# MOLAR: Adaptive Runtime Support for High-End Computing Operating and Runtime Systems*

Christian Engelmann[1,4], Stephen L. Scott[1], David E. Bernholdt[1],
Narasimha R. Gottumukkala[2], Chokchai Leangsuksun[2], Jyothish Varma[3],
Chao Wang[3], Frank Mueller[3], Aniruddha G. Shet[5], P. Sadayappan[5]

[1]Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
[2]Louisiana Tech University, Ruston, LA 71272, USA
[3]North Carolina State University, Raleigh, NC 27695, USA
[4]University of Reading, Reading, Berkshire, RG6 6AH, UK
[5]The Ohio State University, Columbus, OH 43210, USA

{engelmannc,scottsl,bernholdtde}@ornl.gov, {nrg003,box}@latech.edu,
{jsvarma,wchao}@ncsu.edu, mueller@cs.ncsu.edu, {shet,saday}@cse.ohio-state.edu

http://www.fastos.org/molar

## ABSTRACT

MOLAR is a multi-institutional research effort that concentrates on adaptive, reliable, and efficient operating and runtime system (OS/R) solutions for ultra-scale high-end scientific computing on the next generation of supercomputers. This research addresses the challenges outlined in FAST-OS (forum to address scalable technology for runtime and operating systems) and HECRTF (high-end computing revitalization task force) activities by exploring the use of advanced monitoring and adaptation to improve application performance and predictability of system interruptions, and by advancing computer reliability, availability and serviceability (RAS) management systems to work cooperatively with the OS/R to identify and preemptively resolve system issues. This paper describes recent research of the MOLAR team in advancing RAS for high-end computing OS/Rs.

## Keywords

High-End Computing, RAS, Reliability, Availability, Fault Tolerance, Monitoring, Group Membership

## 1. OVERVIEW

Current operating systems and runtime systems (OS/Rs) for high-end scientific computing (HEC) are not able to meet the various requirements to run large applications efficiently on future ultra-scale computers. Building on the current open-source operating system, Linux, we target HEC applications for the next generation of supercomputers. Undoubtedly, these HEC OS/Rs must scale to the levels predicted by hardware architects for both shared memory and distributed memory platforms. Furthermore, they must enable applications to operate efficiently and reliably on any of these architectures as transparently as possible. As described in recent reports by FAST-OS [14] (Forum to Address Scalable Technology for Runtime and Operating Systems), HECRTF [24] (High-End Computing Revitalization Task Force) and ScaLeS [41] (Science Case for Large-scale Simulation) activities, system software is a key challenge in exploiting the promise of extreme-scale scientific computing. Conceptually, the MOLAR [15] research has the following goals to address these issues.

- Explore the use of advanced monitoring and adaptation to improve application performance and predictability of system interruptions.
- Advance computer reliability, availability and serviceability (RAS) management systems to work cooperatively with the OS/R to identify and preemptively resolve system issues.

As part of MOLAR, our research focuses on the development of a RAS-aware job and resource management solution for Linux clusters based on HA-OSCAR [21, 22], a high availability Linux clustering software suite. This work includes the design of a RAS-aware Federated System Management (fSM) for HA-OSCAR with partition-centric service and management nodes to provide highly available critical services for local and intra-partition requests, such as local and global job scheduling and monitoring. Furthermore, based on our experience with HA-OSCAR and Harness [6, 19, 35], a heterogeneous distributed metacomputing environment, the MOLAR team targets the development of a flexible, pluggable high availability framework that is capable of providing service-level active/hot-standby and active/active high availability to critical system services, such as user login, resource management, job scheduling, data storage and I/O. As part of this research, we aim to develop

*Research sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy (DOE). This work was performed in part at Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under DOE Contract No. DE-AC05-00OR22725, where it was also supported in part by the Laboratory's Directed Research and Development Program. This work was supported at Louisiana Tech University by DOE grant DE-FG02-04ER4614, at North Carolina State University in part by DOE grant DE-FG02-05ER25664 and NSF grants CAREER CCR-0237570 and CCF-0429653, and at the Ohio State University by DOE grant DE-FG02-05ER25660.

efficient, scalable algorithms for high availability without single points of failure and without single points of control.

With this paper, we report our recent research in advancing RAS for high-end computing OS/Rs. This paper is structured as follows. First, we present our work in performance instrumentation to characterize computation-communication overlap in message-passing systems. Second, we describe our efforts in devising a high availability taxonomy specifically adapted to HEC environments that takes into account that the purpose of a HEC system is to offer computing cycles for parallel applications, which requires a different approach than for single services, such as Web servers. Third, we portray our accomplishments in providing a reliability-aware job and resource management solution for the HA-OSCAR Linux clustering software suite. Fourth, we continue by illustrating results of ongoing work in developing a flexible, pluggable component-based high availability framework that allows adaptation to system properties and application needs. Fifth, we describe two replication methods for equipping existing critical HEC system services with service-level active/active high availability. Lastly, we show early results of developing a scalable fault-tolerant membership algorithm for MPI communication. This paper closes with a short summary of the presented research and a brief description of planned work.

