Symmetric Active/Active High Availability for High-Performance Computing System Services

Christian Engelmann[1,2] and Stephen L. Scott[1]
[1]Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA
[2]Department of Computer Science, The University of Reading, Reading, UK
Email: {engelmannc, scottsl}@ornl.gov

Chokchai (Box) Leangsuksun
Computer Science Department, Louisiana Tech University, Ruston, LA, USA
Email: box@latech.edu

Xubin (Ben) He
Department of Electrical and Computer Engineering, Tennessee Tech University, Cookeville, TN, USA
Email: hexb@tntech.edu

Abstract— **This work aims to pave the way for high availability in high-performance computing (HPC) by focusing on efficient redundancy strategies for head and service nodes. These nodes represent single points of failure and control for an entire HPC system as they render it inaccessible and unmanageable in case of a failure until repair. The presented approach introduces two distinct replication methods, internal and external, for providing symmetric active/active high availability for multiple redundant head and service nodes running in virtual synchrony utilizing an existing process group communication system for service group membership management and reliable, totally ordered message delivery. Resented results of a prototype implementation that offers symmetric active/active replication for HPC job and resource management using external replication show that the highest level of availability can be provided with an acceptable performance trade-off.**

Index Terms— **high-performance computing, high availability, virtual synchrony, group communication**

I. Introduction

During the last decade, high-performance computing (HPC) has become an important tool for scientists worldwide to understand problems, such as in climate dynamics, nuclear astrophysics, fusion energy, nanotechnology, and human genomics. Computer simulations of real-world and theoretical experiments exploiting multi-processor parallelism on a large scale using mathematical models have provided us with the advantage to gain scientific knowledge without the immediate need or capability of performing physical experiments.

Every year, new larger scale HPC systems emerge on the market with better raw performance capabilities. This growth in system scale poses a substantial challenge for system software and scientific applications with respect to reliability, availability and serviceability (RAS).

With only very few exceptions, the availability of recently installed HPC systems has been much lower in comparison to the same deployment phase of their predecessors. As a result, HPC centers may artificially set allowable job run time to very low numbers in order to force an application to store intermediate results, essentially a forced checkpoint, as insurance against lost computation time on long running jobs. However, this forced checkpoint itself wastes valuable computation time and resources as it does not produce scientific results.

In contrast to the experienced loss of availability, the demand for continuous availability has risen dramatically with the recent trend towards capability computing, which drives the race for scientific discovery by running applications on the fastest machines available while desiring significant amounts of time (weeks and months) without interruption. These HPC systems must be able to run in the event of frequent failures in such a manner that the capability is not severely degraded.

Both, the telecommunication and the general IT community, have dealt with these issues and have been able to provide high-level RAS using traditional high availability concepts, such as active/standby, for some time now. It is time for the HPC community to follow the IT and telecommunication industry lead and provide high-level RAS for HPC centers.

This work aims to pave the way for high availability in HPC by focusing on efficient redundancy strategies

This paper is based on "Active/Active Replication for Highly Available HPC System Services," by C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He, which appeared in the Proceedings of the 1^{st} International Conference on Availability, Reliability and Security (ARES), Vienna, Austria, April 2006, and on "JOSHUA: Symmetric Active/Active Replication for Highly Available HPC Job and Resource Management," by K. Uhlemann, C. Engelmann, and S. L. Scott, which appeared in the Proceedings of the International Conference on Cluster Computing (Cluster), Barcelona, Spain, September 2006. © 2006 IEEE.

© 2006 ACADEMY PUBLISHER

for head and service nodes. These nodes represent single points of failure and control for an entire HPC system as they render it inaccessible and unmanageable in case of a failure until repair.

The approach presented in this paper introduces two distinct replication methods, internal and external, for providing symmetric active/active high availability for multiple redundant head and service nodes running in virtual synchrony. We utilize an existing process group communication system for service membership management and reliable, totally ordered message delivery in oder to provide symmetric active/active replication.

This paper is structured as follows. First, we illustrate the overall background of the conducted research, including a more detailed problem description, conceptual models, and a review of existing solutions. Second, we describe two different replication methods for providing symmetric active/active high availability for HPC system services. Third, we present results of a prototype implementation that offers symmetric active/active replication for HPC job and resource management. We conclude with a short summary of the presented research and a brief overview of future work.

II. Background

High-performance computing has its historical roots in *parallel and distributed computing*, which is based on the general idea of solving a problem faster using more than one processor [1], [2]. While distributed computing takes a decentralized approach, parallel computing uses the opposite centralized concept. Both are extremes in a spectrum of *concurrent computing* with everything in-between. For example, a distributed computer system may be loosely coupled, but it is parallel.

Parallel and distributed computing on a large scale is commonly referred to as *supercomputing* or *high-performance computing* (HPC). Today's supercomputers are typically parallel architectures that have some distributed features. They scale from a few hundred processors to more than a hundred thousand. The elite of supercomputing systems, *i.e.*, the fastest systems in the world that appear in the upper ranks of the Top 500 List of Supercomputer Sites [3], are referred to as *high-end computing* (HEC) systems, or ultra-scale HEC systems due to the number of processors they employ.

Scientific computing (or *Computational science*) is the field of study concerned with constructing mathematical models and numerical solution techniques, and using computers to analyze and solve scientific and engineering problems [4], [5]. It utilizes computer simulations and other forms of computation to solve problems in various scientific disciplines, such as in climate dynamics, nuclear astrophysics, fusion energy, nanotechnology, and human genomics. *Ultra-scale scientific high-end computing* exploits multi-processor parallelism on a large scale for scientific discovery using the fastest machines available for days, weeks, or even months at a time.

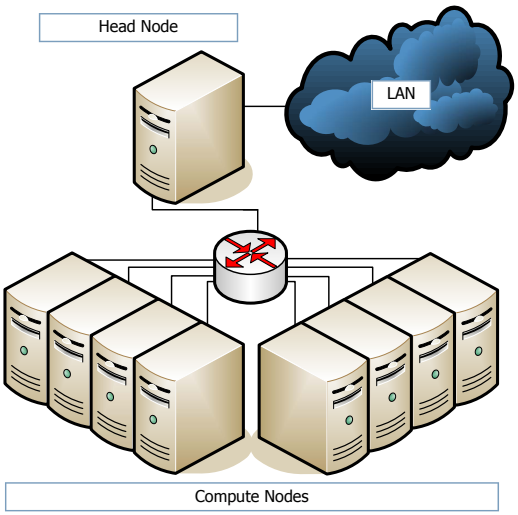

Figure 1. Beowulf Cluster Architecture with Single Head Node (© 2006 IEEE)

For example, the Terascale Supernova Initiative [6] is a multidisciplinary collaborative project that aims to develop models for core collapse supernovae and enabling technologies in radiation transport, radiation hydrodynamics, nuclear structure, linear systems and eigenvalue solution, and collaborative visualization. Recent breakthrough accomplishments include a series of 3D hydrodynamic simulations that show the extent of a supernova shock.

III. System Architecture

The emergence of cluster computing in the late 90's made scientific computing affordable to everyone, while it introduced the *Beowulf* cluster system architecture [7], [8] (Figure 1) with its single head node controlling a set of dedicated compute nodes. This parallel architecture has been proven to be very efficient as it permits customization of nodes and interconnects to their purpose. Many supercomputing vendors adopted the Beowulf architecture either completely in the form of HPC clusters or in part by developing hybrid HPC solutions.

