; only the bulkio handler has to be ported to take full advantage of the hardware.

9.4 Continuous File System Access

PXFS provides continuous access to file system data, even in the face of failures. Dual or multi-ported disks are required for this features. PXFS uses the mini-transaction support provided by the system infrastructure to hide failures and to provide exactly one-semantics to its clients (Figure 11).

During normal operation, each file system has one primary and at least one secondary, with each replica running on a different node accessing the disks through a different

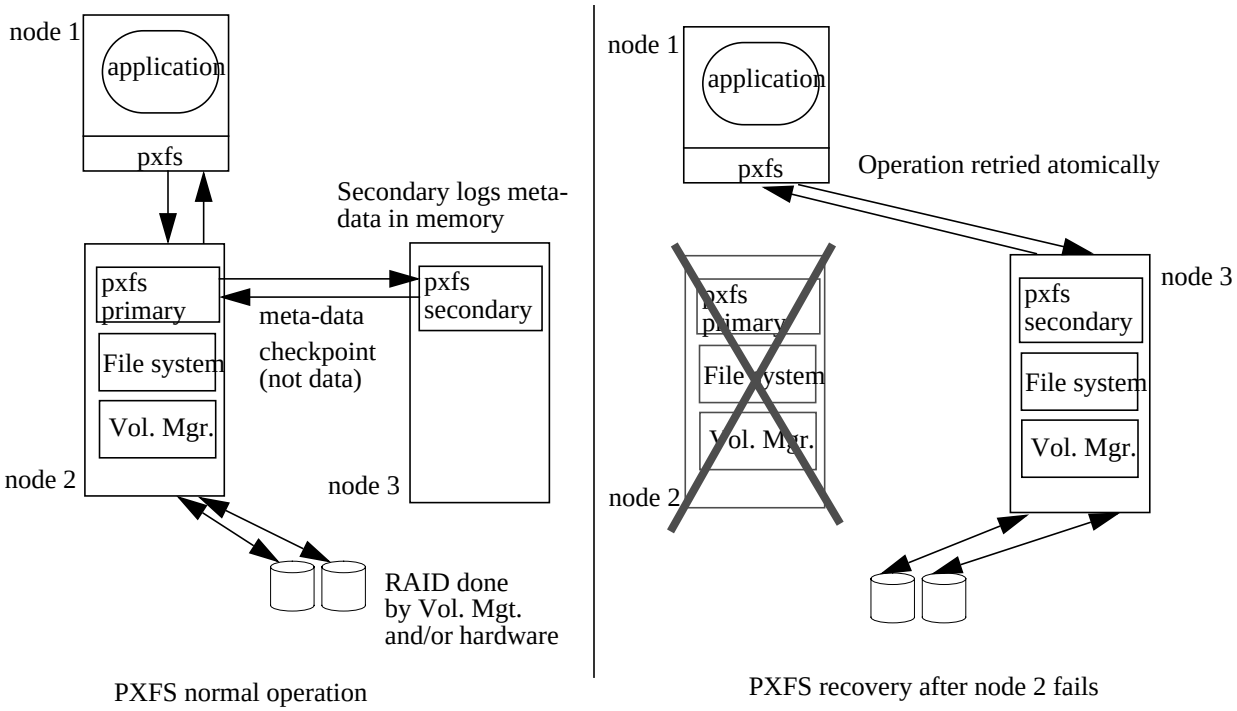


Figure 11. PXFS use of mini-transactions to provide continuous access to files

path. PXFS provides standard UNIX file system semantics. Therefore, reads and writes are normally cached at the client. During a file sync operation (*fsync*) or when the client node is paging data out (e.g., every 30 seconds), the client node writes its data out to the primary. A failure before an fsync follows the normal UNIX semantics: writes maybe lost. After a successful fsync, however, data is highly available. Other file system operations, such as renames are written through atomically to the primary directly, and follow the mini-transaction path shown in Figure 11.

9.5 Optimizations for PXFS

- Data, attributes, directories, ACLs cached coherently
- "Klustering" and async I/O between client and server components of PXFS
- Direct disk transfers when allowed by hardware

9.6 Comparison to Other Approaches

Three possible approaches to providing a global file system are:

- Design and implement a new global file system with on-disk access and device access from scratch.
- Take an existing file system (e.g, UFS) and make it global. Basically, take a shared memory program and add distributed locks using a DLM. Do the same for device access at some level (e.g., specfs).