# 2. PERFORMANCE INSTRUMENTATION AND MONITORING

MPI has been the *de facto* standard for writing parallel scientific applications. Available performance tools for MPI programs convey different kinds of insights using approaches like profiling, tracing or monitoring of hardware counters [29]. Current tools do not characterize the overlap between user computation and data communication for a running MPI program. In this work, we attempt to quantify computation-communication overlap by adding low-overhead instrumentation to the MPI library. The instrumentation has been prototyped in MVAPICH2 [33], a MPICH2 library [32] for InfiniBand.

## 2.1 Motivation

User-level networking architectures like InfiniBand [25] support OS-bypass communication that reduces the involvement of host CPU in the actual data transfer path and frees the CPU to do user computation instead. MPI libraries, like MVAPICH2, exploit RDMA support in modern networks to implement zero-copy transfers of large messages. The MPI interface defines non-blocking point-to-point calls like MPI_Isend and MPI_Irecv that provide a mechanism for seperating the initiation and completion of communication, thereby allowing application programmers to interleave useful computation in between. Thus, overlapping computation with communication using non-blocking calls can be an effective technique for masking the latency of data transfer, yielding significant performance gains and reducing execution time of parallel applications. However, previous studies [46, 34] have shown that MPICH-based MPI libraries achieve very low overlap for large messages even with non-blocking calls. It has been observed that the extent of overlap for an MPI program varies depending upon the type, frequency and order in which MPI calls are made. Therefore, instrumentation quantifying the overlap realized by a running application on a given system can be a valuable aid in explaining and improving its performance on that system.

## 2.2 Instrumentation Approach

Each process timestamps the initiation and completion of data transfer as well as the invocation and completion of MPI calls in the MPI library. Timestamps corresponding to invocation and completion of MPI calls serve to demarcate the user computation and MPI regions. Time for communicating data of different sizes on the network is either obtained apriori or projected from gathered times. The overlap obtained for a message is measured by tracing its communication time interval into user computation region. This calculation is done for all data communicated over the length of execution of a MPI program. Thus a per-process characterization of computation-communication overlap is obtained.

## 2.3 Implementation

A precise overlap characterization for MPICH-based libraries is not currently possible due to the following reasons:

1. NICs do not currently timestamp data transfers.
2. MPICH-based libraries have a single-threaded monolithic architecture, a polling type of progress engine and a synchronous message completion and notification mechanism.
3. Depending on the underlying protocol primitives, the initiation of communication may not be known to one of the participating processes.

Hence, we generate lower and upper bounds on the total communication time that was overlapped with user computation on a per-process basis. When the initiation of communication is transparent to a process, potentially the entire communication time may have been overlapped, but it is impossible to be conclusive; hence only the upper bound is updated. On the other hand, if a process can accurately timestamp communication, we update both the lower and upper bound values.

## 2.4 Experimental Results of Overlap Measurement

We ran NAS benchmarks from the NPB3.2 suite, compiled with instrumented MVAPICH2-0.6.5, for problem sizes A and B, on different numbers of processors of the Intel Pentium 4 cluster at the Ohio Supercomputer Center. This machine is a distributed/shared memory hybrid system constructed from commodity PC components running the Linux operating system. It has 112 compute nodes for parallel jobs, each configured with 4GB RAM, two 2.4GHz Intel P4 Xeon processors and one InfiniBand 10Gb interface. Tables 1 - 4 show the overlap measures for NAS BT, CG, LU and MG for problem size B. The results for the complete set of NAS benchmarks appear in [42]. The various measures are: 1) data xfer - the transfer time for all messages communicated (sent or received) by a process 2) min ovpd xfer - the lower bound on transfer time that was overlapped with user computation 3) max ovpd xfer - the upper bound on transfer

time that was overlapped with user computation 4) user comp - the aggregate user computation time 5) mpi call - the summation of time spent executing MPI calls.

| Time in sec | BT-B-4 | BT-B-9 | BT-B-16 |
|---|---|---|---|
| data xfer | 1.665141 | 1.525912 | 1.315637 |
| min ovpd xfer | 0.000028 | 0.004158 | 0.005585 |
| max ovpd xfer | 0.832581 | 0.76708 | 0.663369 |
| user comp | 235.606291 | 104.105301 | 58.433879 |
| mpi call | 9.489946 | 6.163819 | 6.322014 |

