An Introduction to Single System Image (SSI) Cluster Technique

Tarun Kumawat [CSE] , JECRC UDML College of Engineering. Kukas, Jaipur, Rajasthan, India[1]
Sandeep Tomar [CSE] , Arya College of Engineering & I.T. Kukas, Jaipur, Rajasthan, India[2]
Mohit Gupta [CSE] , Arya College of Engineering & I.T. Kukas, Jaipur, Rajasthan, India[3]
[1]tarun.kumawat04@gmail.com
[2]sandeeptomar12@yahoo.com
[3]mohit15.1990@gmail.com

***Abstract*-Cluster computing is not a new area of computing. It is, however, evident that there is a growing interest in its usage in all areas where applications have traditionally used parallel or distributed computing platforms. A Single System Image (SSI) is the property of a system that hides the heterogeneous and distributed nature of the available resources and presents them to users and applications as a single unified computing resource. SSI can be enabled in numerous ways, this range from those provided by extended hardware through to various software mechanisms. SSI means that users have a globalised view of the resources available to them irrespective of the node to which they are physically associated.**

Keywords: Cluster SSI, SCO UnixWare, GLUnix, MOSIX

I. INTRODUCTION

A Single System Image (SSI) is the property of a system that hides the heterogeneous and distributed nature of the available resources and presents them to users and applications as a single unified computing resource. SSI can be enabled in numerous ways, this range from those provided by extended hardware through to various software mechanisms. SSI means that users have a globalised view of the resources available to them irrespective of the node to which they are physically associated. Furthermore, SSI can ensure that a system continues to operate after some failure (high availability) as well as ensuring that the system is evenly loaded and providing communal multiprocessing (resource management and scheduling).

SSI design goals for cluster-based systems are mainly focused on complete transparency of resource management, scalable performance, and system availability in supporting user applications [1][2][3][5][7]. A SSI can be defined as the illusion [1][2], created by hardware or software, that presents a collection of resources as one, more powerful unified resource.

II. SERVICES AND BENEFITS

The key services of a single-system image cluster include the following [1][3][4]:

- *Single entry point:* A user can connect to the cluster as a virtual host (like telnet beowulf.myinstitute.edu), although the cluster may have multiple physical host nodes to serve the login session. The system transparently distributes user's connection requests to different physical hosts to balance load.
- *Single user interface:* The user should be able to use the cluster through a single GUI. The interface must have the same look and feel than the one available for workstations (e.g., Solaris OpenWin or Windows NT GUI).
- *Single process space:* All user processes, no matter on which nodes they reside, have a unique cluster-wide process id. A process on any node can create child processes on the same or different node (through a UNIX fork). A process should also be able to communicate with any other process (through signals and pipes) on a remote node. Clusters should support globalised process management and allow the management and control of processes as if they are running on local machines.
- *Single memory space: Users* have an illusion of a big, centralised main memory, which in reality may be a set of distributed local memories. Software DSM approach has already been used to achieve single memory space on clusters. Another approach is to let the compiler distribute the data structure of an application across multiple nodes. It is still a challenging task to develop a single memory scheme that is efficient, platform independent, and able to support sequential binary codes.
- *Single I/O space (SIOS):* This allows any node to perform I/O operations on local or remotely located peripheral or disk device. In this SIOS design, disks associated to cluster nodes, network-attached RAIDs, and peripheral devices form a single address space.
- *Single file hierarchy:* On entering into the system, the user sees a single, huge file-system image as a single hierarchy of files and directories under the same root directory that transparently integrates local and global disks and other file devices. Examples of single file hierarchy include NFS, AFS, xFS, and Solaris MC Proxy.

- *Single virtual networking:* This means that any node can access any network connection throughout the cluster domain even if the network is not physically connect to all nodes in the cluster. Multiple networks support a single virtual network operation.
- *Single job-management system:* Under a global job scheduler, a user job can be submitted from any node to request any number of host nodes to execute it. Jobs can be scheduled to run in either batch, interactive, or parallel modes. Examples of job management systems for clusters include GLUnix, LSF, and CODINE.
- *Single control point and management:* The entire cluster and each individual node can be configured, monitored, tested and controlled from a single window using single GUI tools, much like an NT workstation managed by the Task Manger tool.
- *Checkpointing and Process Migration:* Checkpointing is a software mechanism to periodically save the process state and intermediate computing results in memory or disks. This allows the roll back recovery after a failure. Process migration is needed in dynamic load balancing among the cluster nodes and in supporting Checkpointing. Figure 1 shows the functional relationships among various key middleware packages.