- Provide a proxy layer at the VFS interface that globalizes any VFS-based file system. This is the approach taken by PXFS.

The advantages and disadvantages of each approach is summarized in Table 2.

Approach	Advantages	Disadvantages
Implement new cluster file system from scratch	• "Do it right" • Optimal local on-disk and global access	• Very costly, both in terms of time and effort • Force customers to adopt yet another file system • Need to fit with or implement volume management • Need a separate mechanism for device access • Need a separate mechanism for other vnode types • Need new administration procedures and backup tools
Add DLM calls to existing file systems	• May be able to provide direct disk access in hardware shared-everything topology	• Need to extend every file system to be made global, including VxFS, UFS, HSFS, etc. We don't own VxFS and UFS may not be worth the investment • Need to rewrite/restructure internal locking of each file system or performance will suffer. • DLM traffic and lock granularity overhead may offset any direct disk advantages • Not clear how to provide highly-available access to files similar to what PXFS guarantees • Need a separate mechanism for device access • Need a separate mechanism for other vnode types
Provide a proxy layer at VFS	• Make any underlying file system global (e.g., VxFS, UFS, HSFS, UDF) • Work with any volume manager • Continuous file system access even when failures occur • Global access to all devices, including pseudo-devices • Can provide direct access in hardware shared-everything topologies • Key to single system image	• Need to qualify the proxy layer with underlying file systems • Need support from underlying file system and/or volume manager for direct access to every disk

TABLE 2. Advantages and disadvantages of three possible ways to build a cluster file system

10.0 Cluster-wide Scalable Networking

10.1 Overview

A goal of SC 3.0 is to provide a platform for scalable network services, yet enable all existing networking applications to continue to work.

In SC 3.0, each node in the cluster retains its own IP address(es) and can be addressed separately from the LAN. Normal Solaris networking semantics and APIs/ABIs apply to each node in the cluster. An application that follows the normal Solaris networking interfaces does not span node boundaries, and it is called a *localized* service.

SC 3.0 adds the notion of *scalable* services that span node boundaries and are used to provide scalable network services.

SC 3.0 extends support for highly-available LAN adaptors/addresses for localized services from SC 2.2, to also include adaptors and addresses used by scalable services. For the new scalable services, an adaptor failover between nodes is transparent for live TCP connections that are not served by the node where the adaptor failed.

Network clients of SC 3.0 do not distinguish between the different types of network services. However, since applications on remote hosts are able to access scalable services on the cluster by identifying the IP address of a scalable service (see below) rather than being required to identify a particular node of the cluster, network clients benefit from scalable services transparently. No changes to any network client is required (nor desired) to use SC 3.0.

10.2 Definition of a Scalable Service

A scalable service consists of:

- A set of processes, where each process runs on a different node.
- A protocol type (TCP or UDP) and a port number used by the service.
- A set of one or more global IP addresses.

An API will be defined to specify scalable services (e.g., protocol, port, processes that make up the service). Whether to make this API public for SC 3.0 is still TBD.

A scalable service is defined such that any process that is part of the service may accept TCP connections (or UDP packets) sent to the global IP address(es) of the service. New TCP connections are forwarded, transparently, by the system to one of the processes in the service. All subsequent TCP traffic for a given connection is delivered to the same process.

