On Programming Models for Service-Level High Availability*

C. Engelmann[1,2], S. L. Scott[1], C. Leangsuksun[3], X. He[4]
[1]*Computer Science and Mathematics Division*
Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
[2]*Department of Computer Science*
The University of Reading, Reading, RG6 6AH, UK
[3]*Computer Science Department*
Louisiana Tech University, Ruston, LA 71272, USA
[4]*Department of Electrical and Computer Engineering*
Tennessee Technological University, Cookeville, TN 38505, USA
engelmannc@ornl.gov, scottsl@ornl.gov, box@latech.edu, hexb@tntech.edu

Abstract

This paper provides an overview of existing programming models for service-level high availability and investigates their differences, similarities, advantages, and disadvantages. Its goal is to help to improve reuse of code and to allow adaptation to quality of service requirements. It further aims at encouraging a discussion about these programming models and their provided quality of service, such as availability, performance, serviceability, usability and applicability. Within this context, the presented research focuses on providing high availability for services running on head and service nodes of high-performance computing systems. The proposed conceptual service model and the discussed service-level high availability programming models are applicable to many parallel and distributed computing scenarios as a networked system's availability can invariably be improved by increasing the availability of its most critical services.

*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 the 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.

1. Introduction

Today's scientific research community relies on high-performance computing (HPC) to understand the complex nature of open research questions and to drive the race for scientific discovery in various disciplines, such as in nuclear astrophysics, climate dynamics, fusion energy, nanotechnology, and human genomics. Scientific HPC applications, like the Terascale Supernova Initiative [29] or the Community Climate System Model (CCSM) [3], allow to model and simulate complex experimental scenarios without the need or capability to perform real-world experiments.

HPC systems in use today can be divided into two distinct use cases, capacity and capability computing. In capacity computing, scientific HPC applications run in the order of a few seconds to several hours, typically on small-to-mid scale systems. The main objective for operating capacity HPC systems is to satisfy a large user base by guaranteeing a certain computational job submission-to-start response time and overall computational job throughput. In contrast, capability computing focuses on a small user base running scientific HPC applications in the order of a few hours to several months on the largest scale systems available. Operating requirements for capability HPC systems include guaranteeing full access to the entire machine, one user at a time for a significant amount of time.

In both use cases, demand for continuous availability of HPC systems has risen dramatically as capacity system users experience HPC as an on-demand service with certain quality of service guarantees, and capability users require exclusive access to large HPC systems

over a significant amount of time with certain reliability guarantees. HPC systems must be able to run in the event of failures in such a manner that their capacity or capability is not severely degraded.

However, both variants partially require different approaches for providing quality of service guarantees as use case scenario, system scale and financial cost determine the feasibility of individual solutions. For example, a capacity computing system may be equipped with additional spare compute nodes to tolerate a certain predictable percentage of compute node failures [30]. The spare compute nodes are able to quickly replace failed compute nodes using a checkpoint/restart system, like the Berkeley Lab Checkpoint/Restart (BLCR) [1] solution, for the recovery of currently running scientific HPC applications. In contrast, the notion of capability computing implies the use of all available compute nodes. Using spare compute nodes in a capability HPC system just degrades the capability upfront, even in the failure-free case.

Nevertheless, capacity and capability computing systems have also similarities in their overall system architectures, such that some solutions apply to both use cases. For example, a single head node is typically managing a HPC system, while optional service nodes may offload certain head node services to provide better performance and scalability. Furthermore, since the head node is the gateway for users to submit scientific HPC applications as computational jobs, multiple gateway nodes may exists as virtual head nodes to provide dedicated user group access points and to apply certain user group access policies.

A generic architectural abstraction for HPC systems can be defined, such that a group of closely coupled compute nodes is connected to a set of service nodes. The head node of a HPC system is as a special service node. The system of users, service nodes and compute nodes is interdependent, *i.e.*, users depend on service nodes and compute nodes, and compute nodes depend on service nodes. Providing high availability for service nodes invariably increases the overall system availability, especially when removing single points of failure and control from the system.

