Inside UnixWare® NonStop® Clusters

Douglas Steel,
(douglass@sco.com)
Cluster Technology Group,
SCO Ltd.,
Watford WD1 8YN,
U.K.

ABSTRACT

UnixWare® NonStop® Clusters (NSC) runs on a distributed cluster of independent nodes connected by a System Area Network. Each node in the cluster runs an independent copy of the kernel and yet appears as a single system to users, applications and system administrators. NSC hides the distributed nature of the underlying kernel and hardware, whilst maintaining UnixWare functionality and semantics, and providing advanced system scalability and tolerance to node failures.

The SSI means that a process is presented with the same interface wherever in the cluster it runs and it may be transparently migrated on the fly, enabling loadbalancing and process eviction prior to node shutdown.

This article gives an overview of enhancements to UnixWare that make this possible, including process management, IPC, STREAMS, networking, filesystems, device access and memory management. Additional API's are provided to allow control and monitoring of functionality such as cluster membership and process placement.

INTRODUCTION

UnixWare® NonStop® Clusters(NSC) is the result of many years research and development based on the original Locus project [Popek and Walker 1985]. It has undergone revision and iteration by Locus, Platinum, Tandem, Compaq and SCO. Implementations have been developed based on different base UNIX® kernel technologies including AIX, OSF/1 AD, SVR4.x and SVR5. UnixWare NonStop Clusters builds on this base to provide a robust and scalable cluster at the operating system level that makes a collection of computing nodes appear as a single system to users, applications and system administrators.

SINGLE SYSTEM IMAGE

Greg Pfister [Pfister 98] defined a Single System Image as "*the illusion, created by*

® UnixWare is a registered trademark of The Santa Cruz Operation, Inc., UNIX is a registered trademark of The Open Group, NonStop is a trademark of Tandem Computers Inc *software or hardware, that a collection of computing elements is a single computing resource*". Pfister also defined several "*levels*" of SSI:

- application and subsystem
- operating system kernel
- hardware

NonStop Clusters for UnixWare (NSC) is an operating system kernel level SSI. This provides a complete SSI by making all operating system resources transparently usable cluster wide. This in contrast to application and subsystem level SSI's such as Beowulf [Becker et al 1995], PVM [Sunderam 1990], Wolfpack [Microsoft 1997], GOLDRUSH [Kittur et al. 1996], and NFS[Sandberg et al. 1985]. Application and subsystem level SSI's tend to concentrate on one specific area e.g. filesystem, distributed shared memory, system management.

By making all operating system resources (such as processes, files and devices) usable cluster wide in a transparent manner whilst maintaining UnixWare semantics, NSC allows standard UnixWare applications to run unmodified on the cluster. Applications automatically benefit from the system's inherent scalability and resilience. Since NSC's kernel level SSI makes use of the cluster transparent system management is achieved by trivial extensions to the core UnixWare system management facilities. System resilience can be administered by operations such as the ability to specify alternative routes to multiported disks.

To make full use of NSC's facilities additional API's are provided to allow applications to control placement and migration of processes, and to monitor and react to changes in the cluster's configuration. An application can make use of the inherent parallelism and scalability by simply forking multiple processes and using standard UNIX IPC mechanisms (e.g. sockets, pipes, signals, SYSV IPC) to communicate.

KERNEL LEVEL SSI REQUIREMENTS

In order to provide a complete operating system kernel level SSI, NSC supports all standard UnixWare API's and system calls. These are location transparent such that access to any resource (device, file, process, etc) can be made from any node in the cluster. For example it is possible for a process running on node 1 to open a device located on node 2, read from it, copy the data to a file hosted by a filesystem on node 3 and to signal process on node 4 to say that it has finished, without any knowledge of where those resources are located.

If a cluster operating system is to provide a complete SSI, all operating system resources must be accessible clusterwide without needing to know where the resources are physically located. Such resources include:

- processes
- IPC objects (including shared memory)
- files
- devices

NSC provides location transparent access to all of these resources.

In NSC many resources (processes, filesystems, etc.) can migrate between nodes in the cluster to support loadbalancing, process eviction prior to node shutdown, etc. Core system resources are resilient to node failure and can be transparently migrated, rebuilt or restarted on another node.

