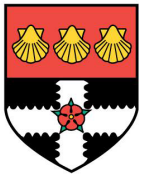
THE UNIVERSITY OF READING



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

This thesis is submitted for the degree of
Doctor of Philosophy

School of Systems Engineering

Christian Engelmann
December 2008

Preface

I originally came to Oak Ridge National Laboratory (ORNL), USA, in 2000 to perform research and development for my Master of Science (MSc) thesis in Computer Science at the University of Reading, UK. The thesis subject "Distributed Peer-to-Peer Control for Harness" focused on providing high availability in a distributed system using process group communication semantics. After I graduated, I returned to ORNL in 2001 and continued working on several aspects of high availability in distributed systems, such as the Harness Distributed Virtual Machine (DVM) environment.

In 2004, I transitioned to a more permanent staff member position within ORNL, which motivated me to engage in further extending my academic background. Fortunately, my former university offered a part-time Doctor of Philosophy (PhD) degree program for working-away students. After reconnecting with my former MSc thesis advisor, Prof. Vassil N. Alexandrov, I formally started studying remotely at the University of Reading, while performing my research for this PhD thesis at ORNL.

As part of this partnership between ORNL and the University of Reading, I not only regularly visited the University of Reading, but I also got involved in co-advising MSc students performing their thesis research and development at ORNL. As an institutional co-advisor, I placed various development efforts, some of which are part of this PhD thesis research, in the hands of several very capable students. I also was able to get a PhD student from Tennessee Technological University, USA, to commit some of his capabilities to a development effort. Their contributions are appropriately acknowledged in the respective prototype sections of this thesis.

The research presented in this PhD thesis was primarily motivated by a continued interest in extending my prior MSc thesis work to providing high availability in high-performance computing (HPC) environments. While this work focuses on HPC system head and service nodes and the state-machine replication approach, some of my other research that was conducted in parallel and is not mentioned in this thesis targets advanced fault tolerance solutions for HPC system compute nodes.

It is my believe that this thesis is a significant step in understanding high availability aspects in the context of HPC environments and in providing practical state-machine replication solutions for service high availability.

Abstract

In order to address anticipated high failure rates, reliability, availability and serviceability have become an urgent priority for next-generation high-performance computing (HPC) systems. This thesis aims to pave the way for highly available HPC systems by focusing on their most critical components and by reinforcing them with appropriate high availability solutions. Service components, such as head and service nodes, are the "Achilles heel" of a HPC system. A failure typically results in a complete system-wide outage. This thesis targets efficient software state replication mechanisms for service component redundancy to achieve high availability as well as high performance.

Its methodology relies on defining a modern theoretical foundation for providing service-level high availability, identifying availability deficiencies of HPC systems, and comparing various service-level high availability methods. This thesis showcases several developed proof-of-concept prototypes providing high availability for services running on HPC head and service nodes using the symmetric active/active replication method, *i.e.*, state-machine replication, to complement prior work in this area using active/standby and asymmetric active/active configurations.

Presented contributions include a generic taxonomy for service high availability, an insight into availability deficiencies of HPC systems, and a unified definition of service-level high availability methods. Further contributions encompass a fully functional symmetric active/active high availability prototype for a HPC job and resource management service that does not require modification of service, a fully functional symmetric active/active high availability prototype for a HPC parallel file system metadata service that offers high performance, and two preliminary prototypes for a transparent symmetric active/active replication software framework for client-service and dependent service scenarios that hide the replication infrastructure from clients and services.

Assuming a mean-time to failure of 5,000 hours for a head or service node, all presented prototypes improve service availability from 99.285% to 99.995% in a two-node system, and to 99.99996% with three nodes.

Acknowledgements

I would like to thank all those people who made this thesis possible and an enjoyable experience for me. This dissertation would have been impossible without their encouragement and support. Specifically, I would like to express my gratitude to Stephen L. Scott and Al Geist from Oak Ridge National Laboratory (ORNL) and to my advisor Prof. Vassil N. Alexandrov from the University of Reading for their continued professional and personal advice, encouragement, and support.

I would also like to thank my students, Kai Uhlemann (MSc, University of Reading, 2006), Matthias Weber (MSc, University of Reading, 2007), and Li Ou (PhD, Tennessee Technological University, 2007), for their contributions in the form of developed prototypes that offered practical proofs of my concepts, but also demonstrated their limitations. Their prototyping work performed under my guidance was instrumental in the incremental progress of my PhD thesis research.

Further, I would like to thank all my friends for their continued encouragement and support. Especially, I would like to thank my family in Germany, which has supported me over all those years I have been away from my home country.

Finally, this thesis research was sponsored by the Office of Advanced Scientific Computing Research of the U.S. Department of Energy as part of the Forum to Address Scalable Technology for Runtime and Operating Systems (FAST-OS). The work was performed at ORNL, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725. The work by Li Ou from Tennessee Tech University was additionally sponsored by the Laboratory Directed Research and Development Program of ORNL.

Declaration

I confirm that this is my own work and the use of all material from other sources has been properly and fully acknowledged. The described developed software has been submitted separately in form of a CD-ROM.

Christian Engelmann

Publications

I have co-authored 2 journal, 6 conference, and 7 workshop publications, co-advised 2 Master of Science (MSc) theses, and co-supervised 1 internship for another Doctor of Philosophy (PhD) thesis as part of the 4-year research and development effort conducted in conjunction with this PhD thesis. At the time of submission of this PhD thesis, 1 co-authored journal publication is under review.

Journal Publications

[1] Christian Engelmann, Stephen L. Scott, Chokchai Leangsuksun, and Xubin He. Symmetric active/active high availability for high-performance computing system services. *Journal of Computers (JCP)*, 1(8):43–54, 2006. Academy Publisher, Oulu, Finland. ISSN 1796-203X. URL http://www.csm.ornl.gov/~engelman/publications/engelmann06symmetric.pdf.

[2] Christian Engelmann, Stephen L. Scott, David E. Bernholdt, Narasimha R. Gottumukkala, Chokchai Leangsuksun, Jyothish Varma, Chao Wang, Frank Mueller, Aniruddha G. Shet, and Ponnuswamy Sadayappan. MOLAR: Adaptive runtime support for high-end computing operating and runtime systems. *ACM SIGOPS Operating Systems Review (OSR)*, 40(2):63–72, 2006. ACM Press, New York, NY, USA. ISSN 0163-5980. URL http://www.csm.ornl.gov/~engelman/publications/engelmann06molar.pdf.

Pending Journal Publications

[1] Xubin He, Li Ou, Christian Engelmann, Xin Chen, and Stephen L. Scott. Symmetric active/active metadata service for high availability parallel file systems. *Journal of Parallel and Distributed Computing (JPDC)*, 2008. Elsevier, Amsterdam, The Netherlands. ISSN 0743-7315. Submitted, under review.