**Table 1: Overlap measurements for NAS BT**

| Time in sec | CG-B-4 | CG-B-8 | CG-B-16 |
|---|---|---|---|
| data xfer | 1.431546 | 1.535215 | 1.53522 |
| min ovpd xfer | 0.003482 | 0.013272 | 0.011878 |
| max ovpd xfer | 0.71926 | 0.780887 | 0.779498 |
| user comp | 61.908774 | 19.930214 | 11.485414 |
| mpi call | 11.194951 | 7.026514 | 7.320148 |

**Table 2: Overlap measurements for NAS CG**

| Time in sec | LU-B-4 | LU-B-8 | LU-B-16 |
|---|---|---|---|
| data xfer | 1.594567 | 1.378256 | 1.161946 |
| min ovpd xfer | 0.358967 | 0.311651 | 0.295738 |
| max ovpd xfer | 1.156219 | 1.000722 | 0.876629 |
| user comp | 214.339707 | 108.037748 | 49.033089 |
| mpi call | 12.473109 | 9.74484 | 9.341727 |

**Table 3: Overlap measurements for NAS LU**

With the NAS benchmark applications, each process performs a mix of of eager sends and receives as well as rendezvous sends and receives. The non-overlapped transfer time is dominated by the transfer time for receiving rendezvous messages. Because the invocation of RDMA Read by the receiver is transparent to the sender and the invocation of RDMA Write by the sender is transparent to the receiver, the transfer time for sending rendezvous messages and receiving eager messages account for the gap between the minimum and maximum overlapped transfer times. The minimum overlapped transfer time is attributable to the overlap achieved for sending eager messages.

For the applications, it is clear that varying amounts of overlap of communication with computation is being achieved, but there is a significant amount of non-overlapped communication overhead.

In order to determine the overhead introduced by the monitoring infrastructur, we ran the applications iunder a standard implementation of MVAPICH2 without any instrumentation, to compare the total execution time with that obtained under the instrumented version of MVAPICH2. The overhead numbers for NAS CG appear in table 5. Measurements for other NAS benchmarks may be found in [42]. It is seen that the instrumentation only marginally affects the running time of the benchmarks across varying processor counts.

# 3. HIGH AVAILABILITY MODELS

The research conducted under the MOLAR project focuses on the design and development of core technologies and

| Time in sec | MG-B-4 | MG-B-8 | MG-B-16 |
|---|---|---|---|
| data xfer | 0.252065 | 0.211685 | 0.155991 |
| min ovpd xfer | 0.007279 | 0.016654 | 0.017567 |
| max ovpd xfer | 0.133281 | 0.122449 | 0.095637 |
| user comp | 10.444187 | 4.561646 | 2.444882 |
| mpi call | 0.961956 | 0.746142 | 0.628745 |

**Table 4: Overlap measurements for NAS MG**

| Time in sec | CG-B-4 | CG-B-8 | CG-B-16 |
|---|---|---|---|
| Without instr. | 78.57 | 26.07 | 17.41 |
| With instr. | 79.55 | 25.87 | 18.05 |

**Table 5: Measurement of instrumentation overhead for NAS CG**

proof-of-concept prototypes that improve the overall RAS of HEC systems. As part of this effort, we devised a high availability taxonomy specifically adapted to HEC environments [9]. It takes into account that the purpose of a HEC system is to offer computing cycles for *parallel applications* via different subsystems (head, service and compute nodes) at drastically different scale (a few head and service nodes versus tens of thousands of compute nodes) using various critical system services (job and resource management, data storage and I/O, and messaging). HEC systems are inherently complex and high availability solutions need to address individual deficiencies without introducing new ones. Furthermore, raw performance delivered to scientific applications is an essential quality of service. High availability solutions are required to have a low overhead in a failure-free environment. Considering the delicate balance between the overhead in a failure-free environment and the impact of unmasked failures is essential for the acceptance of any fault tolerance solution.

Head and service nodes running critical system services, such as user login, resource management, job scheduling, data storage and I/O, typically represent *single points of failure* and *single points of control* as they render an entire HEC system inaccessible and unmanageable in case of a failure until repair, causing a significant downtime. Compute nodes running the actual scientific application typically represent *single points of failure* as they interrupt a HEC system in case of a failure. However, they also may run critical system services, such as essential parts of the message passing layer. In the worst case scenario, compute node failures may cause outages similar to head and service node failures.

There are various techniques to implement high availability for critical system services [9, 40]. They include *active/standby* and *active/active*. *Active/standby high availability* [21, 22, 43] follows the fail-over model. Service state is saved regularly to some shared stable storage. Upon failure, a new service is restarted or an idle one takes over with the most recent or even current state. This implies a short interruption of service for the time of the fail-over and may involve a rollback to an old backup. *Asymmetric active/active high availability* [28] further improves the RAS properties of a system. In this model, two or more active services offer the same capabilities at tandem without coordination, while an optional idle service is ready to

take over in case of a failure. This technique allows continuous availability of a service with improved throughput performance. However, it has limited use cases due to the missing coordination between active services. *Symmetric active/active high availability* [12, 45] offers a continuous service provided by two or more active services that provide the same capabilities and maintain a common global state using *distributed control* [11] or *virtual synchrony* [30]. The symmetric active/active model is superior in many areas including throughput, availability, and responsiveness, but is significantly more complex due to need for advanced commit protocols.