A HPC system typically consists of several nodes, where each node has at least one processor, some memory and at least one network interface. While a significant number of *compute nodes* perform the actual parallel scientific computation, a single *head node* and optional *service nodes* handle system management tasks, such as user login, resource management, job scheduling, data storage, and I/O. A separate network interconnect for system management may exist permitting scientific applications running on compute nodes to communicate without interference. Furthermore, multi-processor nodes may be used to significantly reduce communication latency.

In order to improve scalability and to enable isolation in case of failures, large-scale HPC systems may be partitioned, so that several sets of *partition compute nodes* are supported by their respective *partition service nodes*. A single head node is still controlling the entire system, but individual responsibilities, such as data storage and

© 2006 ACADEMY PUBLISHER

I/O, are delegated to partition service nodes to balance the ratio of the number of compute nodes per service node and allow for partition outages.

IV. Critical System Services

HPC systems run *critical* and *non-critical system services* on head, service, and compute nodes.

A service is *critical* to its system if it can't operate without it. Any such service is a single point of failure and control for the entire system. As long as one of them is down, the entire system is not available. Critical system services may involve a set of distributed communicating processes, and they may cause a loss of system and application state in case of a failure. If a critical system service depends on another service, this other service is an additional point of failure and control for the critical system service and therefore also a critical system service by itself. Interdependent critical system services do not necessarily reside at the same physical location, *i.e.*, not on the same node. Any node and any network connection a critical system service depends on is an additional point of failure and control for the critical system service and therefore also for the entire system.

A service is *non-critical* to its system if it can operate without it in a degraded mode. Any such service is still a single point of failure for the entire system. Non-critical system services may also cause a loss of system and application state in case of a failure.

Typical critical HPC system services are: user login, network file system, job and resource management, communication services, and in some cases the OS, *e.g.*, for single system image (SSI) systems. User management, software management, and programming environment are usually non-critical system services.

V. Single Points of Failure and Control

If a system has a *head node running critical system services*, this head node is a single point of failure and control for the entire system. As long as it is down, the entire system is not available. A head node failure may cause a loss of system and application state. A typical head node on HPC systems may run the following critical system services: user login, job and resource management, and network file system. It may also run the following non-critical services: user management, software management, and programming environment.

Most HPC systems employ a head node, such as clusters, vector machines, massively parallel processing (MPP) systems, and SSI solutions. Examples are: Cray X1 [9], Cray XT3 [10], IBM Blue Gene/L [11], IBM MareNostrum [12], and SGI Altix [13].

If a system employs *service nodes running critical system services*, any such service node is a single point of failure and control for the entire system. As long as one of them is down, the entire system is not available. Similar to a head node failure, a service node failure may cause a loss of system and application state. If a system has *service nodes running non-critical system services*, any such service node is a single point of failure for the entire system. A failure of a service node running non-critical system services may still cause a loss of system and application state.

Service nodes typically offload head node system services, *i.e.*, they may run the same critical and non-critical system services. Most of the advanced HPC systems currently in use employ service nodes, *e.g.*, Cray X1, Cray XT3, IBM Blue Gene/L, and IBM MareNostrum.

VI. Availability Taxonomy

In the following, principles, assumptions and techniques employed in providing high availability in modern systems are explained. The content of this section is based on earlier work in refining a modern high availability taxonomy for generic computing systems [14], [15]. As part of this research effort [16], it has been adopted to the complexity of HPC system architectures.

A. Faults, Failures, and Outages

Conceptually, a *fault* can be described as an unexpected behavior of a system and can be classified as reproducible or non-reproducible. While a fault is any unexpected non-compliance within the system, a *failure* is a fault that is externally visible to the end user. The terms fault and failure are often used synonymously when a distinction between visible and non-visible faults can't be observed.

High availability computing does not make this distinction as a software system can only preempt or react to faults it already expects and can detect, *i.e.*, faults that are visible either directly as *abnormal* system behavior or through *detection* and *event propagation*. There are many different causes for failures in HPC systems:

- *Design errors* can cause failures, since the system is not designed to correctly perform the expected behavior. Programming errors fit under this description as well as any discrepancies between implementation, specification, and use cases. Such failures are reproducible if the software itself is deterministic.
- *System overloading* can cause failures, since the system is being used in a way that exceeds its resources to perform correctly. An example is a denial-of-service attack. System overloading can also be triggered by using a system beyond its specification, *i.e.*, by a preceding design error.
- *Wearing down* can cause failures, since the system is exposed to mechanical or thermal stress. Typical examples are power supply, processor, cooling fan, and disk failures. They are typically non-reproducible with the original hardware as it gets rendered unusable during the failure.
- *Preemptive, protective measures* of the system can also cause failures if the system forces a failure in order to prevent permanent or more extensive damage. For example, a system may automatically shut down if its processor heat monitor reports unusually high temperature readings.

© 2006 ACADEMY PUBLISHER

- *Other causes* for failures exist, such as race conditions between processes, distributed deadlocks, and network transmission errors.
- Further causes are *catastrophic events*, such as flood, hurricane, tornado, and terrorist attack.

There is a differentiation between *hard errors* and *soft errors*. A hard error is related to the failure of a hardware component that needs replacement, such as a failed memory chip. A soft error is related to software or data corruption, such as a bit flip in a memory chip, which can be resolved by simply restarting the system.

There is also a further distinction between *benign* and *malicious* (or *Byzantine*) failures. While a benign failure, such as a disk crash, is an easily detectable event, a malicious failure, like in the *Byzantine Generals Problem* [17] or the earlier mentioned denial-of-service attack, follows a malevolent plan. Extensive failure detection mechanisms, like network intrusion detection for example, often use probabilistic approaches to identify such failures.

The term *outage* is used to describe any kind of deviation from specified system behavior, whether it is expected or unexpected. All faults and failures can be categorized as *unplanned outages*, while intentional prevention of delivering specified system functionality, such as to perform maintenance operations, software upgrades, etc., are *planned outages*.

B. Availability Domains and Configurations

Availability is a system property that can be generalized into the following three distinctive domains.

1) Basic Availability: A system which is designed, implemented and deployed with sufficient components (hardware, software, and procedures) to satisfy its functional requirements, but no more, has *basic availability*. It will deliver the correct functionality as long as no failure occurs and no maintenance operations are performed. In case of failures or maintenance operations, a system outage may be observed.

2) High Availability: A system that additionally has sufficient *redundancy* in components (hardware, software, and procedures) to mask certain defined failures, has *high availability*. There is a continuum of high availability configurations with this definition due to the ambiguity of the terms "sufficient", "mask", and "certain". A clarification of these ambiguous terms follows.

"Sufficient", is a reflection of the system's requirements to tolerate failures. For computing systems, this typically implies a particular level of hardware and software redundancy that guarantees a specific quality of service.

"Certain", is a recognition of the fact that not all failures can or need to be masked. Typically, high availability solutions mask the most likely failures. However, mission critical applications (military, banking, and telecommunication) may mask even catastrophic failures.