Conference Publications

[1] Christian Engelmann, Stephen L. Scott, Chokchai Leangsuksun, and Xubin He. Symmetric active/active replication for dependent services. In *Proceedings of the 3rd International Conference on Availability, Reliability and Security (ARES) 2008*, pages 260–267, Barcelona, Spain, March 4-7, 2008. IEEE Computer Society. ISBN 978-0-7695-3102-1. URL http://www.csm.ornl.gov/~engelman/publications/engelmann08symmetric.pdf.

[2] Li Ou, Christian Engelmann, Xubin He, Xin Chen, and Stephen L. Scott. Symmetric active/active metadata service for highly available cluster storage systems. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS) 2007*, Cambridge, MA, USA, November 19-21, 2007. ACTA Press, Calgary, AB, Canada. ISBN 978-0-88986-703-1. URL http://www.csm.ornl.gov/~engelman/publications/ou07symmetric.pdf.

[3] Li Ou, Xubin He, Christian Engelmann, and Stephen L. Scott. A fast delivery protocol for total order broadcasting. In *Proceedings of the 16th IEEE International Conference on Computer Communications and Networks (ICCCN) 2007*, Honolulu, HI, USA, August 13-16, 2007. IEEE Computer Society. ISBN 978-1-4244-1251-8, ISSN 1095-2055. URL http://www.csm.ornl.gov/~engelman/publications/ou07fast.pdf.

[4] Christian Engelmann, Stephen L. Scott, Chokchai Leangsuksun, and Xubin He. On programming models for service-level high availability. In *Proceedings of the 2nd International Conference on Availability, Reliability and Security (ARES) 2007*, pages 999–1006, Vienna, Austria, April 10-13, 2007. IEEE Computer Society. ISBN 0-7695-2775-2. URL http://www.csm.ornl.gov/~engelman/publications/engelmann07programming.pdf.

[5] Kai Uhlemann, Christian Engelmann, and Stephen L. Scott. JOSHUA: Symmetric active/active replication for highly available HPC job and resource management. In *Proceedings of the 8th IEEE International Conference on Cluster Computing (Cluster) 2006*, Barcelona, Spain, September 25-28, 2006. IEEE Computer Society. ISBN 1-4244-0328-6. URL http://www.csm.ornl.gov/~engelman/publications/uhlemann06joshua.pdf.

[6] Daniel I. Okunbor, Christian Engelmann, and Stephen L. Scott. Exploring process groups for reliability, availability and serviceability of terascale computing systems. In *Proceedings of the 2nd International Conference on Computer Science and Information*

Systems (ICCSIS) 2006, Athens, Greece, June 19-21, 2006. URL http://www.csm.ornl.gov/~engelman/publications/okunbor06exploring.pdf.

Workshop Publications

[1] Christian Engelmann, Stephen L. Scott, Chokchai (Box) Leangsuksun, and Xubin (Ben) He. Symmetric active/active high availability for high-performance computing system services: Accomplishments and limitations. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid) 2008: Workshop on Resiliency in High Performance Computing (Resilience) 2008*, pages 813–818, Lyon, France, May 19-22, 2008. IEEE Computer Society. ISBN 978-0-7695-3156-4. URL http://www.csm.ornl.gov/~engelman/publications/engelmann08symmetric2.pdf.

[2] Christian Engelmann, Hong H. Ong, and Stephen L. Scott. Middleware in modern high performance computing system architectures. In *Lecture Notes in Computer Science: Proceedings of the 7th International Conference on Computational Science (ICCS) 2007, Part II: 4th Special Session on Collaborative and Cooperative Environments (CCE) 2007*, volume 4488, pages 784–791, Beijing, China, May 27-30, 2007. Springer Verlag, Berlin, Germany. ISBN 3-5407-2585-5, ISSN 0302-9743. URL http://www.csm.ornl.gov/~engelman/publications/engelmann07middleware.pdf.

[3] Christian Engelmann, Stephen L. Scott, Chokchai Leangsuksun, and Xubin He. Transparent symmetric active/active replication for service-level high availability. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid) 2007: 7th International Workshop on Global and Peer-to-Peer Computing (GP2PC) 2007*, pages 755–760, Rio de Janeiro, Brazil, May 14-17, 2007. IEEE Computer Society. ISBN 0-7695-2833-3. URL http://www.csm.ornl.gov/~engelman/publications/engelmann07transparent.pdf.

[4] Christian Engelmann, Stephen L. Scott, Chokchai Leangsuksun, and Xubin He. Towards high availability for high-performance computing system services: Accomplishments and limitations. In *Proceedings of the High Availability and Performance Workshop (HAPCW) 2006, in conjunction with the Los Alamos Computer Science Institute (LACSI) Symposium 2006*, Santa Fe, NM, USA, October 17, 2006. URL http://www.csm.ornl.gov/~engelman/publications/engelmann06towards.pdf.

[5] Christian Engelmann, Stephen L. Scott, Chokchai Leangsuksun, and Xubin He. Active/active replication for highly available HPC system services. In *Proceedings of the*

1^{st} *International Conference on Availability, Reliability and Security (ARES) 2006:* 1^{st} *International Workshop on Frontiers in Availability, Reliability and Security (FARES) 2006*, pages 639–645, Vienna, Austria, April 20-22, 2006. IEEE Computer Society. ISBN 0-7695-2567-9. URL http://www.csm.ornl.gov/~engelman/publications/engelmann06active.pdf.

[6] Christian Engelmann and Stephen L. Scott. Concepts for high availability in scientific high-end computing. In *Proceedings of the High Availability and Performance Workshop (HAPCW) 2005, in conjunction with the Los Alamos Computer Science Institute (LACSI) Symposium 2005*, Santa Fe, NM, USA, October 11, 2005. URL http://www.csm.ornl.gov/~engelman/publications/engelmann05concepts.pdf.

[7] Christian Engelmann and Stephen L. Scott. High availability for ultra-scale high-end scientific computing. In *Proceedings of the* 2^{nd} *International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2) 2005, in conjunction with the* 19^{th} *ACM International Conference on Supercomputing (ICS) 2005*, Cambridge, MA, USA, June 19, 2005. URL http://www.csm.ornl.gov/~engelman/publications/engelmann05high.pdf.

Co-Advised MSc Theses

[1] Matthias Weber. High availability for the Lustre file system. Master's thesis, Department of Computer Science, University of Reading, UK, March 14, 2007. URL http://www.csm.ornl.gov/~engelman/students/weber07high.pdf. Double diploma in conjunction with the Department of Engineering I, Technical College for Engineering and Economics (FHTW) Berlin, Berlin, Germany. Advisors: Prof. Vassil N. Alexandrov (University of Reading, Reading, UK); Christian Engelmann (Oak Ridge National Laboratory, Oak Ridge, TN, USA).