High availability for scientific applications running on compute nodes is based on a different approach due to scalability and cost constraints. Reactive fault-tolerance solutions, such as message logging [31] and checkpoint/restart [26], can lessen the impact of occurring failures by effectively offering active/standby support. More advanced reactive methods provide adaptive checkpointing (presented in this paper, see section 4), survivability of the message passing layer, like PVM [18, 39] and FT-MPI [13, 16]), diskless checkpointing [2, 5] and fault tolerant scientific algorithms [1, 7]. Future research in this area, outside the scope of MOLAR, needs to target proactive fault-tolerance, i.e. anticipation of failures and respective preemptive measures.

A more detailed description of our adapted high availability taxonomy can be found in an earlier paper [9].

# 4. RELIABILITY-AWARE JOB AND RESOURCE MANAGEMENT

In this section, we investigate the effect of reliability-aware runtime adaptation on job completion times of large scale parallel applications in HEC environments. Our main goal is to unveil insights in order to enhance runtime fault tolerance and improve system resource utilization. Key attributes in this study are reliability (time to failure) and availability (time to recover). Both play important roles in enabling HEC OS/Rs to make smarter decisions and to deal with faulty situations more efficiently. For example, ideally, we want to place a checkpoint immediately before the failure or optimally place tasks in failure-prone HEC environments. We analyze and discuss the failures and downtime event logs of major production HEC systems, perform runtime availability and reliability analysis, and develop a performance and reliability tradeoff based on the actual runtime mean-time-to-failure (MTTF) and mean-time-to-recover (MTTR).

We analyzed the system logs of a major HEC computing infrastructure [27]. The system log file contains significant system events from years past, collected from four ASC machines, namely White, Frost, Ice, and Snow. We then performed a detailed analysis on these data sets. For the purpose of brevity, we present only the analysis results of White. White, the largest among the aforementioned systems, is a 512-node, 16-way symmetric multiprocessor (SMP) parallel computer. All nodes are of IBM's RS/6000 POWER3 symmetric multiprocessor architecture. Each node is a standalone machine possessing its own memory, operating system (IBM AIX), local disk, and 16 CPUs.

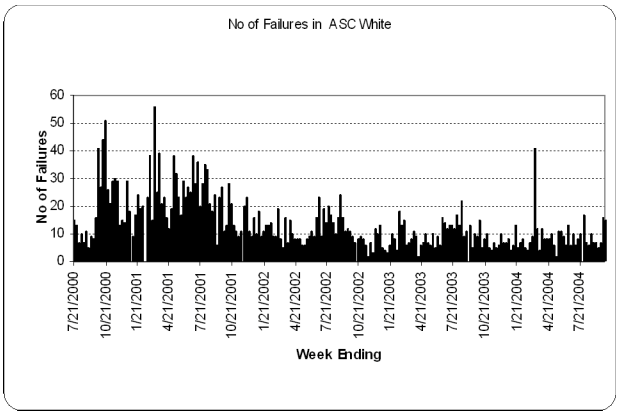



**Figure 1: Number of Failures on ASC White**

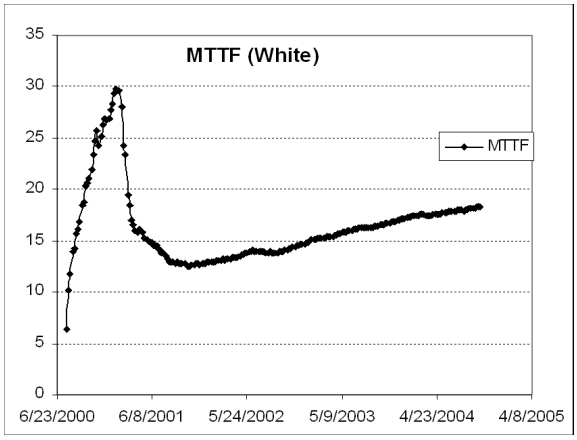
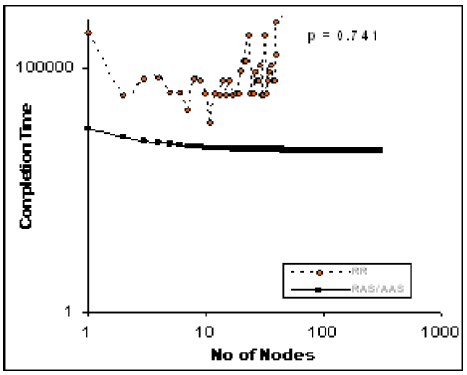