Within this context, our research focuses on providing high availability for service nodes using the service model and service-level replication as a base concept. The overarching goal of our long-term research effort is to provide high-level reliability, availability and serviceability (RAS) for HPC by combining advanced high availability (HA) with HPC technology, thus allowing scientific HPC applications to better take advantage of the provided capacity or capability.

The goal of this paper is to provide an overview of programming models for service-level high availability in order to investigate their differences, similarities, advantages, and disadvantages. Furthermore, this paper aims at encouraging a discussion about these programming models and their provided quality of service, such as availability, performance, serviceability, usability and applicability.

While our focus is on capability HPC systems, like the 100 Tflop/s Cray XT4 system [35] at Oak Ridge National Laboratory [23], in particular, the presented service model and service-level high availability programming models are applicable to many parallel and distributed computing scenarios as a networked system‘s availability can invariably be improved by increasing the availability of its most critical services.

This paper is structured as follows. First we discuss previous work on providing high availability for services in HPC systems, followed by a brief overview of previously identified research and development challenges in service-level high availability for HPC systems. We continue with a definition of the conceptual service model and a description of various service-level high availability programming models and their properties. This paper concludes with a discussion about the presented work and future research and development directions in this field.

2. Previous Work

While there is a substantial amount of previous work on fault tolerance and high availability for single services as well as for services in networked systems, only a few solutions for providing high availability for services in HPC systems existed when we started our research a few years ago. Even today, high availability solutions for HPC system services are rare.

The PBS Pro job and resource management for the Cray XT3 [24] supports high availability using hot standby redundancy 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 a mean time to recover (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.

The Simple Linux Utility for Resource Management (SLURM) [26, 36] job and resource manager as well as the metadata servers of the Parallel Virtual File System (PVFS) 2 [25] and of the Lustre [19] cluster file system employ an active/standby solution using a shared storage device, which is a common technique for providing service-level high availability using a heart-

beat mechanism [14]. 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. As with most shared storage solutions, correctness and quality of service are not guaranteed due to the lack of strong commit protocols.

High Availability Open Source Cluster Application Resources (HA-OSCAR) [12, 13, 18] is a high availability framework for the OpenPBS [22] and TORQUE [31] job and resource management systems developed by our team at Louisiana Tech University. It employs a warm standby redundancy strategy. State is replicated to a standby head node upon modification. The standby monitors the health of the active node using a heartbeat mechanism and initiates the fail-over based on the current state. However, OpenPBS/TORQUE does temporarily loose control of the system. All previously running jobs are automatically restarted. HA-OSCAR integrates with the compute node checkpoint/restart layer for LAM/MPI [15], BLCR [1], improving its MTTR to 3-5 seconds for detection and fail-over plus the time to catch up based on the last checkpoint.

As part of the HA-OSCAR research, an asymmetric active/active prototype [17] has been developed that offers high availability in a high-throughput computing scenario. Two different job and resource management services, OpenPBS and the Sun Grid Engine (SGE) [28], 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. OpenPBS and SGE do loose control of the system during fail-over, requiring a restart of currently running jobs controlled by the failed node. Only one failure is completely masked at a time due to the $2 + 1$ configuration. However, the system is still operable during a fail-over due to the second active head node. A second failure results in a degraded mode with one head node serving the entire system.

JOSHUA [33, 34] developed by our team at Oak Ridge National Laboratory 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 [9] based on the PBS service interface providing high availability without any interruption of service and without any loss of state. JOSHUA relies on the Transis [5, 32] group communication system for reliable, totally ordered message delivery. The introduced communication overhead in a 3-way active/active head node system with 99.9997% service uptime, a job submission latency of 300 milliseconds, and a throughput of 3 job submissions per second is in an acceptable range.

Similar to the JOSHUA solution, our team at Tennessee Technological University recently developed an active/active solution for the PVFS metadata server (MDS) using the Transis group communication system with an improved total message ordering protocol for lower latency and throughput overheads, and utilizing load balancing for MDS read requests.

Further information about the overall background of high availability for HPC systems, including a more detailed problem description, conceptual models, a review of existing solutions and other related work, and a description of respective replication methods can be found in our earlier publications [10, 8, 6, 7].