[2] Kai Uhlemann. High availability for high-end scientific computing. Master's thesis, Department of Computer Science, University of Reading, UK, March 6, 2006. URL http://www.csm.ornl.gov/~engelman/students/uhlemann06high.pdf. Double diploma in conjunction with the Department of Engineering I, Technical College for Engineering and Economics (FHTW) Berlin, Berlin, Germany. Advisors: Prof. Vassil N. Alexandrov (University of Reading, Reading, UK); George A. Geist and Christian Engelmann (Oak Ridge National Laboratory, Oak Ridge, TN, USA).

Co-Supervised PhD Thesis Internships

[1] Li Ou. *Design of a High-Performance and High-Availability Distributed Storage System*. PhD thesis, Department of Electrical and Computer Engineering, Tennessee Technological University, Cookeville, TN, USA, December 2006. URL http://www.csm.ornl.gov/~engelman/students/ou06design.pdf. Graduate advisory committee: Prof. Xubin He, Periasamy K. Rajan, Prof. Roger L. Haggard, Prof. Martha J. Kosa, Prof. Kwun Lon Ting (Tennessee Technological University, Cookeville, TN, USA); Prof. Jeffrey Norden (State University of New York, Binghamton, NY, USA), and Stephen L. Scott (Oak Ridge National Laboratory, Oak Ridge, TN, USA). Internship supervisors: Christian Engelmann and Stephen L. Scott (Oak Ridge National Laboratory, Oak Ridge, TN, USA).

Contents

List of Figures

List of Tables

Glossary

AIS	Application Interface Specification
API	Application Programming Interface
ASCI	Advanced Simulation and Computing Initiative
BASE	Byzantine Fault Tolerance with Abstract Specification Encapsulation
BLCR	Berkeley Lab Checkpoint Restart
CCSM	Community Climate System Model
CNK	Compute Node Kernel
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
DAS	Direct Attached Storage
DMR	Dual Modular Redundancy
DomU	Privileged Domain
DomU	Unprivileged Domain
DRBD	Distributed Replicated Block Device
DVM	Distributed Virtual Machine
ECC	Error-Correcting Code
FPGA	Field-Programmable Gate Array
FT-FW	Fault-Tolerant Firewall
FT-MPI	Fault-Tolerant Message Passing Interface
HA-OSCAR	High Availability Open Source Cluster Application Resources

HAVEN	High Availability in a Virtualised Environment
HEC	High-End Computing
HP	Hewlett-Packard
HPC	High-Performance Computing
I/O	Input/Output
Id	Identifier
IP	Internet Protocol
IPC	Inter-Process Communication
IPCC	Intergovernmental Panel on Climate Change
IT	Information Technology
LAM	Local Area Multicomputer
MDS	Metadata Service
MPI	Message Passing Interface
MPP	Massively Parallel Processing
MSc	Master of Science
MTBF	Mean-Time Between Failures
MTTF	Mean-Time To Failure
MTTR	Mean-Time To Recover
NAS	Network Attached Storage
NASA	National Aeronautics and Space Administration
NFS	Network File System
Open MPI	Open Message Passing Interface
OpenAIS	Open Application Interface Specification
OS	Operating System

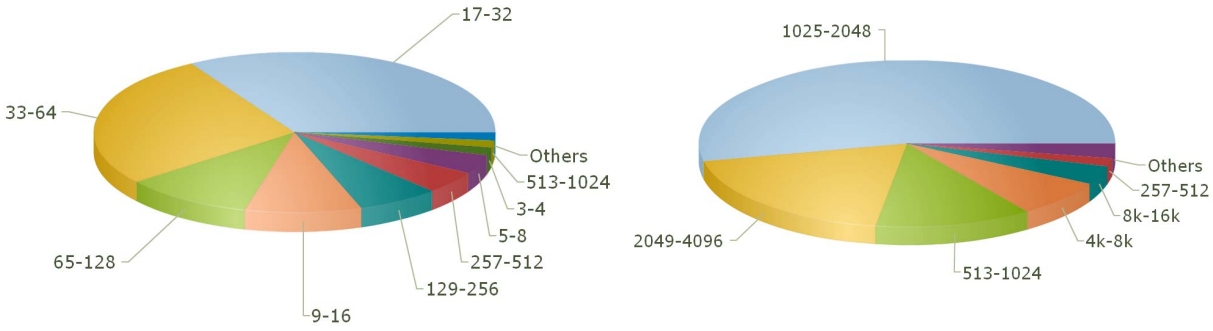
OSS	Object Storage Service
PBS	Portable Batch System
PBS Pro	Portable Batch System Professional
PDF	Probability Density Function
PE	Processing Element
PFlop/s	10^{15} Floating Point Operations Per Second
PhD	Doctor of Philosophy
POSIX	Portable Operating System Interface
PVFS	Parallel Virtual File System
PVM	Parallel Virtual Machine
RAID	Redundant Array of Independent Drives
RAS	Reliability, Availability and Serviceability
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SCSI	Small Computer System Interface
SGE	Sun Grid Engine
SLURM	Simple Linux Utility for Resource Management
SOA	Service-Oriented Architecture
SSI	Single-System Image
STONITH	Shoot The Other Node In The Head, *i.e.,* node fencing
T2CP-AR	Transparent Transmission Control Protocol Active Replication
TCP	Transmission Control Protocol
TICK	Transparent Incremental Checkpointing at Kernel-level
TORQUE	Terascale Open-Source Resource and QUEue Manager

TSI	Terascale Supernova Initiative
UNEP	United Nations Environment Programme
VCL	Virtual Communication Layer
VM	Virtual Machine
WMO	World Meteorological Organisation

1 Introduction

During the last decade, scientific high-performance computing (HPC) has become an important tool for scientists world-wide 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. Furthermore, HPC has played a significant role in scientific engineering applications, where computer simulations have aided the design and testing of electronic components, machinery, cars, air planes, and buildings.

Every year, new larger-scale HPC systems emerge on the market with better raw performance capabilities. Over the last decade, the performance increase of the world-wide fastest HPC systems was not only aided by advances in network and processor design and manufacturing, but also by employing more and more processors within a single closely-coupled HPC system. The Top 500 List of Supercomputer Sites clearly documents this trend toward massively parallel computing systems (Figure 1.1) [1].



(a) November 1997 (b) November 2007

Figure 1.1: Growth of HPC system scale in the Top 500 List of Supercomputer Sites during the last decade (in number of processors/systems) [1]

This growth in system scale poses a substantial challenge for system software and scientific applications with respect to reliability, availability and serviceability (RAS). Although the mean-time to failure (MTTF) for each individual HPC system component, *e.g.*, processor, memory module, and network interface, may be above typical consumer product

standard, the combined probability of failure for the overall system scales proportionally with the number of interdependent components. Recent empirical studies [2, 3] confirm that the enormous quantity of components results in a much lower MTTF for the overall system, causing more frequent system-wide interruptions than displayed by previous, smaller-scale HPC systems.