"Masking" a fault implies shielding its external observation, *i.e.*, preventing the occurrence of a failure. Since faults are defined as an unexpected deviation from specified behavior, masking a fault means that no deviations (or only precisely defined deviations) from specified behavior may occur. This is invariably achieved through a replication mechanism appropriate to the component, a redundancy strategy. When a component fails, the redundant component replaces it. The degree of transparency in which this replacement occurs can lead to a wide variation of configurations:

- *Manual masking* requires human intervention to put the redundant component into service.
- *Cold Standby* requires an automatic procedure to put the redundant component into service, while service is interrupted and component state is lost. A cold standby solution typically provides hardware redundancy, but not software redundancy.
- *Warm Standby* requires some component state replication and an automatic fail-over procedure from the failed to the redundant component. The service is interrupted and some service state is lost. A warm standby solution typically provides hardware redundancy as well as some software redundancy. Component state is regularly replicated to the redundant component. In case of a failure, it replaces the failed one and continues to operate based on the previously replicated state. Only those component state changes are lost that occurred between the last replication and the failure.
- *Hot Standby* requires full component state replication and an automatic fail-over procedure from the failed to the redundant component. The service is interrupted, but no component state is lost. A hot-standby solution provides hardware redundancy as well as software redundancy. However, component state is replicated to the redundant component on any change, *i.e.*, it is always up-to-date. In case of a failure, it replaces the failed one and continues to operate based on the current state.

Manual masking is a rarely employed configuration for computing systems as it needs human intervention. Cold, warm and hot standby are *active/standby* configurations commonly used in high availability computing. The number of standby components may be increased to tolerate more than one failure at a time. The following *active/active* configurations require more than one redundant system component to be active, *i.e.*, to accept and execute state change requests.

- *Asymmetric active/active* requires two or more active components that offer the same capabilities at tandem without coordination, while optional standby components may replace failing active components ($n+1$ and $n+m$). Asymmetric active/active provides high availability with improved throughput performance. While it is heavily used in the telecommunication sector for stateless components, it has limited use cases due to the missing coordination between active components.
- *Symmetric active/active* requires two or more active components that offer the same capabilities and

© 2006 ACADEMY PUBLISHER

maintain a common global component state using *virtual synchrony* [18] or *distributed control* [19], *i.e.*, using a state change commit protocol. There is no interruption of service and no loss of state, since active services run in virtual synchrony without the need to failover. Symmetric active/active is superior in many areas, but is significantly more complex due to need for advanced commit protocols.

3) Continuous Availability: A system that has high availability properties and additionally applies these to planned outages as well, has *continuous availability*. This implies a masking strategy for planned outages, such as maintenance. Furthermore, a service interruption introduced by a high availability solution is a planned outage that needs to be dealt with as well. Continuous availability requires complete masking of all outages.

Application areas are typically mission critical, *i.e.*, in the military, banking, and telecommunication sector. Employed technologies range from hot standby with transparent fail-over and multiple standbys to active/active.

C. Availability Metrics

A systems availability can be between 0 and 1 (or 0% and 100% respectively), where 0 stands for no availability, *i.e.*, the system is inoperable, and 1 means continuous availability, *i.e.*, the system does not have any outages. Availability, in the simplest form, describes a ratio of system uptime t_{up} and downtime t_{down}:

$$A = \frac{t_{up}}{t_{up} + t_{down}} \tag{1}$$

When only considering unplanned outages, availability can be calculated (Equation 2) based on a systems mean time to failure ($MTTF$) and mean time to recover ($MTTR$). While the $MTTF$ is the average interval of time that a system will operate before a failure occurs, the $MTTR$ of a system is the average amount of time needed to repair, recover, or otherwise restore its service. However, there is a distinction between $MTTF$ and mean time between failures ($MTBF$), which is the average interval of time in which any failure occurs again, *i.e.*, the inverse of the failure frequency. A systems $MTBF$ covers both, $MTTF$ and $MTTR$ (Equation 3). The estimated annual downtime of a system in terms of hours t_{down} can be calculated using its availability (Equation 4). Planned outages of a system may be considered by respectively adjusting its $MTTF$ and $MTTR$.

$$A = \frac{MTTF}{MTTF + MTTR} \tag{2}$$

$$MTBF = MTTF + MTTR \tag{3}$$

$$t_{down} = 8760 \cdot (1 - A) \tag{4}$$

Since a component can be described as a system in itself, the availability of a system depends on the availability of its individual components. System components can be coupled serial, *e.g.*, component 1 depends on component 2, or parallel, *e.g.*, component 3 is redundant to component 4. Availability of serial and parallel component blocks can be calculated as follows:

$$A_{serial} = \prod_{i=1}^{n} A_i \tag{5}$$

$$A_{parallel} = 1 - \prod_{i=1}^{n} (1 - A_i) \tag{6}$$

The more interdependent (serial) components a system has, the less availability it provides. The more redundant (parallel) components a system has, the more availability it offers. High availability systems are build upon adding redundant components to increase overall availability.

D. Fail-Stop

The *fail-stop* model assumes that system components, such as individual services, nodes, and communication links, fail by simply stopping. Employed failure detection mechanisms only react to hard errors or catastrophic soft errors. Redundancy solutions based on this model do not guarantee correctness if a failing system component violates this assumption by producing false output due to an occurring soft error or a system design error.

The research presented in this paper is entirely based on the fail-stop model.

VII. Existing Solutions

High availability solutions for head and service nodes of HPC systems are typically based on service-level replication techniques. If a service running on a head or service node fails, a redundant one running on another node takes over. This may imply a head/service node fail-over, where the failed node is completely replaced by the standby node, *i.e.*, the standby node assumes the network address of the failed node.

Individual high availability solutions are usually tied directly to the services they provide high availability for. Each solution uses its own failure detection mechanism and redundancy strategy.

A. Active/Standby using Shared Storage

The concept of using a shared storage device (Figure 2) for saving service state is a common technique for providing service-level high availability, but it has its pitfalls. Service state is saved on the shared storage device upon modification, while the standby service takes over in case of a failure of the active service. The standby service monitors the health of the active service using a heartbeat mechanism [20] and initiates the fail-over procedure. An extension of this technique uses a crosswise hot standby redundancy strategy in an asymmetric active/active fashion. In this case, both are active services and additional standby services for each other. In both cases, the MTTR depends on the heartbeat interval.