These middleware packages are used as interfaces between user applications and cluster hardware and OS platforms. They support each other at the management, programming, and implementation levels.

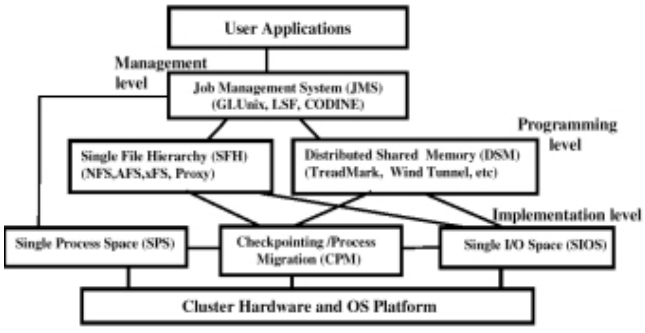

Figure 1. The relationship between middleware modules [3].

The most important benefits of SSI include the following [1]:

- It provides a simple, straightforward view of all system resources and activities, from any node in the cluster.
- It frees the end-user from having to know where in the cluster an application will run.
- It allows the use of resources in a transparent way irrespective of their physical location.
- It lets the user work with familiar interface and commands and allows the administrator to manage the entire cluster as a single entity.
- It offers the same command syntax as in other systems and thus reduces the risk of operator errors, with the result that end-users see an improved performance, reliability and higher availability of the system.
- It allows to centralise/decentralise system management and control to avoid the need of skilled administrators for system administration.
- It greatly simplifies system management and thus reduced cost of ownership.
- It provides location-independent message communication.
- It benefits the system programmers to reduce the time, effort and knowledge required to perform task, and allows current staff to handle larger or more complex systems.
- It promotes the development of standard tools and utilities.

III. SSI LAYERS/LEVELS

The two important characteristics of SSI [1][2] are:
1. Every SSI has a boundary,
2. SSI support can exist at different levels within a system — one able to be built on another.

SSI can be implemented in one or more of the following levels:

- Hardware,
- Operating System (so called *underware* [5]),
- Middleware (runtime subsystems),
- Application.

A good SSI is usually obtained by a co-operation between all these levels as a lower level can simplify the implementation of a higher one.

A. Hardware Level

Systems such as Digital/Compaq Memory Channel [8] and hardware Distributed Shared Memory (DSM) [8] offer SSI at hardware level and allow the user to view a cluster as a shared-memory system. Digital's Memory Channel is designed to provide a reliable, powerful and efficient clustering interconnect. It provides a portion of global virtual shared memory by mapping portions of remote physical memory as local virtual memory (called reflective memory).
Memory Channel consists of two components: a PCI adapter and a hub. Adapters can also be connected directly to another adapter without using a hub. The host interfaces exchange heartbeat signals and implement flow control timeouts to detect node failure or blocked data transfers. The link layer provides error detection through a 32 bit CRC generated and checked in hardware. Memory Channel uses point-to-point, full-duplex switched 8x8 crossbar implementation.

To enable communication over the Memory Channel network, applications map pages as read- or write-only into their virtual address space. Each host interface

contains two page control tables (PCT), one for write and one for read mappings. For read-only pages, a page is pinned down in local physical memory. Several page attributes can be specified: receive enable, interrupt on receive, remote read etc. If a page is mapped as write-only, a page table entry is created for an appropriate page in the interface 128 Mbytes of PCI address space. Page attributes can be used to store a local copy of each packet, request acknowledgement message from receiver side for each packet, and define the packets as broadcast or point-to-point packets. Broadcasts are forwarded to each node attached to the network. If a broadcast packet enters a crossbar hub, the arbitration logic waits until all output ports are available. Nodes, which have mapped the addressed page as a readable area, store the data in their local pinned down memory region. All other nodes simply ignore the data. Therefore once the data regions are mapped and set up, simple store instructions transfer data to remote nodes, without OS intervention. Besides this basic data transfer mechanism, Memory Channel supports a simple remote read primitive, a hardware-based barrier acknowledges, and a fast lock primitive. To ensure correct behaviour, Memory Channel implements a strict in-order delivery of written data. A write invalidates cache entries on the reader side, thus providing cluster-wide cache coherence.