NSC supports the a kernel le vel SSI through additional kernel modules which are invoked through a small number of "hooks" located at strategic points throughout the base UnixWare kernel code (see Figure 1). Hooks exist in areas such as process management, file system framework, virtual memory, STREAMS, TCP/IP stack. Additional kernel modules include distributed process management, cluster membership, file token management, cluster file system, distributed STREAMS.

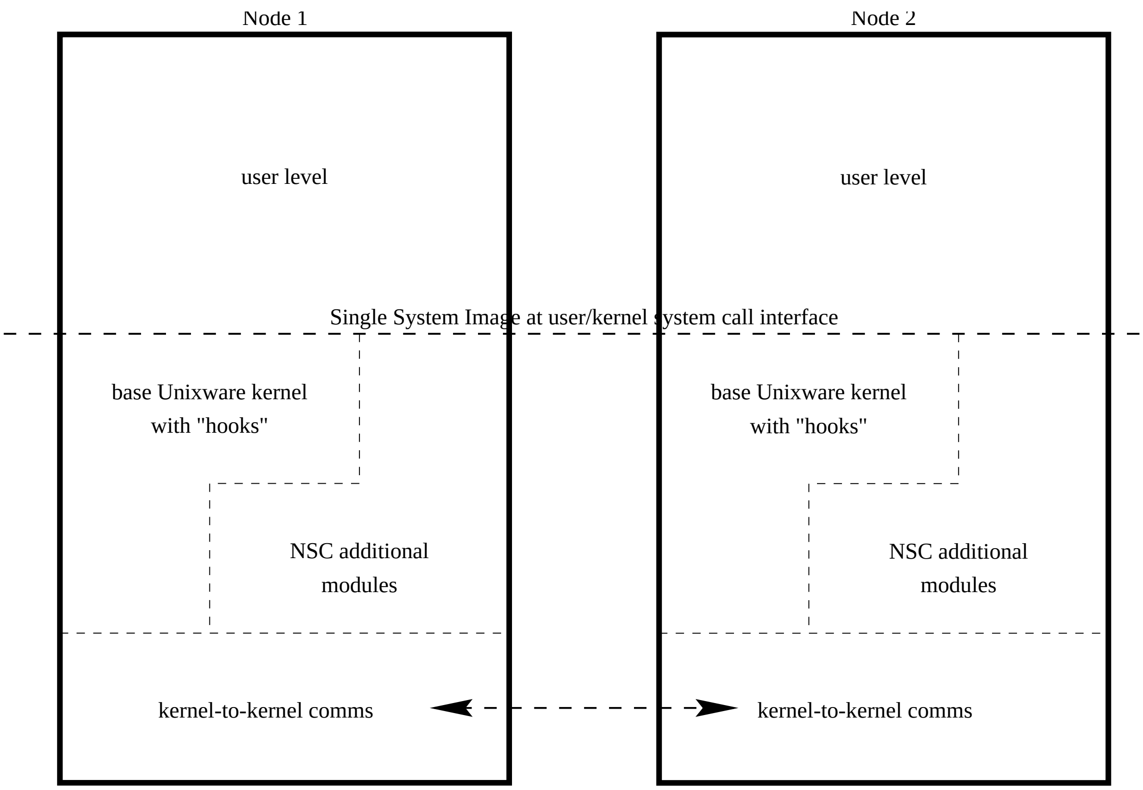

Figure 1 – relationship of kernel components to SSI

The kernel level SSI is built upon a low−level low−latency kernel−to−kernel communications layer currently based on top of ServerNet, but designed to be portable to other Server Area Networks (SANs) compliant with the Virtual Interface (VI) Architecture [VIARCH 1997]). Kernel modules use an RPC mechanism to talk to each other in contrast to SolarisMC which uses a CORBA based remote method invocation mechanism [Khalidi et al. 1996].

PROCESS MANAGEMENT

Processes and IPC objects are often not covered by many cluster operating systems and hence they cannot be considered to provide a full SSI. NSC gives full location transparency for processes and IPC objects, allowing processes executing on a collection of nodes to cooperate and communicate as they would on a single operating system instance. NSC also allows processes to migrate between nodes whilst they are executing in contrast to some cluster OS's require processes to shutdown and restart from a saved state.