As a result, HPC centres may artificially set allowable job run time for their HPC systems to very low numbers in order to require a scientific 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 loss of HPC system availability caused by low MTTF and resulting frequent failure-recovery cycles, the demand for continuous HPC system availability has risen dramatically with the recent trend toward 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 extreme-scale HPC machines 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 information technology (IT) communities, have dealt with these issues for their particular solutions 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 extreme-scale HPC systems.

This thesis aims to pave the way for highly available extreme-scale HPC systems by focusing on their most critical system components and by reinforcing them with appropriate high availability solutions. Service components, such as head and service nodes, are the "Achilles heel" of a HPC system. A failure typically results in a complete system-wide outage until repair. This research effort targets efficient software state replication mechanisms for providing redundancy of such service components in extreme-scale HPC systems to achieve high availability as well as high performance.

The next Sections 1.1-1.5 provide a more detailed description of the overall research background, motivation, objectives, methodology, and contribution, followed by a short description of the overall structure of this thesis in Section 1.6. Section 1.7 gives a brief summary of this Chapter.

1.1 Background

Scientific 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 [4]. While distributed computing [5] takes a decentralised approach, parallel computing [6] uses the opposite centralised 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 *high-performance computing* or *supercomputing*. 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 [1], are typically referred to as *high-end computing* (HEC) systems, or extreme-scale HEC systems due to the number of processors they employ.

Scientific computing [7] is the field of study concerned with constructing mathematical models and numerical solution techniques, and using computers to analyse and solve scientific and engineering problems. *Computational science* combines domain-specific science, computational methods, parallel algorithms, and collaboration tools to solve problems in various scientific disciplines, such as in climate dynamics, nuclear astrophysics, fusion energy, nanotechnology, and human genomics. *Extreme-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.

For example, the Community Climate System Model (CCSM) [8] is a fully-coupled, global climate model that provides state-of-the-art computer simulations of the Earth's past, present, and future climate states. The results of climate change scenarios simulated with the CCSM are included in reports to the Intergovernmental Panel on Climate Change (IPCC), which has been established by the World Meteorological Organisation (WMO) and United Nations Environment Programme (UNEP) to assess scientific, technical and socio-economic information relevant for the understanding of climate change, its potential impacts and options for adaptation and mitigation. The IPCC has been awarded the 2007 Nobel Peace Prize, together with former U.S. Vice President Al Gore, for creating "an ever-broader informed consensus about the connection between human activities and global warming" [9–11].

Another example is the Terascale Supernova Initiative (TSI) [12], 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 visualisation. Recent breakthrough accomplishments include a series of 3-dimensional hydrodynamic simulations that show the extent of a supernova shock and the birth of a neutron star.

Both scientific applications, the CCSM and the TSI, have been allocated millions of hours of processor time on the Jaguar Cray XT system [13] at the National Center for Computational Sciences [14] located at Oak Ridge National Laboratory in Oak Ridge, Tennessee, USA. Since this system is equipped with 11,508 dual-core AMD Opteron processors, an application run over the full system would take several weeks. For the year 2007 alone, the CCSM has been allocated 4,000,000 hours of processor time equivalent to 2 weeks and 6 hours of uninterrupted full system usage, and the TSI been allocated 7,000,000 hours of processor time, *i.e.*, 3 weeks, 4 days, and 9 hours.

1.2 Motivation

Recent trends in HPC system design and architecture (see Top 500 List of Supercomputer Sites [1], and Figure 1.1 shown previously) have clearly indicated future increases in performance, in excess of those resulting from improvements in single-processor performance, will be achieved through corresponding increases in system scale, *i.e.*, using a significantly larger component count. As the raw computational performance of the world's fastest HPC systems increases to next-generation extreme-scale capability and beyond, their number of computational, networking, and storage components will grow from today's 10,000 to several 100,000 through tomorrow's 1,000,000 and beyond.

With only very few exceptions, the availability of recently installed systems (Table 1.1) has been lower in comparison to the same deployment phase of their predecessors. In some cases the overall system mean-time between failures (MTBF) is as low as 6.5 hours. Based on personal communication with various HPC centre personnel and publicly available statistics [15–19], the overall system availability of currently operational HPC systems is roughly above 96% and below 99%.

Installed	System	Processors	MTBF	Measured	Source
2000	ASCI White	8,192	40.0h	2002	[15]
2001	PSC Lemieux	3,016	9.7h	2004	[16]
2002	Seaborg	6,656	351.0h	2007	[17]
2002	ASCI Q	8,192	6.5h	2002	[18]
2003	Google	15,000	1.2h	2004	[16]
2006	Blue Gene/L	131,072	147.8h	2006	[19]

Table 1.1: Publicly available past and current HPC system availability statistics

However, these availability statistics are not comparable due to missing strict specifications for measuring availability metrics for HPC systems. For example, most HPC centres measure the MTBF without including the application recovery time in contrast to the more strict definition for MTBF as the actual time spent on useful computation between *full* system recovery and the next failure. The same HPC system with and without high-bandwith networked file system support would achieve the same MTBF, since parallel application restart time, *i.e.*, the time it takes to load the last checkpoint from the file system to all compute nodes, is not included. Additionally, the time spent on fault tolerance measures, such as writing a checkpoint from all compute nodes to the file system, is typically not included as planned downtime either.

In order to provide accurate and comparable availability statistics, a RAS taxonomy standard for HPC systems is needed that incorporates HPC system hardware and software architectures as well as user-, centre-, and application-specific use cases.

A recent study [20] performed at Los Alamos National Laboratory estimates the MTBF for a next-generation extreme-scale HPC system. Extrapolating from current HPC system performance, scale, and overall system MTTF, this study suggests that the MTBF will fall to only 1.25 hours of useful computation on a 1 PFlop/s (10^{15} floating point operations per second) next-generation supercomputer. Another related study from Los Alamos National Laboratory [21] shows that a Cray XD1 system [22] with the same number of processors as the Advanced Simulation and Computing Initiative (ASCI) Q system [23], with almost 18,000 field-programmable gate arrays (FPGAs) and 16,500 GB of error-correcting code (ECC) protected memory, would experience one non-recoverable radiation-induced soft error (double-bit memory or single-bit logic error) once every 1.5 hours, if located at Los Alamos, New Mexico, USA, *i.e.*, at 7,200 feet altitude.

The first Los Alamos National Laboratory study [20] also estimates the overhead of the current state-of-the-art fault tolerance strategy, checkpoint/restart, for a 1 PFlop/s system, showing that a computational job that could normally complete in 100 hours in a failure-free environment will actually take 251 hours to complete, once the cost of checkpointing and failure recovery is included. What this analysis implies is startling: more than 60% of the processing power (and investment) on these extreme-scale HPC systems may be lost due to the overhead of dealing with reliability issues, unless something happens to drastically change the current course.

In order to provide high availability as well as high performance to future-generation extreme-scale HPC systems, research and development in advanced high availability and fault tolerance technologies for HPC is needed.