Digital provides two software layers for Memory Channel: the Memory Channel Services and Universal Message Passing (UMP). The first is responsible for allocating and mapping individual memory page. UMP implements a user-level library of basic message passing mechanisms. It is mainly used as a target for higher software layers, such as MPI, PVM or HPF. Both layers have been implemented for the Digital UNIX and the Windows NT operating systems. Memory Channel reduces communication to the minimum, just simple store operations. Therefore, latencies for single data transfers are very low. This also enables the Memory Channel to reach the maximal sustained data rate of 88 Mbytes/s with relative small data packets of 32 bytes. The largest possible configuration consists out of 8 12-CPU Alpha server nodes, resulting in a 96-CPU cluster.

B. Operating System Level

Cluster operating systems support an efficient execution of parallel applications in an environment shared with sequential applications. A goal is to pool resources in a cluster to provide better performance for both sequential and parallel applications. To realise this goal, the operating system must support gang scheduling of parallel programs, identify idle resources in the system (such as processors, memory, and networks), and offer globalised access to them. It should optimally support process migration to provide dynamic load balancing as well as fast inter-process communication for both the system and user-level applications. The OS must make sure these features are available to the user without the need for additional system calls or commands. OS kernel supporting SSI include SCO UnixWare NonStop Clusters [5][6], Sun Solaris-MC [9], GLUnix [11], and MOSIX [12].

1) SCO UnixWare

UnixWare NonStop Clusters is SCO's high availability software. It significantly broadens hardware support making it easier and less expensive to deploy the most advanced clustering software for Intel systems. It is an extension to the UnixWare operating system where all applications run better and more reliably inside a Single System Image (SSI) environment that removes the management burden. It features standard IP as the interconnect, removing the need for any proprietary hardware.

The UnixWare kernel has been modified via a series of modular extensions and hooks to provide: single cluster-wide file-system view; transparent cluster-wide device access; transparent swap-space sharing; transparent cluster-wide IPC; high performance inter-node communications; transparent cluster-wide process migration; node down cleanup and resource failover; transparent cluster-wide parallel TCP/IP networking; application availability; cluster-wide membership and cluster time-sync; cluster system administration; and load levelling.

UnixWare NonStop Clusters architecture offers built-in support for application failover using an "n + 1" approach. With this approach, the backup copy of the application may be restarted on any of several nodes in the cluster. This allows one node to act as a backup node for all other cluster nodes.

UnixWare NonStop Clusters also supports active process migration, which allows any application process to be moved to another node between instruction steps. This allows continuation without disruption to the application. Active process migration allows dynamic removal and addition of nodes within the cluster.

With the Single System Image (SSI) capability of UnixWare NonStop Clusters, both applications and users view multiple nodes as a single, logical system. SSI also provides automatic process migration and dynamic load balancing. Depending on the workload and available resources in the cluster, the system automatically reassigns processes among available nodes, delivering optimal overall performance. The cluster offers a single UNIX system name space and appears to the application as a very large n-way SMP server. The cluster services maintain the standard service call interface, so upper levels of the operating system do not need to be changed. Applications access clustered services through standard UNIX system libraries, which in turn access clustered services through the service-call interface. Applications do not need to be cluster aware and may run unmodified in the cluster.

The cluster service determines whether a request can be handled locally or must be forwarded to another node. If the request is passed to another node, it uses an internode communication system over ServerNet to communicate to the service peer on another node. The request is then

handled by the standard UNIX system service on the targeted node.

2) Sun Solaris MC

Solaris MC is a prototype extension of the single node Solaris Kernel. It provides single system image and high availability at the kernel level. Solaris MC is implemented through object-oriented techniques. It extensively uses the object-oriented programming language C++, the standard COBRA object model and its Interface Definition Language.