In order to support location transparency for the process management subsystem NSC provides a virtual process (vproc) abstraction above the normal UNIX process [Walker et al. 1993]. The vproc is analogous to the UNIX filesystem virtual node (vnode) abstraction [Klieman 1986] which sits over a filesystems inode and provides filesystem type transparency. The vproc abstraction hides from the system call layer of the process management subsystem the physical implementation of the process which may be either on the local node or on a remote node. This is invisible to the user (unless specifically requested) and all standard UNIX process management interfaces can be used cluster wide e.g. ps, kill, /proc. There is no restriction on placement of processes, for example a child process may exist on a different node from its parent and they may form part of a process group which spans the whole cluster.

NSC's implementation of process management involves splitting the base UnixWare process management kernel software into an upper half which handles system calls (and calls from other parts of the kernel) by performing operations on vprocs, and a lower half which performs the physical implementation of a local process (known as a pproc). In between these section of base kernel software is inserted the vproc layer which is responsible for the distributed process management by invoking either the local pproc code or using RPC's to talk to a remote nodes vproc code.

To simplify management of the processes across the cluster the process id (pid) was extended from the 16bits of the base UnixWare to 32bits on NSC. The extra bits are used to record which node the process was created on – the origin node, this is not necessarily where it is currently running, it may have been migrated to another node. When the system tries to contact a process via its pid (e.g. to send a signal) it first checks to see whether it already knows its location (if a local copy of the vproc exists this information is recorded in the vproc's private information), if not it asks the origin node which it calculates from the top bits of the pid. If the origin node goes down the process is assigned a surrogate origin, which all further location requests will be directed to. Once known, the current location is recorded in the local copy of the vproc for that process.

Several options exist to distribute processing amongst the available nodes, a process may migrate itself, it may be asked to migrate, it may direct fork/exec operations to another node (rfork and rexec system calls), or fork/exec operations may be redirected by the kernel to other nodes based on load.

Redirection of fork/exec requests are straightforward to deal with, and which to choose depends on the behaviour of the process. In general it is more expensive to perform a remote fork operation as this involves replicating the entire processes state including file descriptors, memory pages, stack. A remote exec involves only moving the kernel state since the user memory (text, data) is being overwritten by a new image.

A process may either migrate itself (using the migrate system call) or be asked to migrate by being sent the SIGMIGRATE signal. However migration is requested it is performed in the context of the process that it being migrated. If a process wants to "pin" itself onto a node for perfmoance reasons (e.g. to be close to a physical device) it can ignore the SIGMIGRATE signal.

Migrating a process is similar in may ways to a remote fork/exec operation, additional considerations include suspending the process, copying its state to the target node, making itself runnable on the target node and removing its anonymous pages (those pages not backed by a nead file) from the source nodes swap device[1]. A successful migration results in the process resuming execution at the point where it was suspended. If the migration failed (e.g. the target node crashed) the process can be resumed on the node where it was running when the migration request was received.

NSC fully supports the three SystemV IPC mechanisms (message queues, semaphores and shared memory segments) between processes that may be located on different nodes. A global IPC namespace is managed by a nameserver which allocates identifiers to a node that is creating an IPC object. If the IPC nameserver is lost due to a node failure, a new nameserver is started which rebuilds its name database by polling the remaining nodes. An IPC object is managed by the node it was created on however it can be used by a process executing on any node. A message queue or semaphore can be directly accessed if it is local, if it is remote an RPC is performed to the node that is managing the IPC object. A shared memory object is used by a client caching pages, using the underlying cluster filesystem token mechanism to coordinate access, by granting or revoking read or write access to pages or ranges of pages.

FILE SYSTEMS

Many distributed files systems have been implemented that use a client/server model to provide a coherent global view of a filesystem. NFS has been very successful in this domain, although it trades off standard UNIX file semantics and client write performance to achieve stateless operation and data consistency. Several cluster file systems have been developed where peer nodes have concurrent access to shared storage devices, with the overhead of using a Distributed Lock Manager to coordinate access.