Based on an earlier evaluation of our accomplishments [11], we identified several existing limitations, which need to be dealt with for a production-type deployment of our developed technologies. The most pressing issues are stability, performance, and interaction with compute node fault tolerance mechanisms. The most pressing issue for extending the developed technologies to other critical system services is portability and ease-of-use.

Other related work includes the Object Group Pattern [20], which offers programming model support for replicated objects using virtual synchrony provided by a group communication system. In this design pattern, objects are constructed as state machines and replicated using totally ordered and reliably multicast state transitions. The Object Group Pattern also provides the necessary hooks for copying object state, which is needed for joining group members.

Orbix+Isis and Electra are follow-on research projects [16] that focus on extending high availability support to CORBA using object request brokers (ORBs) on top of virtual synchrony toolkits.

Lastly, there is a plethora of past research on group communication algorithms [2, 4] focusing on providing quality of service guarantees for networked communication in distributed systems with failures.

3. Service Model

An earlier evaluation of our accomplishments and their limitations [11] revealed one major problem, which is hindering necessary improvements of developed prototypes and hampering extending the developed technologies to other services. All service-level high availability solutions need a customized active/hot-

standby, asymmetric active/active, or symmetric active/active environment, resulting in insufficient reuse of code. This results not only in rather extensive development efforts for each new high availability solution, but also makes their correctness validation an unnecessary repetitive task. Furthermore, replication techniques, such as active/active and active/standby, cannot be seamlessly interchanged for a specific service in order to find the most appropriate solution based on quality of service requirements.

In order to alleviate and eventually eliminate this problem, we propose a more generic approach toward service-level high availability. In a first step we define a conceptual service model. Various service-level high availability programming models and their properties are described based on this model, allowing us to define programming interfaces and operating requirements for specific replication techniques. In a second step, a replication framework, as proposed earlier in [7], may be able to provide interchangeable service-level high availability programming interfaces requiring only minor development efforts for adding high availability support for services, and allowing seamless adaptation to quality of service requirements.

The proposed more generic approach toward service-level high availability defines any targeted service as a communicating process, which interacts with other local or remote services and/or with users via an input/output interface, such as network connection(s), command line interface(es), and/or other forms of interprocess and user communication. Interaction is performed using input messages, such as network messages, command line executions, etc., which may trigger output messages to the interacting service or user in a request/response fashion, or to other services or users in a trigger/forward fashion.

While a stateless service does not maintain internal state and reacts to input with a predefined output independently of any previous input, stateful services do maintain internal state and change it accordingly to input messages. Stateful services perform state transitions and produce output based on a deterministic state machine. However, non-deterministic behavior may be introduced by non-deterministic input, such as by a system timer sending input messages signaling a specific timeout or time. If the service internally relies on such a non-deterministic component invisible to the outside world, it is considered non-deterministic.

Non-deterministic service behavior has serious consequences for service-level replication techniques as replay capability cannot be guaranteed. Stateless services on the other hand do not require service-level state replication. However, they do require appropriate mechanisms for fault tolerant input/output message routing to/from replacement services.

As most services in HPC systems are stateful and deterministic, like for example the parallel file system metadata server, we will explore their conceptual service model and service-level high availability programming models. Non-determinism, such as displayed by a batch job scheduling system, may be avoided by configuring the service to behave deterministic, for example by using a deterministic batch job scheduling policy.

We define the state of service p at step t as S_p^t, and the initial state of service p as S_p^0. A service state transition from S_p^{t-1} to S_p^t is triggered by request message r_p^t sent to and processed by service p at step $t-1$, such that request message r_p^1 triggers the state transition from S_p^0 to S_p^1. Request messages are processed in order at the respective service state, such as that service p receives and processes the request messages r_p^1, r_p^2, r_p^3, ... There is a linear history of state transitions S_p^0, S_p^1, S_p^2, ... in direct context to a linear history of request messages r_p^1, r_p^2, r_p^3, ...