Solaris MC uses a global file system called Proxy File System (PXFS). The main features include single system image, coherent semantics, and high performance. The PXFS makes file accesses transparent to process and file locations. PXFS achieves this single system image by intercepting file-access operations at the vnode/VFS interface. When a client node performs a VFS/vnode operation, Solaris MC proxy layer first converts the VFS/vnode operation into an object invocation, which is forwarded to the node where the file resides (the server node). The invoked object then performs a local VFS/vnode operation on the Solaris file system of the server node. This implementation approach needs no modification of the Solaris kernel or the file system.
PXFS uses extensive caching on the clients to reduce remote object invocations. PXFS uses a token-based coherency protocol to allow a page to be cached read-only by multiple nodes or read-write by a single node.

Solaris MC provides a single process space by adding a global process layer on top of Solaris kernel layer. There is a node manager for each node and a virtual process (vproc) object for each local process. The vproc maintains information of the parent and children of each process. The node manager keeps two lists: the available node list and local process list, including migrated ones. When a process migrates to another node, a shadow vproc is still kept on the home node. Operations received by the shadow vproc are forwarded to the current node where the process resides.

Solaris MC provides a single I/O subsystem image with uniform device naming. A device number consists of the node number of the device, as well as the device type and the unit number. A process can access any device by using this uniform name as if it were attached to the local node, even if it is attached to a remote node.

Solaris MC ensures that existing networking applications do not need to be modified and see the same network connectivity, regardless of which node the application runs on. Network services are accessed through a service access point (SAP) server. All processes go to the SAP server to locate in which node a SAP is on. The SAP server also ensures that the same SAP is not simultaneously allocated to different nodes. Solaris MC allows multiple nodes to act as replicated SAP server for network services.

3) GLUnix

Another way for the operating system to support a SSI is to build a layer on top of the existing operating system and to perform global resource allocations. This is the approach followed by GLUnix from Berkeley [11]. This strategy makes the system easily portable and reduces development time.

GLUnix is an OS layer designed to provide support for transparent remote execution, interactive parallel and sequential jobs, load balancing, and backward compatibility for existing application binaries. GLUnix is a multi-user system implementation at the user level so that it can be easily ported to a number of different platforms. It is built as a protected, user-level library using the native system services as a building block.

GLUnix aims to provide cluster-wide namespace and uses Network PIDs (NPIDs) and Virtual Node Numbers (VNNs). NPIDs are globally unique process identifiers for both sequential and parallel programs throughout the system. VNNs are used to facilitate communications among processes of a parallel program. A suite of user tools for interacting and manipulating NPIDs and VNNs are supported.

GLUnix is implemented completely in the user level and does not need any kernel modification, therefore it is easy to implement. GLUnix relies on a minimal set of standard features from the underlying system, which are present in most commercial operating systems. So it is portable to any operating system that supports inter-process communication, process signalling, and access to load information. The new features needed for clusters are invoked by procedure calls within the application's address space. There is no need to cross hardware protection boundary, no need for kernel trap or context switching. The overhead for making system calls is eliminated in GLUnix. Using shared-memory segments or interprocess communication primitives can do the co-ordination of the multiple copies of GLUnix, which are running on multiple nodes.

The main features provided by GLUnix include: co-scheduling of parallel programs; idle resource detection, process migration, and load balancing; fast user-level communication; remote paging; and availability support.

4) MOSIX

MOSIX [12] is another software package specifically designed to enhance the Linux kernel with cluster computing capabilities. The core of MOSIX are adaptive (on-line) load-balancing, memory ushering and file I/O optimisation algorithms that respond to variations in the use of the cluster resources, e.g., uneven load distribution or excessive disk swapping due to lack of free memory in one of the nodes. In such cases, MOSIX initiates process migration from one node to another, to balance the load, or to move a process to a node that has sufficient free

memory or to reduce the number of remote file I/O operations.

MOSIX operates silently and its operations are transparent to the applications. This means that you can execute sequential and parallel applications just like you would do in an SMP. You need not care about where your process is running, nor be concerned what the other users are doing. Shortly after the creation of a new process, MOSIX attempts to assign it to the best available node at that time. MOSIX then continues to monitor the new process, as well as all the other processes, and will move it among the nodes to maximise the overall performance. All this is done without changing the Linux interface. This means that you continue to see (and control) all your processes as if they run on your node. The algorithms of MOSIX are decentralised – each node is both a master for processes that were created locally, and a server for (remote) processes, that migrated from other nodes. This means that nodes can be added or removed from the cluster at any time, with minimal disturbances to the running processes. Another useful property of MOSIX is its monitoring algorithms, which detect the speed of each node, its load and free memory, as well as IPC and I/O rates of each process. This information is used to make near optimal decisions where to place the processes.