The NSC cluster filesystem (cfs) uses a stateful client/server with aggressive client and server side caching and token management to achieve full UNIX file semantics and high performance. Different levels of data integrity can be specified on a per filesystem or per file basis, allowing the user control over the tradeoffs between integrity and performance at filesystem mount or file access:

- soft mounts – the client maintains a cache of all dirty pages until they are marked clean when they have been sent to the node serving the filesystem. On a server

---

1 new page out requests are directed at the targets swap device once the process resume execution (if it has one). This purging of anonymous pages from the swap device removes dependencies on the source node’s swap device – if a process migrated several times it would introduce new failures modes if its swapped out pages were spread out across the cluster. Swap space management is covered in more detail later.

failure existing open files are marked failed and further attempts to access to these files returns and error, once a server recovers or is failed over, new opens will succeed.

- hard mounts – the client maintains a cache of all dirty pages until the server has confirmed they have been flushed to disk. On a server failure clients block until the filesystem is available – this handles the case when the server crash and reboots ensuring no client data is lost if the client survived the crash. State in the form of file locks, tokens, etc., is kept on the client so that it can be rebuilt after a server failure.
- hard mount with double caching – similar to hard mounts above except in the case that the client and server are running on the same node. A second node is selected as a backup cache node to ensure that in the case of the client/server node crashing no data is lost.

The cfs implementation is based on the concept of stacking a vnode/vfs on top of a physical(or pseudo) filesystem's vnode/vfs [Skinner and Wong 1993]. When a filesystem is mounted a cfs is stacked on top of the the filesystem's vfs and all vfs operations are directed at the cfs layer. Similarly for vnode operations a cfs vnode (cnode) is stacked on top of another filesystem's vnode (e.g. VxFS vnode) and all access to the file represented by that vnode goes through the cnode which performs and necessary local/remote processing. Using the cfs mechanism means that no physical(or pseudo) filesystem needs to be modified to be made cluster aware. In fact NSC supports several filesystem's unmodified (VxFS, UFS, S5FS, memfs, cdfs). Some special filesystems (e.g. procfs, specfs) have another cluster file system inserted between cfs and their implementation to allow the implementation of cluster specific semantics, these will be discussed later.

Any filesystem needs only be mounted once in the cluster and is immediately visible to all nodes in the cluster via a cluster mount service. A single file namespace and single root "/" hides the physical location of filesystems.

A token management subsystem grants tokens to clients and servers to coordinate access to file data and metadata (such as file length or modification time). Tokens have a rich set of semantics allowing locking of whole files and ranges, file offsets, flags, access time, etc. The token management subsystem ensures fairness and coherency between nodes in the cluster.

NSC supports several nodes having physical connections to the same storage device although only one may be active at a time. Multiple connections are useful for failure recovery. When a node goes down, all filesystems it was hosting are marked temporarily unavailable. If the filesystem has been mounted with the cluster hard mount option (chard analogous to NFS hard mounts) all clients accesses are suspended until either the node hosting the filesystem returns to the cluster or another node takes over hosting the filesystem. When a filesystem has been recovered all suspended operations are resumed. File systems can be migrated by unmounting with a special "remount" option which marks the file system as temporarily unavailable as in the failure case above, and then remounting on another node that has physical access to this file system.

PROCESS FILE SYSTEM

The process filesystem (procfs) on standard UnixWare presents the user level with an interface to individual processes. The procfs is used by user processes to access and control process specific data (e.g. ps uses it to list process status information; debug, the user level debugger, uses it to access the process image and control the process by setting breakpoints).

On NSC each node implements its own procfs providing access to all processes managed by that node. An additional cluster process filesystem (cprocfs) is layered on top of the local procfs to present a unified view of the procfs to the cfs which in turn provides coherent cluster wide access to the process filesystem via a single /proc mountpoint. This means that the ps command, for instance, can remain unchanged and can view the complete set of processes running on all nodes (cf. Beowulf procps with relies on a set of daemons (one per node) to distribute and collate ps requests from a modified ps command). Giving such global access to the /proc filesystem also means that no changes are required to the user level debugger (debug) which can now debug processes that are not running on the same node (i.e. we automatically get a remote debugger).