1.3 Objectives

There are two major aspects in providing high availability for HPC systems. The first considers head and service nodes. A HPC system is typically controlled using a single head node, which stores state informnation about the entire system, provides important system-wide services, and acts as a gateway between the compute nodes inside the system and the outside world. Service nodes typically offload specific head node services to provide for better scalability and performance. The second aspect involves compute nodes, which perform the actual scientific computation. While there is only one head node in a HPC system, there may be hundreds of service nodes, and many more compute nodes. For example, the Jaguar Cray XT system [13] has 11,706 nodes: one head node, 197 service nodes, and 11,508 compute nodes.

Each of these two aspects in providing high availability for HPC systems requires a separate approach as the purpose and scale of head and service nodes are significantly different to the purpose and scale of compute nodes. Solutions that may be applicable to head and service nodes, such as node redundancy, may not suitable for compute nodes due to their resource requirement and performance impact at scale.

This thesis focuses on the aspect of providing high availability for HPC system head and service nodes, as these components are the "Achilles heel" of a HPC system. They are the command and control backbone. A failure typically results in a complete system-wide outage until repair. Moreover, high availability solutions for compute nodes rely on the functionality of these nodes for fault-tolerant communication and reconfiguration.

When this research work started in 2004, there were only a small number of high availability solutions for HPC system head and service nodes (Section 2.1). Most of them focused on an active/standby node redundancy strategy with some or full service state replication. Other previous research in providing high availability in distributed systems using state-machine replication, virtual synchrony, and process group communication systems has been performed in the early-to-mid 1990s (Section 2.3).

This work aims to combine the research efforts in high availability for distributed systems and in high availability for HPC system head and service nodes, and to extend them to provide the highest level of availability for HPC system head and service nodes. This thesis research targets efficient software state replication mechanisms for the redundancy of services running on HPC head and service nodes to achieve high availability as well as high performance. Moreover, it is its intend to offer a theoretical foundation for head and service node high availability in HPC as part of the greater effort in defining a HPC RAS taxonomy standard. This work further aims at offering practical proof-of-concept prototype solutions of highly available HPC head and service nodes. This thesis:

- examines past and ongoing efforts in providing high availability for head, service, and compute nodes in HPC systems, for distributed systems, and for services in the IT and telecommunication industry
- provides a high availability taxonomy suitable for HPC head and service nodes
- generalises HPC system architectures and identifies specific availability deficiencies
- defines the methods for providing high availability for HPC head and service nodes
- designs, implements, and tests the following proof-of-concept prototypes for providing high availability for HPC head and service nodes:
  - a fully functional proof-of-concept prototype using external symmetric active/active replication for the HPC job and resource management service
  - a fully functional proof-of-concept prototype using internal symmetric active/active replication for the HPC parallel file system metadata service
  - a preliminary proof-of-concept prototype using external and internal transparent symmetric active/active replication for a service-level high availability software framework
  - a preliminary proof-of-concept prototype using external and internal transparent symmetric active/active replication for a service-level high availability software framework with support for dependent services

1.4 Methodology

This research combines previous efforts in service-level high availability (Chapter 2) by providing a generic high availability taxonomy and extending it to the needs of high availability and high performance within HPC environments (Section 3.1). This thesis introduces new terms, such as *asymmetric active/active* and *symmetric active/active*, to resolve existing ambiguities of existing terms, such as *active/active*. It also clearly defines the various configurations for achieving high availability of service, such as *active/standby*, *asymmetric active/active* and *symmetric active/active*.

This thesis further examines current HPC system architectures in detail (Section 3.2). A more generalised HPC system architecture abstraction is introduced to allow the identification of architectural availability deficiencies. HPC system services are categorised into *critical* and *non-critical* to describe their impact on overall system availability. HPC system nodes are categorised into *single points of failure* and *single points of control* to

pinpoint their involvement in system failures, to describe their impact on overall system availability, and to identify their individual need for a high availability solution.

Based on the introduced extended high availability taxonomy, this thesis defines various methods for providing service-level high availability (Section 3.3) for identified HPC system head and service nodes using a *conceptual service model* and detailed descriptions of *service-level high availability methods* and their properties. A theoretical comparison of these methods with regards to their performance overhead and provided availability is offered, and similarities and differences in their programming interfaces are examined.

This thesis further details a series of developed proof-of-concept prototypes (Chapter 4) for providing high availability for HPC head and service nodes using the *symmetric active/active replication* method, *i.e.*, *state-machine replication*, to complement prior work in this area using active/standby and asymmetric active/active configurations. In addition to specific objectives and intermediate conclusions, various architecture, design and performance aspects are discussed for each prototype implementation.

The developed proof-of-concept prototypes centre around a multi-layered symmetric active/active high availability framework concept that coordinates individual solutions with regards to their respective field, and offers a modular approach that allows for adaptation to system properties and application needs.

The first proof-of-concept prototype (Section 4.2) provides symmetric active/active high availability for the most important HPC system service running on the head node, the *job and resource management service*, also commonly referred to as batch job scheduler or simply the scheduler. This fully functional prototype relies on the concept of *external symmetric active/active replication*, which avoids modification of existing code of a complex service, such as the HPC job and resource management service, by wrapping a it into a virtually synchronous environment. This proof-of-concept prototype also involves a *distributed mutual exclusion mechanism* that is needed to unify replicated output.

The second proof-of-concept prototype (Section 4.3) offers symmetric active/active high availability for one of the second most important HPC system services running on the head node of small-scale HPC systems and on a dedicated service node in large-scale HPC systems, the latency-sensitive *metadata service of the parallel file system*. This fully functional prototype relies on the concept of *internal symmetric active/active replication*, which requires modification of the service and therefore may yield better performance.

The third proof-of-concept prototype (Section 4.4) is a *transparent symmetric active/active replication* software framework for service-level high availability in *client-service scenarios*. This preliminary proof-of-concept prototype completely hides the replication infrastructure within a virtual communication layer formed by interceptor processes in case of external replication or adaptor components in case of internal replication.

The fourth proof-of-concept prototype (Section 4.5) is a *transparent symmetric active/active replication* software framework for service-level high availability in *client-service as well as dependent service scenarios*. By using a high-level abstraction, this preliminary proof-of-concept prototype, in addition to normal dependencies between clients and services, maps decompositions of service-to-service dependencies into respective orthogonal dependencies between clients and services onto a replication infrastructure consisting of multiple virtual communication layers.

1.5 Contribution

This thesis research yields several major contributions to the field of service-level high availability in general and to high-level HPC system RAS in specific:

1. This thesis offers a modern theoretical foundation for providing service-level high availability that includes traditional redundancy strategies, such as various active/standby configurations, as well as advanced redundancy strategies based on state-machine replication. Existing ambiguities of terms are resolved and the concept of state-machine replication using virtual synchrony is integrated.

2. This work provides an insight into specific availability deficiencies of HPC system hardware and software architectures with respect to critical system services, single points of failures, and single points of control. It also clearly identifies two different failure modes of HPC systems: (1) system-wide outage until repair, and (2) partial system outage in degraded operating mode.