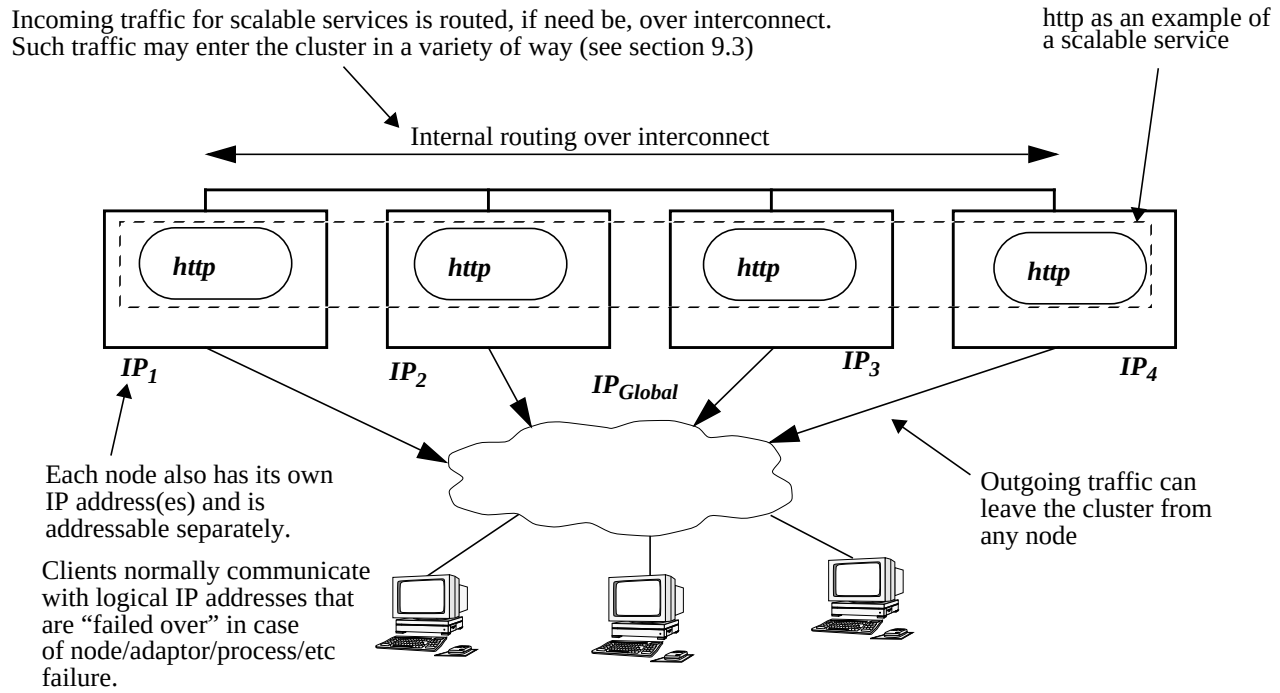


Figure 12. Cluster networking in SC 3.0. Both *localized* (physical or logical) per-node services and *scalable* services are supported.

10.3 Handling Network Traffic for Scalable Services

The SC 3.0 architecture allows for incoming traffic into the cluster to be handled in different ways. The methods differ in performance characteristics and cost of implementation:

- Global IP addresses (IP_G) is assigned to a particular physical interface (node) at a time. In this configuration, all incoming traffic enters the cluster at the node where IP_G is configured.
- IP_G is assigned to a switch that allows traffic "trunking". Trunked traffic from the switch enters the cluster at each node that has an attachment to the switch.
- IP_G is assigned to a virtual subnet, where each node in the cluster acts as a router to this virtual subnet. Traffic can therefore enter the cluster from any node.

The first method will be the default way for handling incoming traffic. It is still TBD if the other two methods will be supported in the final product. Regardless of the method used for incoming traffic:

- Incoming traffic is inspected at the point of entry, and is forwarded to a different node if the local node is heavily loaded (see section 10.4 below).
- Outgoing traffic can leave the cluster from any node.

10.4 Networking Components

There are two major components for cluster networking: a per-node *mc_net* module and a highly-available *clust_netd* daemon.

The mc_net module acts both as a STREAMS module and a distributed component that communicates with other mc_net modules in the cluster over the interconnect. The mc_net module has three major functions:

- It determines if packets received from the LAN should be forwarded to another node or sent up the IP stack on the local node. Packets are forwarded to another node either because the local node is overloaded or because the connection is already established at another node.
- It encapsulates a load-balancing algorithm and a dispatch table that are used when forwarding packets to another node over the interconnect. Optionally, it may encapsulate a packet filter that can be used to make other more sophisticated routing decisions.
- It is used to provide virtual proxy network IP interfaces that correspond to physical network interfaces on other cluster nodes.

The clust_netd server is used for maintaining the dispatch table in each mc_net module, and for initialization and shutdown. As with other servers in SC 3.0, clust_netd is made highly-available through replication.

10.5 Load-balancing Algorithm

The details of the load-balancing algorithms are TBD. In general, the local node will be preferred.