DEVICE ACCESS (specfs)

The special filesystem (specfs) manages "special" files which, in UNIX, represent devices. A file operation on a special file is directed to specfs which will work out from the major/minor (dev_t) number which device driver and device instance to direct the request. In NSC all devices are accessible cluster wide via the cluster specfs (cspecfs). Additional information is encoded in the dev_t, alongside the major/minor combination, which specifies the node where the device is located. The node information is used by cspecfs to direct the device access request to the appropriate node. In a similar manner to the way cfs uses cnode's to hide vnodes, cspecfs uses csnodes to hide snodes (special nodes).

Physical devices all have their physical location encoded in the dev_t, however a certain encoding specifies "this node". This is used for some pseudo devices where it makes no sense to ship the request of to a remote node. Such local pseudo devices include:

- /dev/null and /dev/zero
- pseudo tty's
- /dev/mem and /dev/kmem

However the kernel needs to take special care when accessing these device when a process migrates – the kernel decides when to use the pseudo device on the node where the open took place, and when this doesn't matter. For example, a process that opens /dev/mem should always be given access to the same /dev/mem while the device is open as it may have been accessing a data structure (e.g. traversing a linked list) and the program may break if the actual /dev/mem it was using changed beneath it. On the other hand it doesn't matter which /dev/zero a process gets its zeros from.

NSC implements a distributed STREAMS architecture in which any node can host a STREAM head and all access to that STREAM head is directed at the node hosting it (via cspecfs).

VIRTUAL MEMORY AND SWAP SPACE MANAGEMENT

The hardware model supported by NSC is a traditional NORMA cluster (i.e. it is a collection of independent nodes connected by a network – with NO Remote Memory Access in hardware). All cross node shared memory is managed purely by software.

The mechanism used to share memory is unified with the mechanism used for caching file system data. All access to shared pages is managed by the token subsystem. To gain read or write access to a page, a node is granted a token for the page (tokens can be granted for shared read or exclusive write) which gives the node the right to cache the page. Once a node is finished with the page it may return the tokens to thetoken subsystem, and then dispose of the page if it is clean or write it out to a device (possibly remote) if it has been modified. If two nodes require write access to the same page the token system grants the token to one node and the revokes the token after a suitable period to avoid thrashing and allow some processing to take place on each node.

Swap space on NSC is managed as a global pool with preference given for using local swap. When a swapadd operation is performed, the swap is added to the global pool, the swap reservation manager allocates a portion of the global swap space to each node. If a node runs out of local swap space (or if it never had any in the first place) it requests more from the global pool.

DISTRIBUTED NETWORKING

NSC allows any node to have a number of physical network connections on to an external network. NSC allows loadbalancing of connections on a per external host basis between these physical connections. A node does not require physical connection to access an external network as it may use any of the physical connections of any other node. A cluster virtual IP (cvip) address is common to all nodes and is not tied to any specific physical connection. A physical connection acts as an ARP proxy for the cvip, allowing the connection to move between physical interfaces.

All NSC nodes have a complete TCP/IP stack and see a single port space. Any node may listen on any of the cluster's IP addresses (physical or cvip) – including wildcard listens. A "glue" module is placed between the TCP and IP STREAMS modules and allows any incoming or outgoing connection to be routed across the cluster interconnect to any node. The "glue" module uses a Network Registration Database (NeRD) which ensures that connections and endpoints are unique across the whole cluster.

USER LEVEL UTILITIES AND DAEMONS

An operating system does not just consist of the kernel, but also includes user level utilities and daemons. To build a highly available SSI cluster, several user level utilities and daemons also need to be made cluster aware and/or highly available.

The init process has two main modifications:

- it monitors the network to initiate node specific startup and shutdown scripts

- it needs to restart if the node it was running on

Additional daemons were added for:
- application monitoring and failover (keepalive)
- loadbalancing (loadleveller)

The keepalive daemon monitors a process or process group and if the process or group dies (e.g. the node it was running on fails) then it is restarted elsewhere.

The loadleveller daemon monitors the load of all the nodes and if the load of a node increases beyond a specified threshold it attempts to level the load by:
1. sending some processes a SIGMIGRATE (a program can ask the loadleveller not to migrate it by registering itself in an exclusion list).
2. asking the kernel of the overloaded node to convert some execs into remote execs

Several other user utilities have been modified to become cluster aware. For example an additional parameter is can be passed to ps specifying that only the process on the local node should be listed.