3. This research gives a unified definition of various service-level high availability methods using a conceptual service model. Individual replication algorithms are described in detail, and their performance and provided availability are analysed. The comparison of traditional and advanced redundancy strategies clearly shows the trade-off between performance impact and achieved availability.

4. The developed fully-functional proof-of-concept prototype for providing symmetric active/active high availability for the job and resource management service is the first of its kind in high-level HPC system RAS. The concept of external replication is a new contribution to the field of service-level high availability, which allows service state replication without modifying the service itself. With a node MTTF of 5,000 hours, this solution improves service availability from 99.285% to 99.994% in a two-node system, and to 99.99996% with three nodes. As a great proof-of-concept value,

this prototype demonstrates a distributed mutual exclusion using a process group communication system.

5. The implemented fully-functional proof-of-concept solution for providing symmetric active/active high availability for the metadata service of the parallel file system is the second of its kind in high-level HPC system RAS. The concept of internal replication is a new contribution to the field of service-level high availability, which permits tight integration of service state replication mechanisms with the service for better performance. As a great proof-of-concept value, this prototype demonstrates high performance of a state-machine replication solution.

6. The developed preliminary proof-of-concept prototypes for a transparent symmetric active/active replication software framework for client-service and dependent service scenarios are significant contributions to the field of service-level high availability. Both prototypes demonstrate for the first time that a state-machine replication infrastructure that can be deployed transparently (external replication) or semi-transparently (internal replication), completely or partially invisible to clients and services. Moreover, the replication infrastructure is able to deal with service interdependencies by decomposing service-to-service dependencies into respective orthogonal client-service dependencies.

1.6 Structure

This thesis is structured as follows. While this Chapter 1 contains a more detailed description of the overall research background, motivation, objectives, methodology, and contribution, the following Chapter 2 evaluates previous work within this context. Chapter 3 lays the theoretical ground work by defining a modern service-level high availability taxonomy, describing existing availability deficiencies in HPC system architectures using a generalised model, and detailing methods for providing high availability to services running on HPC system head and service nodes. Chapter 4 presents designs and test results of various developed proof-of-concept prototypes for providing symmetric active/active high availability to services on head and service nodes. While each of these chapters finishes with an individual summary, Chapter 5 concludes this thesis with an overall summary and evaluation of the presented research, and a discussion of future directions.

1.7 Summary

This Chapter provided a more detailed description of the overall thesis research background, motivation, objectives, methodology, and contribution. The research background of scientific HPC and its significance to other science areas, such as climate dynamics, nuclear astrophysics, and fusion energy, has been explained. The trend toward larger-scale HPC systems beyond 100,000 computational, networking, and storage components and the resulting lower overall system MTTF has been detailed. The main objective of this thesis, efficient software state replication mechanisms for the redundancy of services running on HPC head and service nodes, has been stated and motivated by the fact that such service components are the "Achilles heel" of a HPC system.

The methodology of this thesis has been outlined with respect to theoretical ground work and prototype development. The described theoretical ground work included defining a modern theoretical foundation for service-level high availability, identifying availability deficiencies of HPC system architectures, and comparing various service-level high availability methods. The outlined prototype development encompassed several proof-of-concept solutions for providing high availability for services running on HPC head and service nodes using symmetric active/active replication.

The major research contributions of this thesis have been summarised. The theoretical ground work contributed a generic taxonomy for service high availability, an insight into availability deficiencies of HPC system architectures, and a unified definition of service-level high availability methods. The prototype development contributed two fully functional symmetric active/active high availability solutions, for a HPC job and resource management service and for the HPC parallel file system metadata service, and two preliminary prototypes for a transparent symmetric active/active replication software framework, for client-service and for dependent service scenarios.

2 Previous Work

Existing high availability solutions for HPC systems can be categorised into head and service node redundancy strategies, and compute node fault tolerance mechanisms. Different techniques are being used due to the difference in scale, *i.e.*, the number of nodes involved, and purpose, *i.e.*, system services vs. parallel application. In the following, existing high availability and fault tolerance solutions for HPC systems are briefly described and their advantages and deficiencies are examined.

In addition, past research and development efforts in providing high availability in distributed systems using state-machine replication, virtual synchrony, and process group communication systems are illustrated and their applicability for HPC head and service node redundancy solutions is evaluated.

Also, existing solutions and ongoing research and development efforts in providing high availability for services in the IT and telecommunication industry are examined.

2.1 Head and Service Node 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. Individual mechanisms range from active/standby mechanisms to high availability clustering techniques (Table 2.1).

2.1.1 Active/Standby using Shared Storage

The concept of using a *shared storage* device (Figure 2.1) 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

	Max. Service Nodes	Shared Storage	Active/Standby	Crosswise Active/Standby	2 + 1 High Availability Clustering	Linux	DRBD
SLURM	2	×	×			×	
Sun Grid Engine	2	×	×			×	
PVFS Metadata Service	2	×	×	×		×	×
Lustre Metadata Service	2	×	×	×		×	×
HA-OSCAR	2		×		×	×	
PBS Pro for Cray Platforms	2		×			×	
Moab Workload Manager	2		×			×	

×, Requirement or available feature

Table 2.1: Requirements and features comparison between head and service node high availability solutions

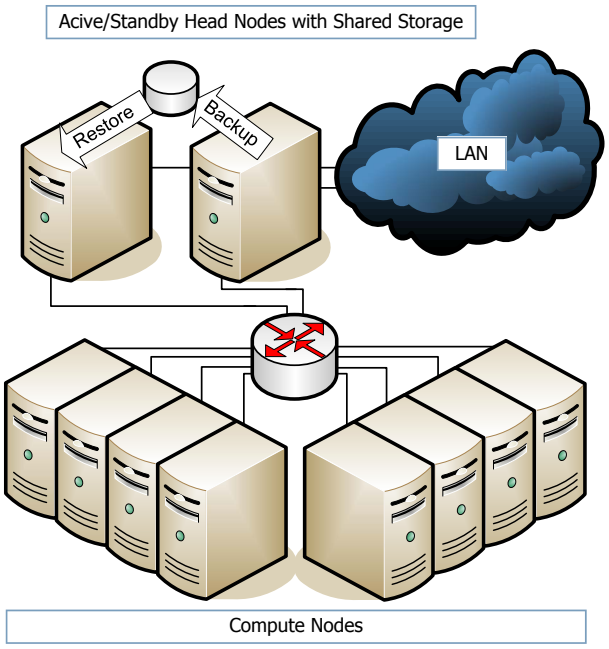


Figure 2.1: Active/standby HPC head nodes using shared storage

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 [24–26] and initiates the fail-over procedure. An extension of this technique uses a crosswise active/standby redundancy strategy. In this case, both are active services and additional standby services for each other. In both cases, the mean-time to recover (MTTR) depends on the heartbeat interval, which may vary between a few seconds and several minutes.