Service state remains unmodified when receiving and processing the x-th query message $q_p^{t,x}$ by service p at step t. Multiple different query messages may be processed at the same service state out of order, such as that service p processes the query messages $r_p^{t,3}$, $r_p^{t,1}$, $r_p^{t,2}$, ... previously received as $r_p^{t,1}$, $r_p^{t,2}$, $r_p^{t,3}$, ...

Each processed request message r_p^t may trigger any number of y output messages ..., $o_p^{t,y-1}$, $o_p^{t,y}$ related to the specific state transition S_p^{t-1} to S_p^t, while each processed query message $q_p^{t,x}$ may trigger any number of y output messages ..., $o_p^{t,x,y-1}$, $o_p^{t,x,y}$ related to the specific query message.

A deterministic service always has replay capability, *i.e.*, different instances of a service have the same state if they have the same linear history of request messages. Furthermore, not only the current state, but also past service states may be reproduced using the respective linear history of request messages.

A service may provide an interface to atomically obtain a snapshot of the current state using a query message $q_p^{t,x}$ and its output message $o_p^{t,x,1}$, or to atomically overwrite its current state using a request message r_p^t with an optional output message $o_p^{t,1}$ as confirmation. Both interface functions are to be used for service state replication only. Overwriting the current state of a service constitutes a break in the linear history of state transitions of the original service instance by assuming a different service instance, and therefore is not considered a state transition by itself.

The failure mode of a service is fail-stop, *i.e.*, the

service, its node, or its communication links, fail by simply stopping. Failure detection mechanisms may be deployed to assure fail-stop behavior in certain cases, such as for incomplete or garbled messages.

4. Active/Standby Replication

In the active/standby replication model for service-level high availability, at least one additional standby service B is monitoring the primary service A for a fail-stop event and assumes the role of the failed active service when detected. The standby service B should preferably reside on a different node, while fencing the node of service A after a detected failure to enforce the fail-stop model (STONITH concept) [27].

Service-level active/standby replication is based on assuming the same initial states for the primary service A and the standby service B, *i.e.*, $S_A^0 = S_B^0$, and on replicating the service state from the primary service A to the standby service B by guaranteeing a linear history of state transitions. This can be performed in two distinctive ways, as active/warm-standby and active/hot-standby replication.

In the warm-standby model, service state is replicated regularly from the primary service A to the standby service B in a consistent fashion, *i.e.*, the standby service B assumes the state of the primary service A once it has been transferred and validated in its entirety. Service state replication may be performed by the primary service A using an internal trigger mechanism, such as a timer, to atomically overwrite state of the standby service B. It may also be performed in the reverse form by the standby service B using a similar internal trigger mechanism to atomically obtain a snapshot of the state of the primary service A. The latter case already provides a failure detection mechanism using a timeout for the response from the primary service A.

A failure of the primary service A triggers the fail-over procedure to the standby service B, which becomes the new primary service A' based on the last replicated state. Since the warm-standby model does assure a linear history of state transitions only up to the last replicated service state, all dependent services and users need to be notified that a service state rollback may have been performed. However, the new primary service A' is unable to verify by itself if a rollback has occurred.

In the hot-standby model, service state is replicated on change from the primary service A to the standby service B in a consistent fashion, *i.e.*, using a commit protocol, in order to provide a fail-over capability without state loss. Service state replication is performed by the primary service A when processing request messages. A previously received request message r_A^t is forwarded by the primary service A to the standby service B as request message r_B^t. The standby service B replies with an output message $o_B^{t,1}$ as an acknowledgment and performs the state transition S_B^{t-1} to S_B^t without generating any output. The primary service A performs the state transition S_A^{t-1} to S_A^t and produces output accordingly after receiving the acknowledgment $o_B^{t,1}$ from the standby service B or after receiving a notification of a failure of the standby service B.