10.6 Advantages of Scalable Services

SC 3.0 has several advantages over third-party solutions that attempt to provide scalable network services through external switches or nodes:

- No external switch is needed. SC 3.0 effectively uses the nodes of the cluster as switching elements and is more cost-effective especially for low-end clusters.
- SC 3.0 is able to handle fluctuations in incoming traffic load, even when an external switch is used. When an external switch is used, SC 3.0 will not defeat the load-balancing decisions of the switch.
- Failures can be detected and services restarted quickly. This is in contrast to external load-balancing schemes that use crude ad-hoc methods for detecting node failures.
- Support for cluster-aware self-balancing applications. Each node in the cluster maintains its own IP address and is accessible as a distinct node.
- Up-to-date load-information is exchanged among the cluster nodes, thus facilitating accurate load-balancing decisions. Third-party solutions generally use simple round-robbin scheduling or depend on crude load information.
- Using a packet filter, SC 3.0 can vary the dispatch and load-balancing decisions based on information such as the source and type of network requests.

- SC 3.0 avoids drawbacks of round-robin DNS, such as naive load-balancing algorithms, and stale DNS cache entries in network clients.
- Finally, the scalable services in SC 3.0 are backed by the cluster file system with full UNIX semantics, and continuous data availability. The fact that all nodes have network access to the file system eliminates the need for ad-hoc application-specific replication.

11.0 Cluster APIs

11.1 Overview

Each node in SC 3.0 is a full-fledged Solaris system, with full Solaris ABI/API support. In addition, the file system and device Solaris ABI/API is supported throughout the cluster. The architecture of SC 3.0 strives to maintain the UNIX model of system APIs and administration commands throughout the cluster.

In addition, SC 3.0 provides a rich set of clustering APIs:[10]

- SC 2.2 HA APIs
- Wolfpack-like APIs with scalability extensions
- Enterprise JavaBeans, with automatic transparent cluster support for applications written to Enterprise JavaBeans
- Java system clustering API
- VIA API for DBMS and HPC high-speed communication

SC 3.0 does not provide a DLM (Distributed Lock Manager, ala VMS/DLM). Instead, transport APIs are available that can be used by an add-on DLM. Note that this approach is consistent with industry trends. In particular, Oracle Parallel Server (OPS), which is the most important application that uses a DLM, now provides its own DLM.

11.2 SC 2.2 HA APIs

SC 3.0 supports all HA APIs from SC 2.2.

11.3 Wolfpack-like APIs with scalability extensions

11.4 Java APIs

11.4.1 Overview

SC 3.0 supports two sets of Java APIs:

- Enterprise JavaBeans (EJB) for *application* programming

10. With the exception of the HA APIs, not all features are committed yet for the final product.

- Java Clustering programming interfaces (JCPI) for *system-level* Java programming

Enterprise JavaBeans (and more generally JavaBeans) constitute the Java component architecture. Enterprise JavaBeans are a standard part of the Java API.

Java Clustering Programming Interface (JCPI) is a new interface that fills a void in the spectrum of Java interfaces: a server-side interface that allows system applications to exploit the high-availability and scalability of computer clusters. JCPI is not an application-level API. JCPI is used to implement system-level applications, such as the transaction manager needed to support Enterprise JavaBeans.

11.4.2 Goals

- Enterprise JavaBeans applications, and Enterprise JavaBeans themselves must run as is on SC 3.0, with no modification.
- Enterprise JavaBeans applications will *automatically* and *transparently* become highly-available when serviced by SC 3.0.
- JCPI is 100% Java Pure and therefore can run on any platform, and is not specific to any particular OS.
- There should be no overlap between JCPI and existing Java APIs. Existing Java APIs are used whenever possible (e.g., component architecture, RMI, naming services, etc.). When appropriate, existing Java APIs maybe extended to clusters (e.g., system management).
- Support for Enterprise JavaBeans need not depend on JCPI. In particular, SC 3.0 should be able to ship support for Enterprise JavaBeans without shipping JCPI.

11.4.3 System-level vs. application-level programming interfaces

A fundamental aspect of the JCPI is that it is intended for system-level cluster-ware programming and not user application programming. End user applications are written using the Java Beans component architecture.

The advantages of this approach are:

- Leverage all JavaBeans applications. The same Java applications will run unmodified on the cluster.
- Leverage application builder tools that support JavaBeans.
- Hide clustering details from the application. Enterprise JavaBeans hide distribution, multi-threading, and transaction management from the application writer. By extending the run-time of the Enterprise JavaBeans, clustering and failover are also now hidden from the application writer.
- Only cluster-aware programs need to know about JCPI.