While the shared storage device is typically a RAID system and therefore highly available, it remains a single

© 2006 ACADEMY PUBLISHER

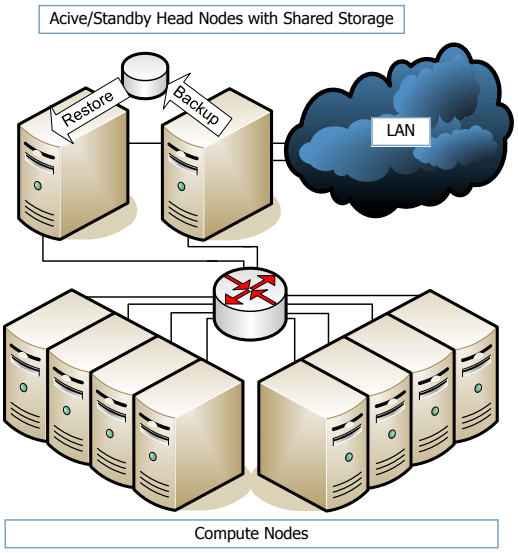

Figure 2. Enhanced Beowulf Cluster Architecture with Active/Standby High Availability for Head Node System Services using Shared Storage (© 2006 IEEE)

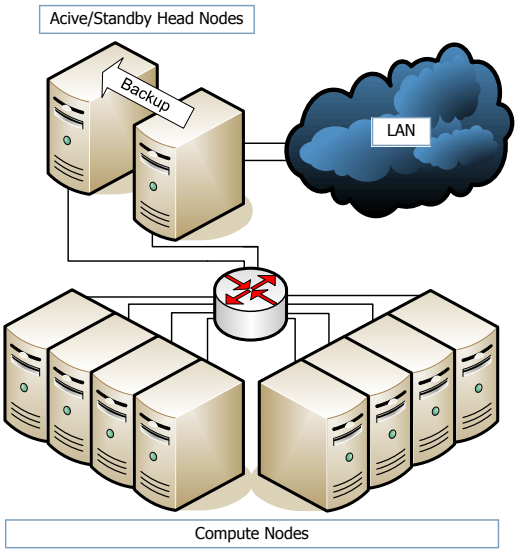

Figure 3. Enhanced Beowulf Cluster Architecture with Active/Standby High Availability for Head Node System Services (© 2006 IEEE)

point of failure and control. Furthermore, file system corruption due to failures occurring during write operations are not masked. Moreover, they lead to further corruption of the standby service state and to a subsequent failure (if detected at all). As with most shared storage solutions, correctness and quality of service are not guaranteed due to the lack of commit protocols.

The following solutions using a shared storage device exist for head and service nodes of HPC systems.

1) SLURM: The Simple Linux Utility for Resource Management (SLURM) [21], [22] is an open source and highly scalable HPC job and resource management system. SLURM is a critical system service running on the head node. It provides job and resource management for many HPC systems, *e.g.*, IBM Blue Gene/L. It offers high availability using a hot standby redundancy strategy with a standby head node and a shared storage device.

2) PVFS 2 Metadata Server: The Parallel Virtual File System (PVFS) 2 [23], [24] is a file system for HPC that utilizes parallel I/O in order to eliminate bottlenecks. One of the main components of any parallel file system is the metadata server (MDS), which keeps records of all stored files in form of a directory server. This MDS is a critical system service typically located on head or service nodes. PVFS 2 offers two high availability configurations for its MDS involving a standby node and a shared storage device, hot standby and asymmetric active/active.

3) Lustre Metadata Server: Similar to PVFS 2, Lustre [25], [26] is a scalable cluster file system for HPC. It runs in production on systems as small as 4 and as large as 15,000 compute nodes. Its MDS keeps records of all stored files in form of a directory server. This MDS is a critical system service typically located on head or service nodes. Lustre provides high availability for its MDS using an active/standby configuration with a shared storage.

B. Active/Standby

Service-level active/standby high availability solutions for head and service nodes in HPC systems (Figure 3) involve active and standby nodes. They use state commit protocols to maintain consistency of backup state. The following solutions using the active/standby configuration exist for head and service nodes of HPC systems.

1) HA-OSCAR: High Availability Open Source Cluster Application Resources (HA-OSCAR) [27]–[29] is a high availability framework for OpenPBS [30]. OpenPBS is the original version of the Portable Batch System (PBS), a flexible batch queuing system developed for NASA in the early to mid-1990s. Its service interface has become a standard in HPC job and resource management.

OpenPBS provides job and resource management for typical low- to mid-end HPC systems. It is a critical system service running on the head node. HA-OSCAR supports high availability for Open PBS using a warm standby redundancy strategy involving a standby head node. Service state is replicated to the standby upon modification, while the standby service takes over based on the current state. The standby node monitors the health of the active node using a heartbeat mechanism and initiates the fail-over. However, OpenPBS does temporarily loose control of the system in this case. All previously running jobs are automatically restarted.

The MTTR of HA-OSCAR depends on the heartbeat interval, the fail-over time, and the time currently running jobs need to recover to their previous state. HA-OSCAR integrates with the compute node checkpoint/restart layer for LAM/MPI [31], BLCR [32], improving its MTTR to 3-5 seconds for detection and fail-over plus the time to catch up based on the last application checkpoint.

2) PBS Pro on the Cray XT3: PBS is the leading job and resource management solution for HPC systems. Its professional edition, PBS Professional (PBS Pro) [33], operates in networked multi-platform UNIX environ-

© 2006 ACADEMY PUBLISHER

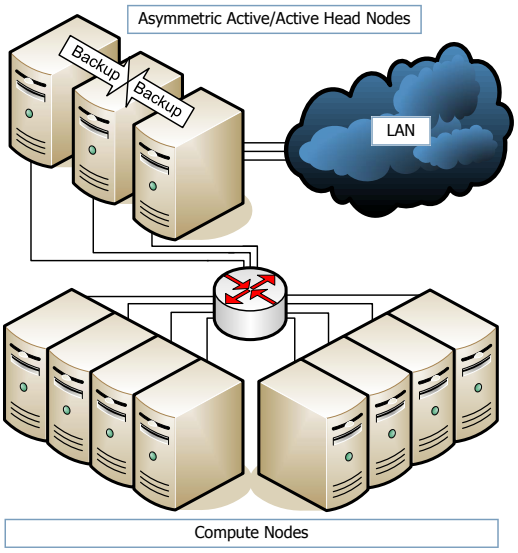

Figure 4. Enhanced Beowulf Cluster Architecture with Asymmetric Active/Active High Availability for Head Node System Services (© 2006 IEEE)

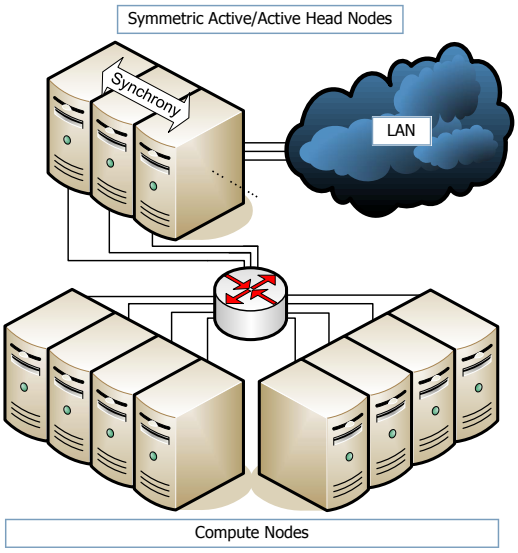

Figure 5. Enhanced Beowulf Cluster Architecture with Symmetric Active/Active High Availability for Head Node System Services (© 2006 IEEE)

ments, and supports heterogeneous clusters of workstations, supercomputers, and massively parallel systems.

PBS Pro for the Cray XT3 system [34] supports high availability using a hot standby redundancy strategy involving Crays proprietary interconnect for replication and transparent fail-over. Service state is replicated to the standby node, which takes over based on the current state without loosing control of the system. This solution has an MTTR of practically 0. However, it is only available for the Cray XT3 and its availability is limited by the deployment of two redundant nodes.