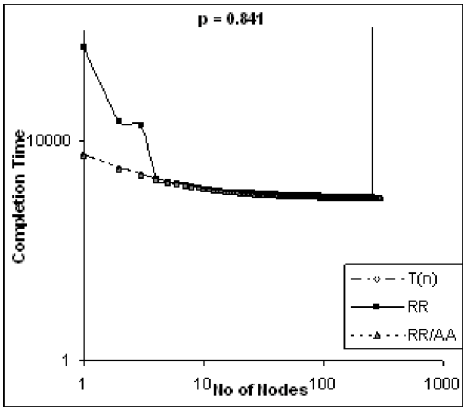

**Figure 2: MTTF on ASC White (in Hours)**

The availability, MTTF and MTTR was analyzed for each node in the system. The MTTF for the a node is equal to (total elapsed time)/(number of failures). Figure 1 shows failure counts at each interval during a four-year span. We assume that the system was utilized with the maximal possible performance. Therefore, the total system reliability MTTF of 512 nodes is 18.28 hrs or the failure rate $\lambda$ of the 512 nodes is 0.054686. Figure 2 presents the ASC white overall system MTTF during the four year period.

Depending on how the HEC resources are utilized, system reliability varies from application to application and execution to execution; it also depends on runtime configurations. In order to effectively deal with reliability issues, runtime mechanisms must be aware of current situations and anticipate any disruption that may occur in the system. We conduct a study on reliability-aware runtime, particularly on a scheduling mechanism. The main objective is to study the benefit of the scheduler that assigns application tasks to a set of nodes while taking into account system reliability in order to beat the odds of failure rate. Let us consider a scenario of parallel application executions under the job queue fault tolerance that supports transparent checkpoint/recovery mechanism. During the execution, there will be a series of checkpoints and perhaps failures until job com-




(a) $p = 0.741$




(b) $p = 0.841$

**Figure 3: Job Completion Time (in Hours)**

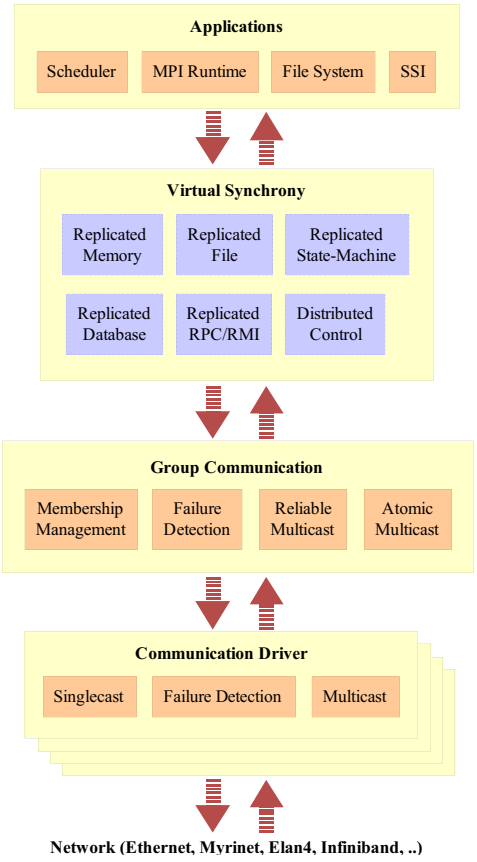



**Figure 4: High Availability Framework**

pletion. We construct an execution model and simulation to study parallel application performance and reliability trade-off. The details can be found in [20].

In our study, we assume that parallel applications executing in an environment similar to ASC White. Thus, the failure model and distributions follow those in the White environment shown in Figures 1 and 2. We present the results of comparing round robin and scheduling algorithms that consider reliability and availability of the nodes. We name these Availability-Aware Scheduling (AAS) and Reliability-Aware Scheduling (RAS) algorithms. We compare the algorithms by varying the number of nodes assigned for the job (k) up to 512 nodes. If any node selected to run the job fails before completion, the job is allocated to the next available k nodes. This is repeated until the job completes successfully. The completion time of the job is the sum of the times the job has spent on the nodes that have failed.

Figures 3(a) and 3(b) show job completion time comparisons among studied scheduling algorithms for parallel applications with different degrees of parallelism ($p = 0.741$ and $p = 0.841$ – a more complete analysis can be found in [20]). Our results indicate that the scheduling algorithms with reliability and availability awareness can improve the job completion time of a parallel program. When we increase the number of nodes to solve a computational problem, the completion time would decrease until a certain threshold, which is the reliability break-even point. Therefore, increasing the number of nodes may not necessarily reduce the completion time. It is therefore imperative to consider availability and reliability as important attributes in scheduling large scale HEC to achieve better performance.