Applications that use JCPI are 100% Pure Java system applications, such as transaction and messaging software (e.g., Iona’s Java ORB). JCPI provides the support for such middleware to become cluster-aware in a portable 100% Pure Java fashion.

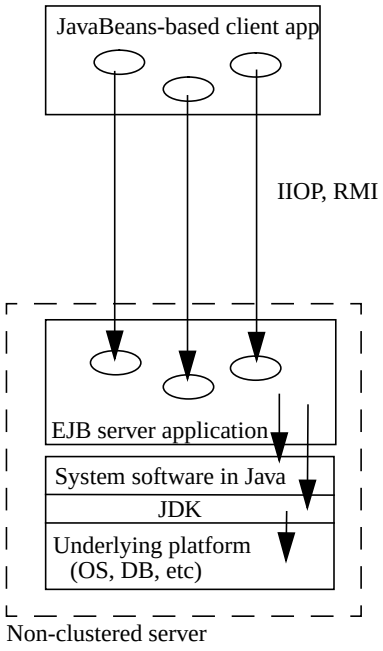


Figure 13. Enterprise JavaBeans in a non-clustered system

11.4.4 Making Enterprise JavaBeans highly-available

Figure 13 illustrates how Enterprise JavaBeans are normally used in a non-clustered environment. Figure 14 shows how the run-time support of EJB can be changed to hide the details of clustering and failover, while running the same existing Java applications. Figure 14 also shows the use of JCPI in the system to provide clustering support for Java system software, such as a native Java implementation of EJB support.

Support for Enterprise JavaBeans has been or will be announced by every major database and middleware vendor, including Oracle, Informix, IBM, Sybase, and BEA. Most of these applications and middleware servers are already highly-available on the Sun Cluster platform using the existing “C” HA APIs. The task of making EJB highly-available on SC 3.0 will require providing failover support for IIOP/RMI from the client to the new failed-over server in the cluster. No changes to the EJB run-time is anticipated.

11.5 VIA API

The Virtual Interface Architecture (VIA) is an emerging industry standard, championed by Intel, Microsoft, and Compaq, and endorsed by over fifty companies, including all the major database vendors. VIA is intended both as a hardware architecture and a software API. The API is targeted at high-speed cluster communication for database applications (e.g., Oracle OPS and Informix XPS), as well as high-performance computing applications (as a portable API for HPC libraries such as MPI and PVM).

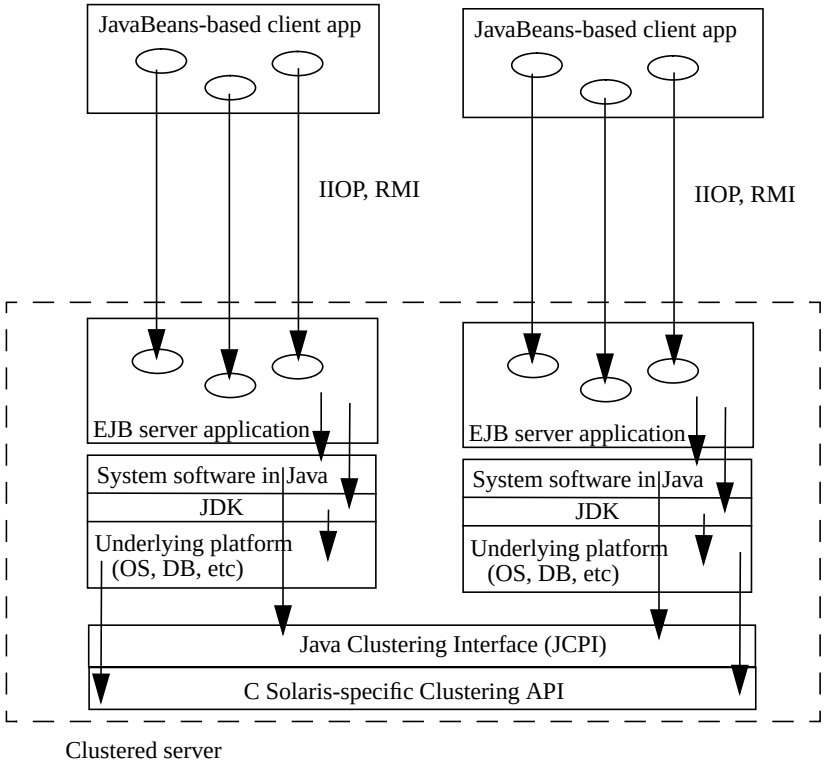


Figure 14. Enterprise JavaBeans and Java Clustering Interface in a clustered system

SC 3.0 will provide an implementation of the software VIA API. This implementation will be based on a user-level LWPI layer, and therefore, will be independent from the underlying transport hardware.