C. Asymmetric Active/Active

Similar to active/standby solutions, service-level asymmetric active/active high availability solutions with optional standby service(s) ($n+1$ or $n+m$) for head and service nodes in HPC systems (Figure 4) also perform state change commit protocols to maintain consistency of backup state. However, the backup states of all active services are kept separately.

At this moment, there is only one proof-of-concept prototype implementation of asymmetric active/active high availability for HPC system services, which has been developed in conjunction with this research effort.

1) Asymmetric Active/Active HA-OSCAR: As part of the HA-OSCAR research, an asymmetric active/active prototype implementation [35] has been developed that offers HPC job and resource management in a high-throughput computing scenario. Two different job and resource management services, OpenPBS and the Sun Grid Engine (SGE) [36], run independently on different head nodes at the same time, while an additional head node is configured as a standby. Fail-over is performed using a heartbeat mechanism and is guaranteed for only one service at a time using a priority-based fail-over policy. Similar to the active/standby HA-OSCAR variant,

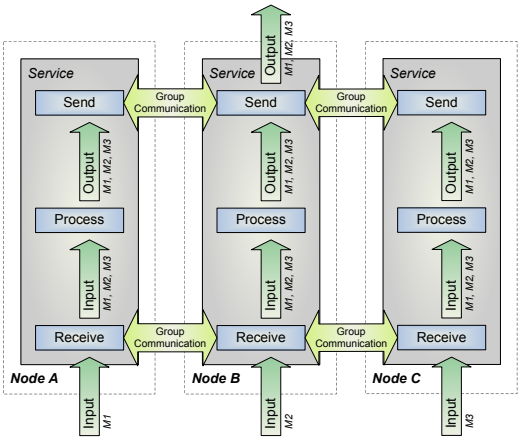

Figure 6. Symmetric Active/Active Replication Architecture (© 2006 IEEE)

OpenPBS and SGE do loose control of the system during fail-over, requiring a restart of lost jobs. Only one failure is completely masked at a time due to the $2+1$ configuration. A second failure results in a degraded operating mode with one head node serving the system.

D. Symmetric Active/Active

Work in symmetric active/active high availability solutions HPC system services (Figure 5) started only very recently as part of this research effort. The following sections describe the developed mechanisms as well as prototype implementation results.

VIII. Symmetric Active/Active Replication

The symmetric active/active high availability configuration for services (Figures 5, 6) allows more than one redundant service to be active, *i.e.*, to accept and perform state changes, while it does not waste system resources using idle standbys. Furthermore, there is no interruption

© 2006 ACADEMY PUBLISHER

of service and no loss of state, since active services run in virtual synchrony without the need to fail over.

Service state replication is performed by totally ordering all state change messages and reliably delivering them to all redundant active services. A process group communication system is used to ensure total message order and reliable message delivery as well as service group membership management. Furthermore, consistent output produced by all active services, *i.e.*, messages sent to other parts of the system or return messages related to service state changes, may be routed through the group communication system, using it for a distributed mutual exclusion to ensure that output is delivered only once. This is only necessary if the receiving component is unable to ignore duplicate return messages.

The number of active services is variable at runtime and can be changed by either forcing an active service to leave the service group or by joining a new service with the service group. This allows adaptation of the service group size, such as adding new nodes when old nodes become less reliable. This also permits reconfiguration of the service group behavior, for example when performing live upgrades in order to fix discovered vulnerabilities or improve throughput performance.

As long as one active service is alive, state is never lost, state changes can be performed and output is produced accordingly to state changes.

Implementing symmetric active/active replication using virtual synchrony supported by a group communication system implies event-based programming, where a service only reacts to event messages using uninterruptible event handler routines. More advanced programming models for virtual synchrony, such as distributed control [37], use the replicated remote procedure call abstraction to provide a request/response programming model.

However, both programming models assume that a service supplies the necessary hooks to perform uninterruptible state transitions. While this is typically the case for networked services that have some sort of event notification, remote procedure call or remote method invocation interface, command line based services, such as the batch job scheduler in HPC systems, and proprietary network services, such as data storage and I/O in HPC systems, do not necessarily offer these hooks.

Adaptation to the event-based or request/response programming model can be performed either internally by modifying the service itself or externally by wrapping it into a virtually synchronous environment [38].

A. Internal Replication

Internal replication (Figure 7) allows each active service of a service group to accept external state change requests individually, while using a group communication system for total message order and reliable message delivery to all members of the service group. All state changes are performed in the same order at all services, thus virtual synchrony is given.

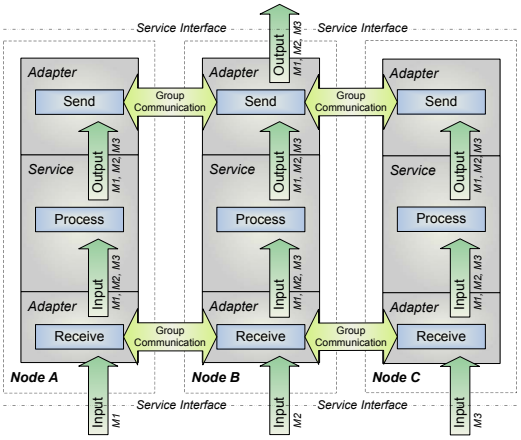

Figure 7. Symmetric Active/Active Replication Architecture using Internal Replication by Service Modification/Adaptation (© 2006 IEEE)

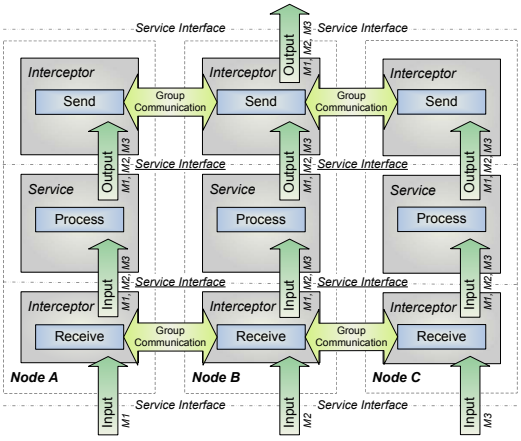

Figure 8. Symmetric Active/Active Replication Architecture using External Replication by Service Interface Utilization (© 2006 IEEE)

For example, a networked server that changes its state based on incoming remote procedure calls, such as a parallel file system metadata server, is modified to replicate all state changes in form of messages to all services in the service group. Upon delivery, state changes are performed in virtual synchrony. Remote procedure calls and respective state changes are executed by separate event handler routines.

This method requires modification of existing code, which may be unsuitable for complex and/or large services. The amount of modification necessary may result in a complete redesign and reimplementation. However, adaptation of the service to the event-based programming model of the group communication system may lead to performance enhancements as internal replication implies fine-grain synchronization of state changes.

B. External Replication

External replication (Figure 8) avoids modification of existing code by wrapping a service into a virtually synchronous environment. Interaction with other services or with the user is intercepted, totally ordered and reliably delivered to the service group using a group commu-

© 2006 ACADEMY PUBLISHER

nication system that mimics the service interface using separate event handler routines.