A failure of the primary service A triggers the fail-over procedure to the standby service B, which becomes the new primary service A' based on the current state. In contrast to the warm-standby model, the hot-standby model does guarantee a linear history of state chance transitions up to the current state. However, the primary service A may have failed before sending all output messages $\ldots$, $o_A^{t,y-1}$, $o_A^{t,y}$ for an already processed request message r_A^t. Furthermore, the primary service A may have failed before sending all output messages $\ldots$, $o_A^{t,x-1,y-1}$, $o_A^{t,x-1,y}$, $o_A^{t,x,y-1}$, $o_A^{t,x,y}$ for previously received query messages $\ldots$, $q_A^{t,x-1}$, $q_A^{t,x}$. All dependent services and users need to be notified that a fail-over has occurred. The new primary service A' resends all output messages $\ldots$, $o_{A'}^{t,y-1}$, $o_{A'}^{t,y}$ related to the previously processed request message $r_{A'}^t$, while all dependent services and users ignore duplicated output messages. Unanswered query messages $\ldots$, $q_A^{t,x-1}$, $q_A^{t,x}$ are reissued to the new primary service A' as query messages $\ldots$, $q_{A'}^{t,x-1}$, $q_{A'}^{t,x}$ by dependent services and users.

The active/standby model always requires to notify dependent services and users about the fail-over. However, in a transparent active/hot-standby model, as exemplified by the earlier mentioned PBS Pro job and resource management for the Cray XT3 [24], the network interconnect hardware itself performs in-order replication of all input messages and at most once delivery of output messages from the primary node, similar to the symmetric active/active model.

Active/standby replication also always implies a certain interruption of service until a failure has been detected and a fail-over has been performed. The MTTR of the active/warm-standby model depends on the time to detect a failure, the time needed for fail-over, and the time needed to catch up based on the last replicated service state. The MTTR of the active/hot-standby model only depends on the time to detect a failure and the time needed for fail-over.

The only performance impact of the active/warm-standby model is the need for an atomic service state snapshot, which briefly interrupts the primary service. The active/hot-standby model adds communication latency to every request message, as forwarding to the

secondary node and waiting for the respective acknowledgment is needed. This communication latency overhead can be expressed as $C_L = 2l_{A,B}$, where $l_{A,B}$ is the communication latency between the primary service A and the standby service B. An additional communication latency and throughput overhead may be introduced if the primary service A and the standby service B are not connected by a dedicated communication link equivalent to the communication link(s) of the primary service A to all dependent services and users.

Using multiple standby services B, C, ... may provide higher availability as more redundancy is provided. However, consistent service state replication to the standby services $B, C, \ldots$ requires fault tolerant multicast capability, *i.e.*, service group membership management. Furthermore, a priority-based fail-over policy, *e.g.*, A to B to C to ..., is needed as well.

5. Asymmetric Active/Active Replication

In the asymmetric active/active replication model for service-level high availability, two or more active services $A, B, \ldots$ provide essentially the same capability at tandem without coordination, while optional standby services $\alpha, \beta, \ldots$ in an $n+1$ or $n+m$ configuration may replace failing active services. There is no synchronization or service state replication between the active services $A, B, \ldots$.

Service state replication is only performed from the active services $A, B, \ldots$ to the optional standby services $\alpha, \beta, \ldots$ in an active/standby fashion as previously explained. The only additional requirement is a priority-based fail-over policy, *e.g.*, $A >> B >> \ldots$, if there are more active services n than standby services m, and a load balancing policy for using the active services $A, B, \ldots$ at tandem.

Load balancing of request and query messages needs to be performed at the granularity of user/service groups, as there is no coordination between the active services $A, B, \ldots$. Individual active services may be assigned to specific user/service groups in a static load balancing scenario. A more dynamic solution is based on sessions, where each session is a time segment of interaction between user/service groups and specific active services. Sessions may be assigned to active services at random, using specific networking hardware, or using a separate service. However, introducing a separate service for session scheduling adds an additional dependency to the system, which requires an additional redundancy strategy.

Similar to the active/standby model, the asymmetric active/active replication model requires notification of dependent services and users about a fail-over and about an unsuccessful priority-based fail-over.

It also always implies a certain interruption of a specific active service until a failure has been detected and a fail-over has been performed. The MTTR for a a specific active service depends on the active/standby replication strategy. However, other active services are available during a fail-over, which interact with their specific user/service groups and sessions and respond to new user/service groups and sessions.