## 5. HIGH AVAILABILITY FRAMEWORK

In order to provide high availability for HEC, we are in the process of developing a flexible, pluggable component framework that allows adaptation to system properties, like network technology and system scale; and application needs, such as programming model and consistency requirements.

The high availability framework (Figure 4) consists of four layers: communication drivers, a group communication system, virtual synchrony interfaces and applications. At the lowest layer, communication drivers provide single- and multicast messaging capability and messaging related failure detection. The group communication layer offers membership management, external failure detection, reliable and atomic multicast. The virtual synchrony layer builds a bridge between the group communication system and applications offering easy-to-use interfaces common to application programmers. The framework itself is component-based, *i.e.*, individual modules within each layer may be interchanged using plug-in technology previously developed in the Har-

ness project [6, 19, 35]. In the following sections, we describe each of the four framework layers in more detail.

## 5.1 Communication Drivers

Today's HEC systems come with a variety network technologies, such as Myrinet, Elan4, Infiniband and Ethernet. The high availability framework is capable of supporting vendor supplied network technologies as well as established standards, such as TCP and UDP, using communication drivers, thus enabling efficient communication. The concept of using communication drivers to adapt specific APIs of different network technologies to a unified communication API in order to make them interchangeable and interoperable is not new. For example, Open MPI [17, 36] uses a component-based framework and encapsulates communication drivers using interchangeable and interoperable components.

We are currently investigating if our framework is able to profit from Open MPI communication driver technology by using the Open MPI framework. This also provides an opportunity for Open MPI to benefit from our high availability framework using active/active high availability for essential Open MPI services. Furthermore, we are also currently considering heterogeneity aspects and high-level protocols. For the moment, communication drivers offer an interface that deals with raw data, only. Future work in this area will also reuse recent research in adaptive, heterogeneous and reconfigurable communication frameworks, such as RMIX [8].

## 5.2 Group Communication Layer

The group communication layer contains all essential protocols and services to run virtual synchronous services for symmetric active/active high availability. It also offers necessary commit protocols for active/hot-standby and asymmetric active/active high availability with multiple standby services using coherent replication. Many (60+) group communication algorithms can be found in literature [4]. Our pluggable component-based high availability framework provides an experimental platform for comparing existing solutions and for developing new ones. Implementations with various replication and control strategies using a common API allow adaptation to system properties and application needs. The modular architecture also enables external contributions based on the common API.

## 5.3 Virtual Synchrony Layer

The supported APIs at the virtual synchrony layer are based on application properties. Deterministic and fully symmetrically replicated applications may use replication interfaces for memory, files, state-machines and databases. Nondeterministic or asymmetrically replicated applications may use more advanced replication interfaces for distributed control and replicated remote procedure calls. These application properties are entirely based on the group communication systems point of view and its limited knowledge about the application.

## 5.4 Applications

There are many, very different, applications for the high availability framework. Typical single points of failure and control involve critical system services on head and service nodes mentioned earlier. Another application area is more deeply involved with the OS itself. For example, single system image (SSI) solutions run one virtual OS across a networked system. Memory page replication is needed for a highly available SSI. Applications on compute nodes include: super-scalable diskless checkpointing [2, 5], scalable MPI recovery and coherent data caching and staging [23].

The framework is implemented as a set of shared and static libraries. Depending on the application area, it may be used within the application by direct linking or via a daemon process by network access. A direct integration into an OS kernel, such as Linux, is not planned.

## 5.5 Prototype Implementation

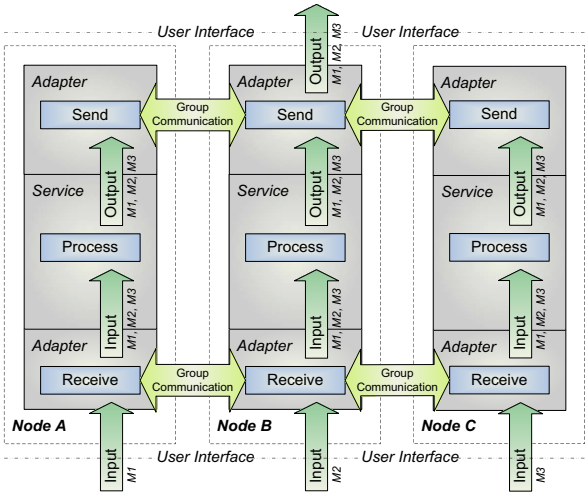
An early prototype has been implemented using the lightweight Harness runtime environment (RTE) [6] as a flexible and pluggable backbone for the described software components. Harness is a pluggable heterogeneous Distributed Virtual Machine (DVM) environment for parallel and distributed scientific computing. Conceptually, the Harness software architecture consists of two major parts: a runtime environment (RTE) and a set of plug-in software modules. The multi-threaded user-space RTE manages the set of dynamically loadable plug-ins. While the RTE provides only basic functions, plug-ins may provide a wide variety of services needed in fault-tolerant parallel and distributed scientific computing, such as messaging, scientific algorithms and resource management. Multiple RTE instances can be aggregated into a DVM.