For example, the command line interface of a service is replaced with an interceptor command that behaves like the original, but forwards all input to an interceptor group. Once totally ordered and reliably delivered, each interceptor group member calls the original command to perform operations at each service group member. Service group output may be routed through the interceptor group for at most once delivery.

This method wraps an existing solution into a virtually synchronous environment without modifying it, which allows reusing the same solution for different services with the same interface. However, the missing adaptation of the service to the event-based programming model of the group communication system may lead to performance degradation as external replication implies coarse-grain synchronization of state changes.

C. Expected Performance

Both methods result in a performance impact in comparison to a single service solution. Based on existing experience with group communication systems [39], [40], both methods increase the response latency and may limit throughput performance. It is expected that internal replication offers overall better performance due to optimizations within the service. However, the overall performance highly depends on the group communication system and its commit protocol.

For both methods, non-conflicting internal replication state changes or external replication service interface operations may be interleaved in order to gain latency and throughput performance. Furthermore, state changes or service interface operations may be split up into smaller atomic operations for fine-grain interleaving.

Overall, we recommend the internal replication method when low latency and high throughput performance is needed, except where the prospect of extensive code modification or foreseeable major service design changes prohibits it. However, we recommend using external replication when the symmetric active/active high availability solution should be reusable for other services with the same or a similar interface.

D. Continuous Availability

Since symmetric active/active replication provides high availability without interruption of service in case of a failure, using its mechanisms for planned outages as well, such as for *live upgrades* and other maintenance, would result in providing continuous availability.

Servicing or upgrading a highly available system service while it is running in a symmetric active/active fashion requires removal, maintenance/upgrade and rejoin of each individual service group member, one at a time. A strict prerequisite of such a procedure is that the new version of the service fully supports the interface of the old version.

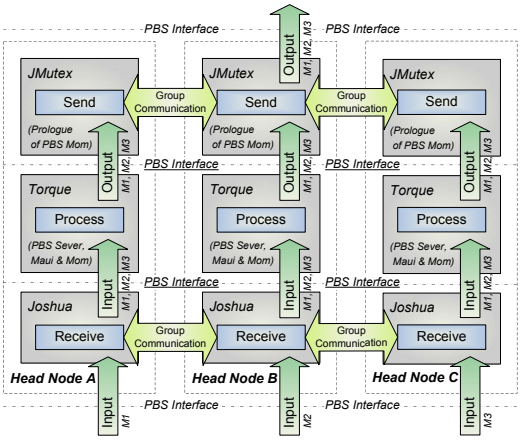

Figure 9. Symmetric Active/Active High Availability Architecture of the JOSHUA solution for HPC Job and Resource Management (© 2006 IEEE)

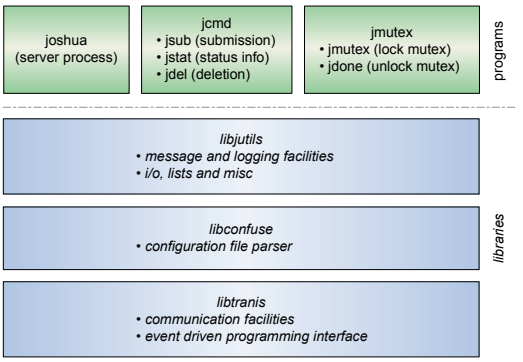

Figure 10. Individual Software Components of the JOSHUA Solution for Highly Available HPC Job and Resource Management (© 2006 IEEE)

Maintaining a consistent interface between service and group communication system over a certain period of time is easier using external replication as it is based on the service interface, while the internal design of the service is not affected and may change with a new version.

IX. Prototype Implementation

The JOSHUA solution [41], [42] offers symmetric active/active high availability for HPC job and resource management services with a PBS compliant service interface. It represents a virtually synchronous environment using external replication based on the PBS service interface (Figure 9) providing high availability without any interruption of service and without any loss of state.

A. Software Architecture

Conceptually, the JOSHUA software architecture (Figure 10) consists of three major parts: a server process (joshua) running on each head node, a set of control commands (jsub, jdel, and jstat) reflecting PBS compliant behavior to the user, and a set of scripts (jmutex and jdone) to perform a distributed mutual exclusion during job launch. Furthermore, JOSHUA relies on the Transis [43], [44] group communication system for reliable, totally ordered message delivery.

© 2006 ACADEMY PUBLISHER

System	#	Latency	Overhead
TORQUE	1	98ms	
JOSHUA/TORQUE	1	134ms	36ms / 37%
JOSHUA/TORQUE	2	265ms	158ms /161%
JOSHUA/TORQUE	3	304ms	206ms /210%
JOSHUA/TORQUE	4	349ms	251ms /256%

TABLE I.
JOB SUBMISSION LATENCY RESULTS FOR JOSHUA (© 2006 IEEE)

System	#	10 Jobs	50 Jobs	100 Jobs
TORQUE	1	0.93s	4.95s	10.18s
JOSHUA/TORQUE	1	1.32s	6.48s	14.08s
JOSHUA/TORQUE	2	2.68s	13.09s	26.37s
JOSHUA/TORQUE	3	2.93s	15.91s	30.03s
JOSHUA/TORQUE	4	3.62s	17.65s	33.32s

TABLE II.
JOB SUBMISSION THROUGHPUT RESULTS FOR JOSHUA (© 2006 IEEE)

The JOSHUA prototype is based on the PBS compliant TORQUE [45] HPC job and resource management system that employs the TORQUE PBS server together with the Maui [46] scheduler on each active head node and a set of PBS mom servers on compute nodes.

B. Test Results

The JOSHUA v0.1 prototype implementation has been deployed on a dedicated Linux cluster for functional and performance testing. Each node contained dual Intel Pentium III (Katmai) 450MHz processors with 512 MB of memory and 8 GB of disk space. All nodes were connected via a single Fast Ethernet (100MBit/s full duplex) hub. Debian GNU/Linux 3.1 (sarge) has been used as operating system in conjunction with Transis v1.03, TORQUE v2.0p5, and Maui v3.2.6p13.

The job submission latency overhead (Table I) introduced by the network communication between the JOSHUA commands, the Transis group communication system and the JOSHUA server was in an acceptable range. The latency overhead between TORQUE and JOSHUA/TORQUE on a single head node (37%) can be attributed to communication on the same node, while the significant latency overhead increase between JOSHUA/TORQUE on a single and on two head nodes (439%) can be explained by off-node communication and the Transis commit protocol overhead. Overall a latency overhead of only 250ms for a 4 head node system is still acceptable for a HPC system.

The job submission throughput overhead of JOSHUA (Table II) reflected similar characteristics. Considering high throughput HPC scenarios, such as in computational biology or on-demand cluster computing, adding 100 jobs to the job queue in 33s for a 4 head node system is also an acceptable performance.