The performance of single active services in the asymmetric active/active replication model is equivalent to the active/standby case. However, the overall service provided by the active service group A, B, ... allows for a higher throughput performance and respectively for a lower respond latency due to the availability of more resources and load balancing.

6. Symmetric Active/Active Replication

In the symmetric active/active replication model for service-level high availability, two or more active services $A, B, \ldots$ offer the same capabilities and maintain a common global service state.

Service-level symmetric active/active replication is based on assuming the same initial states for all active services A, B, ..., *i.e.*, $S_A^0 = S_B^0 = \ldots$, and on replicating the service state by guaranteeing a linear history of state transitions using virtual synchrony [21]. Service state replication among the active services A, B, ... is performed by totally ordering all request messages r_A^t, r_B^t, ... and reliably delivering them to all active services $A, B, \ldots$. A process group communication system is used to perform total message order, reliable message delivery, and service group membership management. Furthermore, consistent output messages $o_A^{t,1}$, $o_B^{t,1}$, ... related to the specific state transitions S_A^{t-1} to S_A^t, S_B^{t-1} to S_B^t, ... produced by all active services A, B, ... is unified either by simply ignoring duplicated messages or by using the group communication system for a distributed mutual exclusion. The latter is required if duplicated messages cannot be simply ignored by dependent services and/or users.

If needed, the distributed mutual exclusion is performed by adding a local mutual exclusion variable and its lock and unlock functions to the active services A, B, ... However, the lock has to be acquired and released by routing respective multicast request messages $r_{A,B,\ldots}^t$ through the group communication system for total ordering. Access to the critical section protected by this distributed mutual exclusion is given to the active service sending the first locking request, which in turn produces the output accordingly. Dependent services and users are required to acknowledge the receiving of

output messages to all active services A, B, ... for internal bookkeeping using a reliable multicast. In case of a failure, the membership management notifies everyone about the orphaned lock and releases it. The lock is reacquired until all output is produced accordingly to the state transition. This implies a high overhead lock-step mechanism for at most once output delivery. It should be avoided whenever possible.

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. Forcibly removed services or failed services are simply deleted from the service group membership without any additional reconfiguration. Multiple simultaneous removals or failures are handled as sequential ones. New services join the service group by atomically overwriting their service state with the service state snapshot from one active service group member, *e. g.*, S_A^t, before receiving following request messages, *e. g.*, r_A^{t+1}, r_A^{t+2}, ...

Query messages $q_A^{t,x}$, $q_B^{t,x}$, ... may be send directly to respective active services through the group communication system to assure total order with conflicting request messages. Related output messages ..., $o_A^{t,x,y-1}$, $o_A^{t,x,y}$, $o_B^{t,x,y-1}$, $o_B^{t,x,y}$ are sent directly to the dependent service or user. In case of a failure, unanswered query messages, *e. g.*, ..., $q_A^{t,x-1}$, $q_A^{t,x}$, are reissued by dependent services and users to a different active service, *e. g.*, B, in the service group as query messages, *e. g.*, ..., $q_B^{t,x-1}$, $q_B^{t,x}$.

The symmetric active/active model also always requires to notify dependent services and users about failures in order to reconfigure their access to the service group through the group communication system and to reissue outstanding queries.

There is no interruption of service and no loss of state as long as one active service is alive, since active services run in virtual synchrony without the need of extensive reconfiguration. Furthermore, the MTTR is practically 0. However, the group communication system introduces a communication latency and throughput overhead, which increases with the number of active head nodes. This overhead directly depends on the used group communication protocol.

7. Discussion

All presented service-level high availability programming models show certain interface and behavior similarities. They all require to notify dependent services and users about failures. Transparent masking of this requirement may be provided by an underlying adaptable framework, which keeps track of active services, their current high availability programming model, fail-over and rollback scenarios, message duplication, and unanswered query messages.

The requirements for service interfaces in these service-level high availability programming models are also similar. A service must have an interface to atomically obtain a snapshot of its current state and to atomically overwrite its current state. Furthermore, for the active/standby service-level high availability programming model a service must either provide the described special standby mode for acknowledging request messages and muting output, or an underlying adaptable framework needs to emulate this capability.