Previous research in Harness already targeted a group communication system to manage a symmetrically distributed virtual machine environment (DVM) using distributed control [10] as a form of RPC-based virtual synchrony. The Harness distributed control plug-in provides virtual synchrony services to the Harness DVM plug-in, which maintains a symmetrically replicated global state database for high availability. The accomplishments and limitations of Harness and other group communication middleware projects were the basis for the flexible, pluggable and component-based high availability framework.

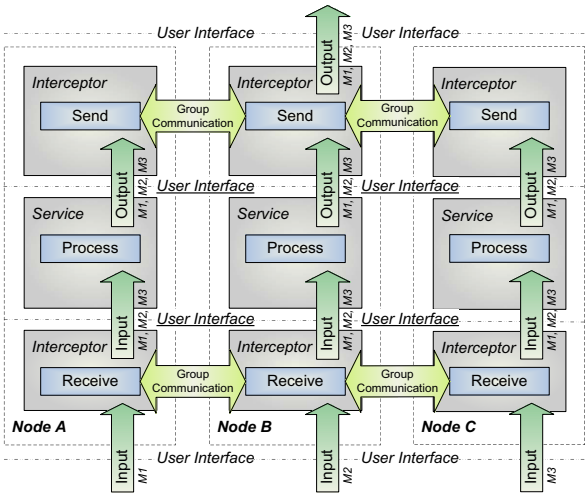
# 6. ACTIVE/ACTIVE REPLICATION

Implementing symmetric active/active high availability 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 [11], 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, command line based HEC services, such as the batch job scheduler, and proprietary networked HEC services, such as data storage and I/O, do not necessarily offer these hooks.

As part of our MOLAR research, we developed two distinct replication methods [12] (Figure 5) for symmetric active/active high availability using virtual synchrony to allow adaptation of existing services to the event-based or request/response programming model either internally by




(a) Internal




(b) External

**Figure 5: Methods for Symmetric Active/Active Replication of Services**

modifying the service itself or externally by wrapping it into a virtually synchronous environment.

## 6.1 Internal Replication

Internal replication (Figure 5(a)) 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. Services may also choose fine-grain synchronization using the group communication system to perform state changes in multiple stages by splitting them into smaller atomic operations. Since state changes are handled in an uninterruptible fashion, splitting them up allows interleaving in order to gain performance and reduce response latency. However, only non-conflicting state changes may be interleaved in order to maintain correctness.

## 6.2 External Replication

External replication (Figure 5(b)) 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 communication 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 is routed through the interceptor group for at most once delivery. External replication implies coarse-grain synchronization. Interleaving state changes is not possible, since the interceptor group forces atomicity for all service interface operations.

## 6.3 Comparison

Internal replication requires modification of code, which may be unsuitable for complex services. The amount of modification necessary may result in a complete redesign. In contrast, external replication forces atomicity at the user-level interface, which may be unsuitable for services with substantially time consuming user interface operations. Another issue emerges when considering *live upgrades*. Upgrading a highly available service while it is running in a symmetric active/active fashion requires removal, upgrade and rejoin of each individual service group member, one at a time. A strict prerequisite is that the new version fully supports the interface and semantics of the old one. Similar to live upgrades, individual symmetric active/active high availability solutions may be reused for different service implementations with the same user interface, for example, different resource management systems supporting the Open PBS [37] interface. Maintaining a consistent interface over a significant period of time is easier using external replication as it is based on the external user interface, while the internal design is not affected and may change with a new version.

Our ongoing work focuses on providing symmetric active/active high availability for the TORQUE [44] resource management system using external replication and for the Parallel Virtual File System (PVFS) [38] metadata server using internal replication. While we have not performed extensive performance benchmarking, early results show that internal replication offers higher performance with interleaved state changes and slightly reduces the response latency introduced by the group communication system. However, interleaving state changes requires substantial knowledge about the internal behavior of a service.

Overall, we recommend the internal replication method if high throughput performance is needed, for example, for file system servers, except where the prospect of extensive code modification or foreseeable major service design changes prohibits it. However, we recommend using external replication if high throughput performance is not a major requirement and the symmetric active/active high availability solution should be reusable for other services that have exactly the same user interface.

# 7. FAULT-TOLERANT MEMBERSHIP FOR MPI COMMUNICATION

Fault-tolerance within the MPI runtime requires that tasks can be dynamically added and removed in response to false if we want to support an environment that can dynamically withstand node failures rather than restarting MPI jobs.