Using a rather low MTTF of 5000 hours and a MTTR of 72 hours for an individual head node, the expected downtime (Equations 2, 4) of a single head node is over 5 days within a year. An availability analysis (Table III) based on parallel system component coupling

#	Availability	Nines	Downtime/Year
1	98.6%	1	5d 4h 21min
2	99.98%	3	1h 45min
3	99.9997%	5	1min 30s
4	99.999996%	7	1s

(Based on MTTF=5000h and MTTR=72h)

TABLE III.
AVAILABILITY AND DOWNTIME/YEAR ESTIMATES FOR JOSHUA (© 2006 IEEE)

(Equation 6) shows that deploying the JOSHUA solution on two head nodes reduces the annual downtime to only 1 hour and 45 minutes with a latency overhead of only 158 milliseconds. Adding another head node decreases the downtime to 1 1/2 minutes with a latency overhead of 206 milliseconds. A 4 head node solution offers an availability of 7 nines with an annual downtime of 1 second and a still acceptable latency overhead of 251 milliseconds.

This analysis shows that the JOSHUA solution can provide the highest level of availability with an acceptable performance trade-off. However, it does not show the impact of correlated failures, such as caused by overheating of a rack or computer room. The deployment of redundant head nodes also needs to take into account these location dependent failure causes. Furthermore, an availability of 7 nines as provided by 4 active head nodes may not be a realistic target for a single HPC system as catastrophic events are not considered.

More extensive implementation details and test results of the JOSHUA solution for highly available HPC job and resource management can be found in earlier publications [41], [42].

X. CONCLUSION

With this paper, we presented our recent research accomplishments in providing the highest level of RAS for HPC systems. As a first step, we focused on efficient redundancy strategies for HPC system services running on head and service nodes. These nodes represent single points of failure and control for an entire HPC system as they render it inaccessible and unmanageable in case of a failure until repair.

We illustrated the overall background of the conducted research, including a more detailed problem description, conceptual models, and a review of existing solutions. We introduced two different replication methods, internal and external, for providing symmetric active/active high availability for multiple redundant nodes running in virtual synchrony utilizing an existing process group communication system for service group membership management and reliable, totally ordered message delivery.

Results of a prototype implementation that offers symmetric active/active replication for HPC job and resource management using external replication shows that the highest level of availability can be provided with an acceptable performance trade-off. The JOSHUA prototype

© 2006 ACADEMY PUBLISHER

is based on the TORQUE/Maui job and resource management combination and is able to provide an availability of 7 nines with an annual downtime of 1 second using a 4 head node system with a job submission latency overhead of 251ms (256%) and a job submission throughput performance of 100 jobs in 33.32s.

Ongoing work in this area focuses on providing symmetric active/active replication for other HPC system services, such as for the PVFS metadata server, as well as on performance enhancements and generalized high availability programming models.

Future research efforts will focus on providing high availability for compute nodes of HPC systems.

ACKNOWLEDGMENTS

This research was partially sponsored by the Mathematical, Information, and Computational Sciences Division; Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed in part at Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725. It was also performed in part at Louisiana Tech University under U.S. Department of Energy Grant No. DE-FG02-05ER25659.

The research performed at Tennessee Tech University was partially supported by the U.S. National Science Foundation under Grant No. CNS-0617528 and the Research Office of the Tennessee Tech University under a Faculty Research Grant.

The work at Oak Ridge National Laboratory and Tennessee Tech University was also partially sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory.

The presented research is part of the MOLAR [47], [48] project, which concentrates on adaptive, reliable, and efficient operating and runtime system solutions for ultra-scale scientific high-end computing as part of the Forum to Address Scalable Technology for Runtime and Operating Systems (FAST-OS) [49].

REFERENCES

[1] "Scientific Computing explained at Wikipedia," Available at http://en.wikipedia.org/wiki/Distributed_computing.

[2] S. Hariri and M. Parashar, *Tools and Environments for Parallel and Distributed Computing*. Wiley-Interscience, 2004.

[3] "Top 500 List of Supercomputer Sites," Available at http://www.top500.org.

[4] M. T. Heath, *Scientific Computing*. McGraw-Hill Higher Education, 2001.

[5] "Scientific Computing explained at Wikipedia," Available at http://en.wikipedia.org/wiki/Scientific_computing.

[6] "Terascale Supernova Initiative at Oak Ridge National Laboratory, Oak Ridge, TN, USA," Available at http://www.phy.ornl.gov/tsi.

[7] T. Sterling, *Beowulf cluster computing with Linux*. Cambridge, MA, USA: MIT Press, 2002.

[8] T. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese, *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. Cambridge, MA, USA: MIT Press, 1999.

[9] "X1 Computing Platform at Cray, Seattle, WA, USA," Available at http://www.cray.com/products/x1.

[10] "XT3 Computing Platform at Cray, Seattle, WA, USA," Available at http://www.cray.com/products/xt3.

[11] "Blue Gene/L Computing Platform at IBM Research," Available at http://www.research.ibm.com/bluegene.

[12] "MareNostrum eServer Computing Platform at IBM," Available at http://www.ibm.com/servers/eserver/linux/power/marenostrum.

[13] "Altix Computing Platform at SGI, Mountain View, CA, USA," Available at http://www.sgi.com/products/servers/altix.

[14] R. I. Resnick, "A modern taxonomy of high availability," 1996, Available at http://www.generalconcepts.com/resources/reliability/resnick/HA.htm.

[15] E. Vargas, "High availability fundamentals," *Sun Blueprints series*, Nov. 2000.

[16] C. Engelmann and S. L. Scott, "Concepts for high availability in scientific high-end computing," in *Proceedings of High Availability and Performance Workshop (HAPCW) 2005*, Santa Fe, NM, USA, Oct. 11, 2005.

[17] L. Lamport, R. E. Shostak, and M. C. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[18] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal, "Extended virtual synchrony," *Proceedings of IEEE* 14^{th} *International Conference on Distributed Computing Systems (ICDCS) 1994*, pp. 56–65, June 21-24, 1994.

[19] C. Engelmann, S. L. Scott, and G. A. Geist, "High availability through distributed control," in *Proceedings of High Availability and Performance Workshop (HAPCW) 2004*, Santa Fe, NM, USA, Oct. 12, 2004.

[20] "Heartbeat," Available at http://www.linux-ha.org/HeartbeatProgram.

[21] "SLURM at Lawrence Livermore National Laboratory, Livermore, CA, USA," Available at http://www.llnl.gov/linux/slurm.

[22] A. Yoo, M. Jette, and M. Grondona, "SLURM: Simple linux utility for resource management," in *Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, vol. 2862, Seattle, WA, USA, June 24, 2003, pp. 44–60.

[23] "Parallel Virtual File System (PVFS)," Available at http://www.pvfs.org/pvfs2.

[24] PVFS2 Development Team, "PVFS2 High-Availability Clustering," Available at http://www.pvfs.org/pvfs2 as part of the PVFS2 source distribution.

[25] "Lustre at Cluster File Systems, Inc., Boulder, CO, USA," Available at http://www.lustre.org.

[26] "Lustre Architecture Whitepaper at Cluster File Systems, Inc., Boulder, CO, USA," Available at http://www.lustre.org/docs/whitepaper.pdf.

[27] "HA-OSCAR at Louisiana Tech University, Ruston, LA, USA," Available at http://xcr.cenit.latech.edu/ha-oscar.

[28] I. Haddad, C. Leangsuksun, and S. L. Scott, "HA-OSCAR: Towards highly available linux clusters," *Linux World Magazine*, Mar. 2004.