The internal algorithms of the active/hot-standby and the active/active service-level high availability programming models are equivalent for reliably delivering request messages in total order to all active services. In fact, the active/hot-standby service-level high availability programming model uses a centralized group communication commit protocol for a maximum of two members with fail-over capability.

Based on the presented conceptual service model and service-level high availability programming models, active/warm-standby provides the lowest runtime overhead in a failure free environment and the highest recovery impact in case of a failure. Conversely, active/active provides the lowest recovery impact and the highest runtime overhead.

Future research and development efforts need to focus on a unified service interface and service-level high availability programming interface that allows reuse of code and seamless adaptation to quality of service requirements while supporting different high availability programming models.

References

[1] Berkeley Lab Checkpoint/Restart (BLCR) project at Lawrence Berkeley National Laboratory, Berkeley, CA, USA. Available at http://ftg.lbl.gov/checkpoint.

[2] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, 2001.

[3] Community Climate System Model (CCSM) at the National Center for Atmospheric Research, Boulder, CO, USA. Available at http://www.ccsm.ucar.edu.

[4] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[5] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[6] C. Engelmann and S. L. Scott. Concepts for high availability in scientific high-end computing. In *Proceed-*

ings of High Availability and Performance Workshop (HAPCW) 2005, Santa Fe, NM, USA, Oct. 11, 2005.

[7] C. Engelmann and S. L. Scott. High availability for ultra-scale high-end scientific computing. In *Proceedings of 2nd 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.

[8] 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)*, 40(2):63–72, 2006.

[9] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He. Active/active replication for highly available HPC system services. In *Proceedings of 1st International Conference on Availability, Reliability and Security (ARES) 2006*, pages 639–645, Vienna, Austria, Apr. 20-22, 2006.

[10] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He. Symmetric active/active high availability for high-performance computing system services. *Journal of Computers (JCP)*, 1(7), 2006. To appear.

[11] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He. Towards high availability for high-performance computing system services: Accomplishments and limitations. In *Proceedings of High Availability and Performance Workshop (HAPCW) 2006*, Santa Fe, NM, USA, Oct. 17, 2006.

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

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

[14] Heartbeat. Available at http://www.linux-ha.org/HeartbeatProgram.

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

[16] S. Landis and S. Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[17] 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 2nd 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.

[18] 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.

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

[20] S. Maffeis. The object group design pattern. *Proceedings of 2nd USENIX Conference on Object-Oriented Technologies (COOTS) 1996*, June 17-21, 1996.

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

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

[23] ORNL Jaguar: Cray XT4 Computing Platform at National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, USA. http://info.nccs.gov/resources/jaguar.

[24] PBS Pro for the Cray XT3 Computing Platform at Altair Engineering, Troy, MI, USA. Available at http://www.altair.com/pdf/PBSPro_Cray.pdf.

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

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

[27] STONITH (node fencing) concept. Explained at http://www.linux-ha.org/STONITH.

[28] Sun Grid Engine (SGE) at Sun Microsystems, Inc, Santa Clara, CA, USA. Available http://gridengine.sunsource.net.

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

[30] A. Tikotekar, C. Leangsuksun, and S. L. Scott. On the survivability of standard mpi applications. In *Proceedings of the 7th LCI International Conference on Linux Clusters: The HPC Revolution 2006*, Norman, OK, USA, May 1-4, 2006.

[31] TORQUE Resource Manager at Cluster Resources, Inc., Spanish Fork, UT, USA. Available at http://www.clusterresources.com/torque.

[32] Transis Project at Hebrew University of Jerusalem, Israel. Available at http://www.cs.huji.ac.il/labs/transis.

[33] K. Uhlemann. High availability for high-end scientific computing. Master's thesis, Department of Computer Science, University of Reading, UK, Mar. 2006.

[34] 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.

[35] XT4 Computing Platform at Cray, Seattle, WA, USA. Available at http://www.cray.com/products/xt4.

[36] 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*, volume 2862, pages 44–60, Seattle, WA, USA, June 24, 2003.