Additional APIs have been provided to allow programs to become cluster aware, including:

- control over process placement and migration
- cluster membership and configuration information

CONCLUSIONS

NonStop Clusters for UnixWare hides the distributed nature of the underlying hardware by implementing a true Single System Image. This enables standard applications to run unaware that they are running on a cluster, and while they are running they may be transparently migrated. Operating services may become temporarily unavailable due to hardware failure and then recovered, without losing the whole cluster and with minimal disruption to services. Additional utilities and APIs allow applications and administrators to make full use of the salability and availability provided by NSC.

Future enhancements include support for cross node disk mirroring, more APIs (including user level VI architecture support), support for additional cluster interconnects (using kernel level VI architecture), migration of more kernel objects (sockets, IPC objects), and online system upgrade.

NSC is available now on UnixWare 2.1.2 in configurations of up to six nodes with a scalability of 97% [SPEC 1998], it has also been demonstrated on a configuration of 30 nodes. NSC is currently being ported to UnixWare 7.1 and this version is expected to be released in Summer 1999. Work is under way to integrate the hooks required into a release of the base UnixWare operating system due for release in mid2000.

REFERENCES

[Becker et al 1995] D. Becker, T. Sterling, D. Saverese, J. Droband, U. Ranawak, C. Packer, "Beowulf: A Parallel Workstation for Scientific Computing", Proceedings International Conference on Parallel Processing 1995

[Khalidi et al 1996] J. Kahlidi, J. Bernabeu, V. Matena, K. Sherriff, M. Thadani, "Solaris MC: A Multi Computer OS", USENIX 1996 Annual Technical Conference, pp191−203

[Kittur et al 1996] S. Kittur, D. Steel, F. Armand, J. Lipkis, "Fault Tolerance in a Distributed CHORUS/MiX System", USENIX 1996 Annual Technical Conference, pp219−228

[Kleiman 1986] S. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", USENIX Summer 1986 Conference, pp238−247

[Microsoft 1997] Microsoft Corporation, "Microsoft 'Wolfpack' Technical Architecture", Microsoft 1997

[Popek and Walker 1985] G. Popek and B. Walker, "The LOCUS Distributed System Architecture", MIT Press, 1985

[Pfister 1998] Greg Pfister, "In Search of Clusters", 2nd Edition, Prentice Hall 1998

[Sandberg et al 1985] R. Sandberg, D. Goldberg, D. Walsh, B. Lyon, "The Design of the Sun Network File System", Proceedings of the Summer USENIX 1985, pp119−131

[Skinner and Wong 1993] G. Skinner and T. Wong, "Stacking Vnodes: A Progress Report", USENIX 1993 Summer Conference

[SPEC 1998] Standard Performance Evaluation Corporation , "SPECint_rate95 results", available from http://www.spec.org/osg/cpu95/results/res98q4/

[Sunderam 1990] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing", Concurrency: Practice and Experience, December 1990, pp315−339

[VIARCH 1997] VI Architecture group, "Virtual Interface Architecture Specification Version 1.0", December 1997, available from http://www.viarch.org (registration required)

[Walker et al 1992] B. Walker, R. Zajcew, G. Thiel, "VPROCS: A Virtual Process Interface for POSIX Systems", Technical Report LA−0920, Locus Computing Corp., 1992

About The Author

Douglas Steel has been involved in research and development of distributed and clustered systems for 11 years since he graduated from the University of Glasgow in 1988. At Queen Mary and Westfield College (Uni. of London) he was involved in post graduate research in operating system support for distributed object oriented programming, and completed a Masters degree. From 1991 to 1993 at UNIX System Laboratories Europe he designed and implemented CORBA interfaces between the UNIX emulation servers of CHORUS/MiX. From 1993 to 1996 he was a senior engineer at ICL High Performance Systems where he was responsible for the design and development of the filesystem failover mechanism for the GOLDRUSH MegaSERVER (based on CHORUS/MiX). In July 1996 he joined the Cluster Technology Group at SCO Ltd., Watford, where he has been involved in the development of ReliantHA and is currently porting NonStop Clusters to UnixWare 7.1 and integrating hooks into the core UnixWare OS.