While the shared storage device is typically an expensive redundant array of independent drives (RAID) and therefore highly available, it remains a single point of failure and control. Furthermore, file system corruption on the shared storage device due to failures occurring during write operations are not masked unless a journaling file system is used and an incomplete backup state is discarded, *i.e.*, a commit protocol for saving backup state is used. Correctness and quality of service are not guaranteed if no commit protocol is used. The requirement for a journaling file system impacts the fail-over procedure by adding a file system check, which in-turn extends the MTTR.

The shared storage device solution for providing service-level high availability has become very popular with the heartbeat program [24–26], which includes failure detection and automatic failover with optional Internet protocol (IP) address cloning feature. Recent enhancements include support for file systems on a Distributed Replicated Block Device (DRBD) [27–29], which is essentially a storage mirroring solution that eliminates the single shared storage device and replicates backup state to local storage of standby services. This measure is primarily a cost reduction, since an expensive RAID system is no longer required. However, the requirement for a journaling file system and commit protocol remain to guarantee correctness and quality of service, since the DRBD operates at the block device level for storage devices and not at the file system level.

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

Simple Linux Utility for Resource Management

The Simple Linux Utility for Resource Management (SLURM) [30–32] 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 [33, 34]. It offers high availability using an active/standby redundancy strategy with a secondary head node and a shared storage device.

Sun Grid Engine

The Sun Grid Engine (SGE) [35] is a job and resource management solution for Grid, cluster, and HPC environments. It provides scalable policy-based workload management and dynamic provisioning of application workloads. SGE is the updated commercial version of the open source Grid Engine project [36]. Its *shadow master* configuration [37] offers high availability using an active/standby redundancy strategy with a secondary head node and a shared storage device.

Parallel Virtual File System Metadata Service

The Parallel Virtual File System (PVFS) [38–40] is a file system for HPC that utilises parallel input/output (I/O) in order to eliminate bottlenecks. One of the main components of any parallel file system is the metadata service (MDS), which keeps records of all stored files in form of a directory service. This MDS is a critical system service typically located on head or service nodes. PVFS offers two high availability configurations for its MDS involving a secondary node and a shared storage device, active/standby and crosswise active/standby. Both configurations are based on the earlier mentioned heartbeat program [24–26].

Lustre Metadata Service

Similar to PVFS, Lustre [41–43] 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 service. 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 based on the earlier mentioned heartbeat program [24–26].

2.1.2 Active/Standby Replication

Service-level *active/standby* high availability solutions for head and service nodes in HPC systems (Figure 2.2) typically perform state change validation to maintain consistency of backup state. The primary service copies its state to the standby service in regular intervals or on any change using a validation mechanism in order to avoid corruption of backup state when failing during the copy operation. The new backup state is validated and the old one is deleted only when the entire state has been received by the standby. Furthermore, active/standby solutions may use additional two-phase commit protocols to ensure consistency between active and standby nodes. In this case, the primary service

first announces its state change to the standby service, and then commits it after it received an acknowledgement back. Once committed, the state at the standby service is updated accordingly. Commit protocols provide active/standby with transparent fail-over, *i.e.*, no state is lost and only an interruption of service may be noticed.

The following active/standby replication solutions exist for head and service nodes of HPC systems.

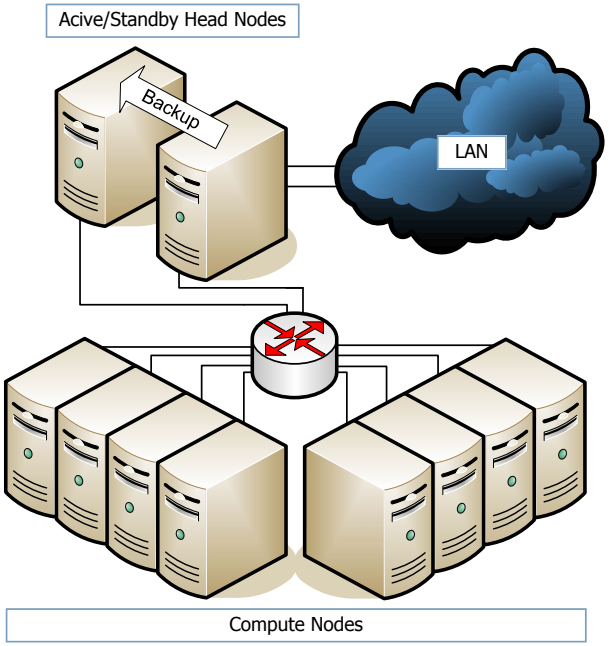


Figure 2.2: Active/standby HPC head nodes using replication

High Availability Open Source Cluster Application Resources

High Availability Open Source Cluster Application Resources (HA-OSCAR) [44–46] is a high availability framework for the OpenPBS [47] batch job and system resource management service on the head node. OpenPBS is the original version of the Portable Batch System (PBS). It is a flexible batch queueing system developed for the National Aeronautics and Space Administration (NASA) in the early- to mid-1990s. The PBS service interface has become a standard in batch job and system resource management for HPC.

OpenPBS offers batch job and system 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 an active/standby redundancy strategy involving a secondary head node. Service state is replicated to the the standby upon modification, while the standby service takes over based on the current state. The standby node monitors the health of the primary node using the heartbeat mechanism and initiates

the fail-over. However, OpenPBS does temporarily loose control of the system in this case. All previously running jobs on the HPC system are automatically restarted.

The MTTR of HA-OSCAR depends on the heartbeat interval and on the time currently running jobs need to recover to their previous state. The HA-OSCAR solution integrates with checkpoint/restart compute node fault tolerance solutions (Section 2.2.1), improving its MTTR to 3-5 seconds for fail-over plus the time currently running jobs need to catch up based on the last checkpoint.

Portable Batch System Professional for Cray Platforms

The professional edition of the original PBS implementation, PBS Professional (PBS Pro) [48], is a commercial variant that operates in networked multi-platform UNIX environments, and supports heterogeneous clusters of workstations, supercomputers, and massively parallel systems.

PBS Pro for Cray HPC platforms [49] supports high availability using an active/standby redundancy strategy involving Crays proprietary interconnect network for message duplication and transparent fail-over. Service state is replicated to the the standby node by delivering all messages to both, the active node and the standby node, while the standby service takes over based on the current state. PBS Pro does not loose control of the system. The network interconnect monitors the health of the primary service and initiates the fail-over. This solution is only available for Cray HPC systems as it is based on Crays proprietary interconnect technology. The MTTR of PBS Pro for Cray HPC systems is close to 0. However, the availability of this two head node system is still limited by the event of both head nodes failing at the same time.

Moab Workload Manager

Moab Workload Manager [50, 51] is a policy-based job scheduler and event engine that enables utility-based computing for clusters. It simplifies management across one or multiple hardware, operating system, storage, network, license and resource manager environments to increase the return of investment of clustered resources, improve system utilisation to run between 90-99%, and allow for expansion. Moab Workload Manager is being used in HPC systems as batch job and system resource management service on the head node and employs an active/standby mechanism with transparent fail-over using an internal heartbeat mechanism as well as an internal standby synchronisation technique.