[29] K. Limaye, C. Leangsuksun, Z. Greenwood, S. L. Scott, C. Engelmann, R. Libby, and K. Chanchio, "Job-site level fault tolerance for cluster and grid environments," in *Proceedings of IEEE International Conference on Cluster Computing (Cluster) 2005*, Boston, MA, USA, Sept. 26-30, 2005.

[30] "OpenPBS at Altair Engineering, Troy, MI, USA," Available at http://www.openpbs.org.

[31] "LAM-MPI Project at Indiana University, Bloomington, IN, USA," Available at http://www.lam-mpi.org.

© 2006 ACADEMY PUBLISHER

[32] "Berkeley Lab Checkpoint/Restart (BLCR) project at Lawrence Berkeley National Laboratory, Berkeley, CA, USA," Available at http://ftg.lbl.gov/checkpoint.
[33] "PBS Pro at Altair Engineering, Troy, MI, USA," Available at http://www.altair.com/software/pbspro.htm.
[34] "PBS Pro for the Cray XT3 Computing Platform at Altair Engineering, Troy, MI, USA," Available at http://www.altair.com/pdf/PBSPro_Cray.pdf.
[35] C. Leangsuksun, V. K. Munganuru, T. Liu, S. L. Scott, and C. Engelmann, "Asymmetric active-active high availability for high-end computing," in *Proceedings of* 2^{nd} *International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2) 2005*, Cambridge, MA, USA, June 19, 2005.
[36] "Sun Grid Engine (SGE) at Sun Microsystems, Inc, Santa Clara, CA, USA," Available http://gridengine.sunsource.net.
[37] C. Engelmann, S. L. Scott, and G. A. Geist, "Distributed peer-to-peer control in Harness," in *Lecture Notes in Computer Science: Proceedings of International Conference on Computational Science (ICCS) 2002, Part II*, vol. 2330, Amsterdam, The Netherlands, Apr. 21-24, 2002, pp. 720–727.
[38] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He, "Active/active replication for highly available HPC system services," in *Proceedings of International Symposium on Frontiers in Availability, Reliability and Security (FARES) 2006, in conjunction with* 1^{st} *International Conference on Availability, Reliability and Security (ARES) 2006*, Vienna, Austria, Apr. 20-22, 2006, pp. 639–645.
[39] X. Defago, A. Schiper, and P. Urban, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, 2004.
[40] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *ACM Computing Surveys*, vol. 33, no. 4, pp. 1–43, 2001.
[41] K. Uhlemann, C. Engelmann, and S. L. Scott, "JOSHUA: Symmetric active/active replication for highly available HPC job and resource management," in *Proceedings of IEEE International Conference on Cluster Computing (Cluster) 2006*, Barcelona, Spain, Sept. 25-28, 2006.
[42] K. Uhlemann, "High availability for high-end scientific computing," Master's thesis, Department of Computer Science, University of Reading, UK, Mar. 2006.
[43] D. Dolev and D. Malki, "The Transis approach to high availability cluster communication," *Communications of the ACM*, vol. 39, no. 4, pp. 64–70, 1996.
[44] "Transis Project at Hebrew University of Jerusalem, Israel," Available at http://www.cs.huji.ac.il/labs/transis.
[45] "TORQUE Resource Manager at Cluster Resources, Inc., Spanish Fork, UT, USA," Available at http://www.clusterresources.com/torque.
[46] "Maui Cluster Scheduler at Cluster Resources, Inc., Spanish Fork, UT, USA," Available at http://www.clusterresources.com/maui.
[47] C. Engelmann, S. L. Scott, D. E. Bernholdt, N. R. Gottumukkala, C. Leangsuksun, J. Varma, C. Wang, F. Mueller, A. G. Shet, and P. Sadayappan, "MOLAR: Adaptive runtime support for high-end computing operating and runtime systems," *ACM SIGOPS Operating Systems Review (OSR)*, vol. 40, no. 2, pp. 63–72, 2006.
[48] "Modular Linux and Adaptive Runtime Support for High-end Computing Operating and Runtime Systems (MOLAR)," Available at http://www.fastos.org/molar.
[49] "Forum to Address Scalable Technology for Runtime and Operating Systems (FAST-OS)," Available at http://www.fastos.org.

Christian Engelmann is a R&D Staff Member in the Network and Cluster Computing Group of the Computer Science and Mathematics Division at the Oak Ridge National Laboratory (ORNL). He is also a Research Assistant at the Department of Computer Science of the University of Reading. He received his MSc in Computer Science from the University of Reading and his German Certified Engineer diploma in Computer Systems Engineering from the Technical College for Engineering and Economics (FHTW) Berlin, both in 2001. He is currently a PhD student in Computer Science at the University of Reading. He is part of the MOLAR research team, where his work focuses on high availability for Linux clusters. He is also a contributor to the Harness HPC Workbench research of ORNL, the University of Tennessee, and Emory University in developing next-generation development tools, deployment mechanisms, and runtime environments. In the past, he was involved with the ORNL/IBM Blue Gene/L research initiative in super-scalable scientific algorithms for next generation supercomputing on systems with 100,000 processors. He was also a contributor to the Harness DVM research effort. He is a member of the ACM, the ACM SIGOPS, the IEEE, the IEEE Computer Society, and its Technical Committee on Scalable Computing and Task Force on Cluster Computing.

Dr. Stephen L. Scott is a Senior Research Scientist in the Network and Cluster Computing Group of the Computer Science and Mathematics Division at the Oak Ridge National Laboratory (ORNL). He received his MSc and PhD in Computer Science from Kent State University, Kent, Ohio, in 1992 and 1996 respectively. His research interest is in experimental systems with a focus on high performance distributed, heterogeneous, and parallel computing. He is a founding member of the Open Cluster Group (OCG) and Open Source Cluster Application Resources (OSCAR). He is presently the OCG steering committee chair and has served as the OSCAR release manager and working group chair. Dr. Scott is the lead principal investigator for the MOLAR research team. He is also the ORNL lead for the Scalable Systems Software project, which targets the development of technologies to scale cluster resources to 10,000's of processors. He is a member of the ACM, the IEEE, the IEEE Computer Society, and its Technical Committee on Scalable Computing and Task Force on Cluster Computing.

Dr. Chokchai (Box) Leangsuksun is an Associate Professor in Computer Science and the Center for Entrepreneurship and Information Technology (CEnIT) at Louisiana Tech University. He received his MSc and PhD in Computer Science from Kent State University, Kent, Ohio, in 1989 and 1995 respectively. His research interests include highly reliable and high performance computing, intelligent component based software engineering, parallel and distributed computing, service-oriented architectures, and service engineering and management.

Dr. Xubin (Ben) He is an Assistant Professor in the Department Electrical and Computer Engineering at the Tennessee Technological University. He received his PhD degree in Electrical Engineering from the University of Rhode Island, USA, in 2002 and both, his BSc and MSc, degrees in Computer Science from the Huazhong University of Science and Technology, China, in 1995 and 1997 respectively. His research interests include computer architecture, storage systems, high performance I/O systems, and performance evaluation. He received the Ralph E. Powe Junior Faculty Enhancement Award in 2004 and the TTU Chapter Sigma Xi Research Award in 2005. He is a member of the IEEE, the IEEE Computer Society, Sigma Xi, and ASEE.

© 2006 ACADEMY PUBLISHER