We have developed a scalable approach to reconfigure the communication infrastructure after node failures suitable for MPI runtime systems. A decentralized (peer-to-peer) protocol maintains membership knowledge in the presence of faults. Instead of a performance overhead of seconds for past frameworks, our protocol shows response times in the order of microseconds for reconfiguration over TCP on FastEther. We also verify experimental results against a performance model. The membership service is suitable for deployment in the communication layer of MPI runtime systems, and we are currently pursuing its integration into Open MPI.

Our algorithm provides a new, consistent view (set) of active nodes at very low overhead. The process of establishing a new view is called *tree stabilization* in the following.

Nodes participating in the membership services are internally represented in two data structures: a radix tree and a linear array of nodes. The former provides an efficient representation for collective communication while the latter support point-to-point communication.

The radix tree provides a hierarchical representation that implicitly encodes routing information in the node ID, which reduces the overhead of algorithms that exploit the membership service. The radix encoding of a node ID can be used to determine the routing path of messages from the root to this node or to determine its position in the tree structure. To allow efficient decoding of routing information, the number of children in the radix tree has to be a power of two.

The radix tree is duplicated on each node and kept up-to-date with respect to a global view in a decentralized manner (consistent state with other nodes). At startup, all nodes have the same initial view. When a failure is detected, *e.g.*, by IPMI health monitoring, the root of the tree is informed.

The root node recalculates its tree structure by eliminating the failed node from its list of nodes and updates corresponding links to its children. It then sends a message to its children to indicate that their views need to be updated as well. This failure indicator transitively propagates down the tree followed by acknowledgments up the tree structure. Once the root has received all acknowledgments, a new stable view has been established. The latency between the initial reception of a node failure at the root node and receiving all acknowledgments is called the stabilization time (Ts), a performance metric used subsequently in experiments. Node deletion can be handled in much the same way by eliminating one node and propagating a message to instruct nodes to exclude a failed node as part of the transitive notification process. Simultaneous insertions or deletions are supported by parallel reconfiguration facilitated by the static routing of the radix tree. Root failures are handled by selecting the closest operational node from the linear node list to reduce the coordination overhead.

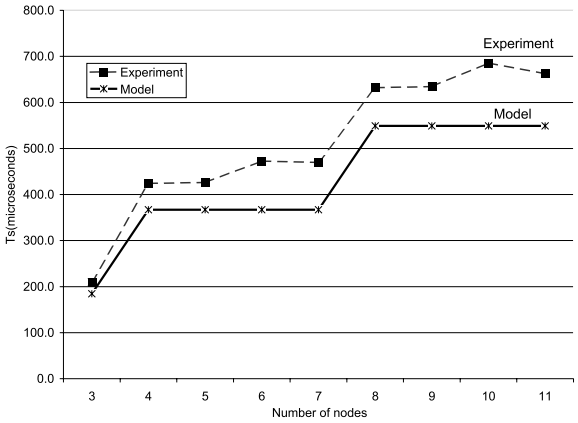



**Figure 6: Stabilization Time (Ts) over TCP**

To measure the *time for stabilization* (Ts), we evaluated our protocol for a binary radix tree using TCP with a FastEthernet switch. We also compared this metric to a model-based on point-to-point latency (L), overhead for communication (Ocm) and computation (Ocp), and the height (H) of the tree [3]:

$$Ts = Ocm + Ocp * H \quad (1)$$

Experiments were conducted under Red Hat 7.3 Linux (kernel version 2.4.18) on a 16 node dual-processor AMD Athlon XP 1800+ machines connected by a full-duplex FastEther switch over TCP/IP. Ocm was measured as 2.3 $\mu$sec.

The results of TCP are shown in Figure 6. The latency for the Ethernet connection was measured as 90 $\mu$sec. The experimental results closely follow the theoretical results resembling a step curve as the tree height and, hence, the number of hops increases. Actual values in experiments were slightly higher than those of the model, which can be attributed to a constant system overhead that the model does not accurately reflect. Overall, the experiments show that the model closely follows the actually observed stabilization overhead, which has a logarithmic complexity in the number of nodes and, hence, is well suited for large-scale machines. We are currently extending our experiments to larger node counts, and preliminary results indicate that our observations still hold in such environments.

# 8. CONCLUSIONS

The MOLAR research we presented in this paper focuses on advancing RAS for HEC in order to address the challenges that OS/R solutions face on the next generation of supercomputers. Our research targets the design and development of core technologies and proof-of-concept prototypes. In this paper, we described our ongoing efforts in performance instrumentation for characterizing the degree of computation-communication overlap in message-passing systems, in devising an appropriate high availability taxonomy for high-end computing, in developing a reliability-aware job and resource management solution, in developing a flexible, adaptable, component-based high availability framework, in equipping existing critical system services