12.0 Carrying SC 2.2 Features into SC 3.0

SC 3.0 will support all SC 2.2 APIs and applications. In order to maximize the reuse of the SC 2.2 framework, SC 3.0 provides a user-level CMM with the same interface as the SC 2.2 CMM. The SC 2.2 framework will modified by removing disk, volume, and file system management, which will now be managed by PXFS directly, as shown in Figure 15. The framework will continue to provide the other 2,2 functionality, including implementing the HA APIs, failing-over IP addresses, and performing call-backs on the layered applications in the proper order.

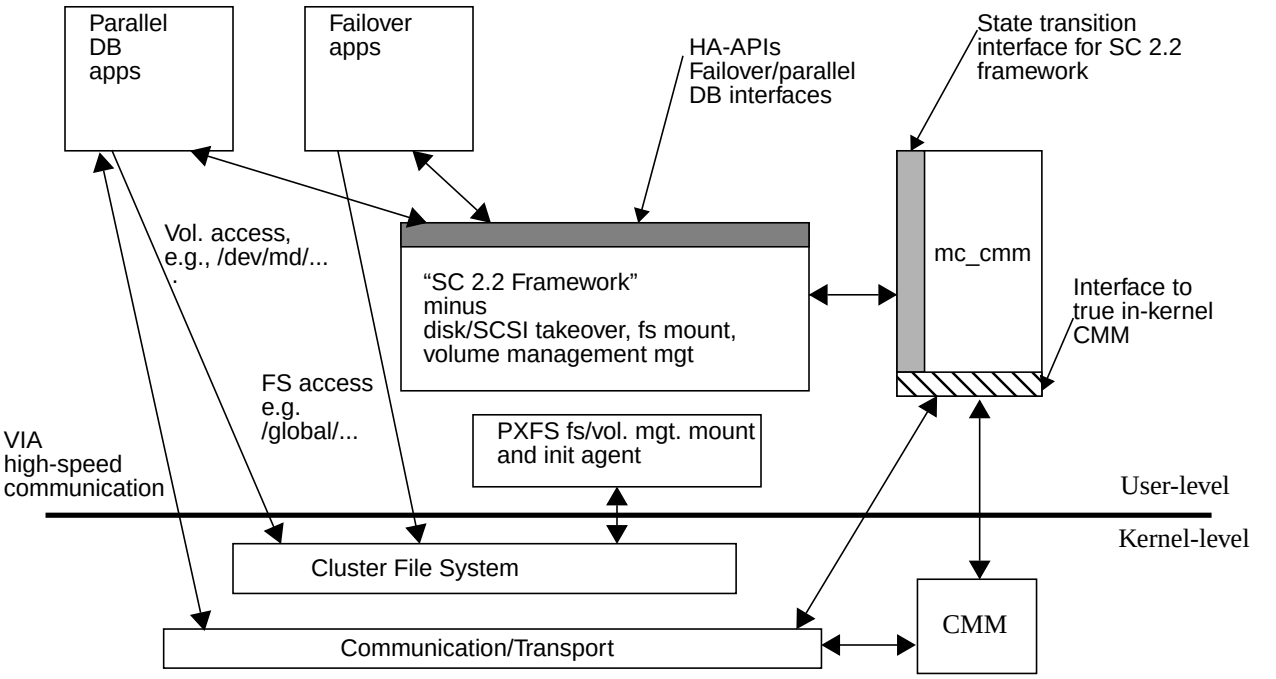


Figure 15. How SC 2.2 framework fits into SC 3.0

13.0 References

[1] "Solaris MC: A Multicomputer OS", Yousef A. Khalidi, Jose Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani, *Proceedings of 1996 USENIX Conference*, January 1996. Also available as Sun Labs Technical Report SMLI-95-48. Available at http://sunlabs.eng/projects/hiperf/home.

[2] "The Design of Solaris MC, A Prototype Multicomputer OS", 2nd ed., edited by Yousef A. Khalidi, Sun Labs internal Technical Report SMLI-94-492. Available at http://sunlabs.eng/projects/hiperf/home.