The system image model of MOSIX is based on the home-node model, in which the entire user's processes seem to run at the user's login-node. Each new process is created at the same site(s) as its parent process. Processes that have migrated interact with the user's environment through the user's home-node, but where possible, use local resources. As long as the load of the user's login-node remains below a threshold value, all the user's processes are confined to that node. However, when this load rises above a threshold value, then some processes may be migrated (transparently) to other nodes.

The Direct File System Access (DFSA) provision extends the capability of a migrated process to perform some I/O operations locally, in the current node. This provision reduces the need of I/O-bound processes to communicate with their home node, thus allowing such processes (as well as mixed I/O and CPU processes) to migrate more freely among the cluster's node, e.g., for load balancing and parallel file and I/O operations. Currently, the MOSIX File System (MFS) meets the DFSA standards. MFS makes all directories and regular files throughout a MOSIX cluster available from all nodes, while providing absolute consistency as files are viewed from different nodes, i.e., the consistency is as if all file accesses were done on the same node.

C. Middleware Level

Middleware, a layer that resides between OS and applications, is one of the common mechanisms used to implement SSI in clusters. They include cluster file system, programming environments such as PVM [13], job-management and scheduling systems such as CODINE [14] and Condor [15], cluster-enabled Java Virtual Machine (JVM) such as JESSICA [18]. SSI offered by cluster file systems makes disks attached to cluster nodes
appear as a single large storage system, and ensure that every node in the cluster has the same view of the data. Global job scheduling systems manage resources, and enable the scheduling of system activities and execution of applications while offering high availability services transparently. Cluster enabled JVM allows execution of Java threads-based applications on clusters without any modifications.

CODINE is a resource-management system targeted to optimise utilisation of all software and hardware resources in a heterogeneous networked environment [14]. It is evolved from the Distributed Queuing System (DQS) created at Florida State University. The easy-to-use graphical user interface provides a single-system image of all enterprise-wide resources for the user and also simplifies administration and configuration tasks.

The CODINE system encompasses four types of daemons. They are the CODINE master daemon, the scheduler daemon, the communication daemons and the execution daemons. The CODINE master daemon acts as a clearinghouse for jobs and maintains the CODINE database. Periodically the master receives information about the host load values in the CODINE cluster by the CODINE execution daemons running on each machine. Jobs,
submitted to the CODINE system, are forwarded to the CODINE master daemon and then spooled to disk. The scheduler daemon is responsible for the matching of pending jobs to the available resources. It receives all necessary information from the CODINE master daemon and returns the matching list to the CODINE master daemon which in turn dispatches jobs to the CODINE execution daemons.

CODINE master daemon runs on the main server and manages the entire CODINE cluster. It collects all necessary information, maintains and administers the CODINE database. The database contains information about queues, running and pending jobs and the available resources in the CODINE cluster. Information to this database is periodically updated by the CODINE execution daemons.

The CODINE master daemon has a critical function, as the system will not operate without this daemon running. To eliminate this potential point of failure, CODINE provides a shadow master functionality. Should the CODINE master daemon fail to provide service, a new CODINE master host will be selected and another CODINE master daemon will automatically be started on that new host. All CODINE ancillary programs providing the user or administrator interface to CODINE directly contacts the CODINE master daemon via a standard TCP port to forward their requests and to receive an acknowledgement or the required information.

The CODINE scheduler daemon is responsible for the mapping of jobs to the most suitable queues. Jobs are submitted to the CODINE master daemon together with a list of requested resources. A job that cannot be dispatched immediately waits in a pending queue until the CODINE scheduler daemon decides the requested resources for this job are available. The result of the mapping is communicated back to the CODINE master daemon to update the database. The CODINE master daemon notifies the CODINE execution daemon on the corresponding machine to start the job.

The CODINE execution daemon runs on every machine in the CODINE cluster where jobs can be executed. It reports periodically the status of the resources on the workstation to the CODINE master daemon. The CODINE execution daemon is also responsible for starting and stopping the jobs. For each job, the CODINE execution daemon starts a subordinate shepherd process, which is responsible for running and monitoring its job. One or more CODINE communication daemons have to run in every CODINE cluster .These daemons are responsible for the communication between the other CODINE daemons. This allows asynchronous communication between the various CODINE daemons, speeding up the system and increasing efficiency. The communication daemons control the whole communication via a standard TCP port.

D. Application Level

Finally, applications can also support SSI. The application-level SSI is the highest and, in a sense, most important because this is what the end-user sees. At this level, multiple cooperative components of an application are presented to the user as a single application. For instance, a GUI based tool such as PARMON [15] offers a single window representing all the resources or services available. The Linux Virtual Server [17] is a scalable and high availability server built on a cluster of real servers. The architecture of the cluster is transparent to end-users as all they see a single virtual server. All other cluster-aware scientific and commercial applications (developed using APIs such as MPI) hide the existence of multiple interconnected computers and co-operative software components, and present themselves as if running on a single system.

The Linux Virtual Server (LVS) [17] directs network connections to the different servers according to scheduling algorithms and makes parallel services of the cluster to appear as a virtual service on a single IP address. Linux Virtual Server extends the TCP/IP stack of Linux kernel to supports three IP load balancing techniques: NAT, IP tunnelling, and direct routing. It also provides four scheduling algorithms for selecting servers from cluster for new connections: round-robin, weighted round-robin, least-connection and weighted Least-Connection. Client applications interact with the cluster as if it were a single server. The clients are not affected by interaction with the cluster and do not need modification. Scalability is achieved by transparently adding or removing a node in the cluster. High availability is provided by detecting node or daemon failures and reconfiguring the system appropriately.

Linux Virtual Server is a three-tier architecture.

- *Load Balancer* is the front end to the service as seen by the outside world. The load balancer directs network connections from clients who know a single IP address for services, to a set of servers that actually perform the work.
- *Server Pool* consists of a cluster of servers that implement the actual services, such as Web, Ftp, mail, DNS, and so on.
- *Backend Storage* provides the shared storage for the servers, so that it is easy for servers to keep the same content and provide the same services.

The load balancer handles incoming connections using IP load balancing techniques. Load balancer selects servers from the server pool, maintains the state of concurrent connections and forwards packets, and all the work is performed inside the kernel, so that the handling overhead of the load balancer is low. The load balancer can handle much larger number of connections than a general server, therefore a load balancer can schedule a large number of servers and it will not be a bottleneck of the whole system. Cluster monitor daemons run on the load balancer to monitor the health of server nodes. If a server node cannot be reached by ICMP ping or there is no response of the service in the specified period, the monitor will remove or disable the server in the scheduling table of the load balancer. The load balancer will not schedule new connections to the failed one and the failure of server nodes can be masked. In order to prevent the load balancer from becoming a single-point-of-failure, a backup of the load balancer is set-up. Two heartbeat daemons run on the primary and the backup, they heartbeat the health message through heartbeat channels such as serial line and UDP periodically. When the heartbeat daemon on the backup cannot hear the health message from the primary in the specified time, it will use ARP spoofing to take over the virtual IP address to provide the load balancing service. When the primary recovers from its failure, then the primary becomes the backup of the functioning load balancer, or the daemon receives the health message from the primary and releases the virtual IP address, and the primary will take over the virtual IP address. The failover or the take-over of the primary will cause the established connection in the state table lost, which will require the clients to send their requests again.

E. Pros and Cons of Each Level

Each level of SSI has its own pros and cons. The hardware-level SSI can offer the highest level of transparency, but due to its rigid architecture, it does not offer the flexibility required during the extension and enhancement of the system. The kernel-level approach offers full SSI to all users (application developers and end-users). However, kernel-level cluster technology is expensive to develop and maintain, as its market-share

is/will be probably limited and it is difficult to keep pace with technological innovations emerging into mass-market operating systems.

An application-level approach helps realise SSI partially and requires that each application be developed as SSI-aware separately. A key advantage of application-level SSI compared to the kernel-level is that it can be realised in stages and the user can benefit from it immediately. Whereas, in the kernel-level approach, unless all components are specifically developed to support SSI, cannot be used or released to the market. Due to this, kernel-level approach appears as a risky and economically non-viable approach. The middleware approach is a compromise between the other two mechanisms used to provide SSI. In some cases, like in PVM, each application needs to be implemented using special APIs on a case by-case basis. This means, there is a higher cost of implementation and maintenance, otherwise the user cannot get any benefit from the cluster. The arguments on the, so-called, "underware" versus middleware level of SSI are presented in [5].

IV CONCLUSIONS

SSI can greatly enhance the acceptability and usability of clusters by hiding the physical existence of multiple independent computers by presenting them as a single, unified resource. SSI can be realised either using hardware or software techniques, each of them have their own advantages and disadvantages. The middleware approach appears to offer an economy of scale compared to other approaches although it cannot offer full SSI like the OS approach. In any case, the designers of software (system or application) for clusters must always consider SSI (transparency) as one of their important design goals in addition to scalable performance and enhanced availability.

REFERENCES

[1] R. Buyya (editor), High Performance Cluster Computing: Architectures and Systems, Vol. 1, *Prentice Hall*, NJ, USA, 1999.

[2] G.F. Pfister, In Search of Clusters, Second Edition, Prentice Hall, NJ, USA, 1998.

[3] K. Hwang, H. Jin, E. Chow, C-L Wang, and Z. Xu, Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space, *IEEE Concurrency*, 7(1), January-March, 1999.

[4] K. Hwang, and Z. Xu, Scalable Parallel Computing – Technology, Architecture, Programming, *WCB/McGraw-Hill*, USA, 1998.

[5] B. Walker and D. Steel, Implementing a Full Single System Image UnixWare Cluster: Middleware vs. Underware, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, USA, 1999.

[6] B. Walker and D. Steel, Implementing a Full Single System Image UnixWare Cluster: Middleware vs Underware, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, USA, July 1999. http://www.sco.com/products/clustering/nscwhtpaper/

[7] G. Popek and B. Walker (Ed.), The Locus Distributed System Architecture, *MIT Press*, 1996

[8] Memory Channel, http://www.digital.com/info/hpc/systems/symc.html

[9] Distributed Shared Memory: http://www.cs.umd.edu/~keleher/dsm.html

[10] K. Shirriff, Y. Khalidi, V. Matena, and J. Auban, Single-System Image: The Solaris MC Approach, *Proceedings of the International Conference on Parallel and DistributedProcessing Techniques and Applications (PDPTA'97)*, USA, July 1997.

[11] D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat and T. Anderson, GLUnix: A Global Layer Unix for a Network of Workstations, Journal of Software Practice and Experience, *John Wiley & Sons*, USA, July 1998. http://now.cs.berkeley.edu/Glunix/glunix.html

[12] A. Barak and O. La'adan, The MOSIX Multicomputer Operating System for High Performance Cluster Computing (ftp), Journal of Future Generation Computer Systems, *Elsevier Science Press*, March 1998. http://www.mosix.cs.huji.ac.il/

[13] A. Geist and V. Sunderam, PVM: A Framework for Parallel Distributed Computing, *Journal of Concurrency: Practice and Experience*, December 1990. http://www.epm.ornl.gov/pvm/

[14] F. Ferstl, Job and Resource Management Systems, High Performance Cluster Computing: Architectures and Systems, Vol. 1, *Prentice Hall*, NJ, USA, 1999. http://www.gridware.com/product/codine.htm

[15] M. Litzkow, M. Livny, and M. Mutka, Condor – A Hunter of Idle Workstations, *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988. http://www.cs.wisc.edu/condor/

[16] Rajkumar Buyya, PARMON: A Portable and Scalable Monitoring System for Clusters, Journal of Software: Practice & Experience, *John Wiley & Sons*, USA, June 2000. http://www.csse.monash.edu.au/~rajkumar/parmon/

[17] W. Zhang, Linux Virtual Servers for Scalable Network Services, Ottawa Linux Symposium 2000, Canada. http://www.LinuxVirtualServer.org/

[18] M. Ma, C. Wang and F. Lau, JESSICA: Java-Enabled Single-System-Image Computing Architecture, *Journal of Parallel and Distributed Computing* (to appear), http://www.srg.csis.hku.hk/jessica.htm.