2.1.3 High Availability Clustering

The term *high availability clustering* (Figure 2.3) is commonly used to describe the concept of using a group of service nodes to essentially provide the same single service using load balancing in order to provide uninterrupted processing of new incoming service requests. Active/standby replication for all service nodes together, for a subset, or for each service node individually may be used to provide continued processing of existing service requests. This leads to a variety of high availability configurations, such as $n+1$ and $n+m$ with n active nodes and 1 or m standby nodes. The earlier mentioned crosswise active/standby redundancy solution where two active service nodes are additional standby service nodes for each other is also sometimes referred to as an *active/active* high availability clustering configuration.

High availability clustering targets high throughput processing of a large number of small service requests with no or minimal service state change, such as a Web service. This technique should not be confused with high availability cluster computing, where applications run on a set of compute nodes in *parallel*. In high availability cluster computing, the compute nodes perform a common task and depend on each other to produce a correct result, while the service nodes in high availability clustering perform independent tasks that do not depend on each other. However, there are use cases where both overlap, such as for embarrassingly parallel applications using task farming. Examples are SETI@HOME [52–54] and Condor [55–57].

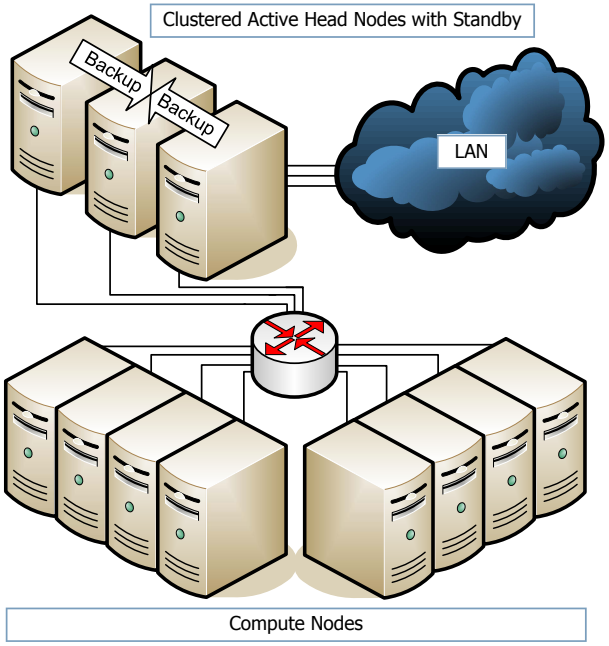


Figure 2.3: Clustered HPC head nodes with standby $(2+1)$

High Availability Open Source Cluster Application Resources

As part of the HA-OSCAR research, a high availability clustering prototype implementation [58] has been developed that offers batch job and system resource management for HPC systems in a high-throughput computing scenario. Two different batch job and system resource management services, OpenPBS and the Sun Grid Engine [35], run independently on different head nodes at the same time, while one 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, OpenPBS and Sun Grid Engine do loose control of the system during fail-over, requiring a restart of lost jobs. The MTTR is 3-5 seconds for fail-over plus the time currently running jobs need to catch up.

Despite the existence of two active and one standby services, 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 healthy head node serving the system.

2.1.4 Node Fencing

The practice of node fencing, also commonly referred to as "shoot the other node in the head" (STONITH) [59], is a mechanism whereby the "other node" is unconditionally powered off by a command sent to a remote power supply. This is a crude, "big hammer" approach of node membership management. It is typically used to avoid further corruption of the system by an incorrect or malicious node. A softer approach just reboots the node or otherwise brings it into a known stable disconnected state.

Node fencing is typically employed in service redundancy strategies, *e.g.*, in order to shut off the failed primary node during a fail-over.

2.2 Compute Node Solutions

High availability solutions for compute nodes of HPC systems are typically based on a state redundancy strategy without node redundancy involving a shared storage device and appropriate validation mechanisms for consistency. Compute node state is copied to a shared storage device regularly or on any change, and validated once a consistent backup has been performed. However, complete redundancy, *i.e.*, a matching standby node for every compute node, is not an appropriate mechanism for HPC systems due to the number of compute nodes involved.

Spare compute nodes may be used to replace lost compute nodes after a failure, or compute nodes may be oversubscribed, *i.e.*, failed computational processes are restarted

on nodes that are already occupied by other computational processes.

Currently, there are two major techniques for compute node fault tolerance: *checkpoint/restart* and *message logging*. Additionally, ongoing research efforts also focus on *algorithmic fault tolerance* and *proactive fault avoidance*.

2.2.1 Checkpoint/Restart

Checkpoint/restart is a commonly employed technique for compute node fault tolerance. Application, process, or the entire operating system (OS) state from all compute nodes is copied to a shared storage device in regular intervals, and validated once a consistent backup has been performed. The shared storage device is typically a high-performance networked file system, such as Lustre [41–43]. The checkpoint/restart mechanism requires coordination of *all* involved compute nodes in order to ensure consistency, *i.e.*, to perform a global snapshot. Upon failure, all surviving compute nodes roll back to the last checkpoint (backup) and failed compute nodes are restarted from the last checkpoint. Progress made between the last checkpoint and the time of failure is lost.

Checkpointing a HPC system, *i.e.*, backing up its current state, is a preemptive measure performed during normal operation and needs to be counted as planned downtime. Restarting a HPC system, *i.e.*, restoring the last backup state, is a reactive measure executed upon failure. The MTTR of checkpoint/restart systems is defined by the time it takes to detect the failure, the time to restore, and the time it takes for the compute nodes to catch up to their previous state.

Checkpoint/restart solutions typically scale approximately linearly, *i.e.*, planned downtime and MTTR grow linear with the number of compute nodes ($O(n)$), assuming that the high-performance networked file system is scaled up appropriately as well to avoid that access to the shared storage device becomes a bottleneck.

Several checkpoint/restart solutions exist. Many applications utilise their own checkpoint/restart mechanism that writes out a checkpoint after a set of iterative computations have been performed or specific computational phases have been finished. Upon failure, these applications are able to restart from such intermediate result. Another solution is to transparently stop and checkpoint an application at a lower software layer, such as messaging or OS, and to transparently reinstate an application after a failure.

Berkeley Lab Checkpoint Restart

The Berkeley Lab Checkpoint Restart (BLCR) [60–62] solution is a hybrid kernel/user implementation of checkpoint/restart for applications on the Linux OS. BLCR performs checkpointing and restarting inside the Linux OS kernel. While this makes it less portable

than solutions that use user-level libraries, it also means that it has full access to all kernel resources, and is able to restore resources, like process identifiers, that user-level libraries can‘t.

BLCR does not support checkpointing certain process resources. Most notably, it will not checkpoint and/or restore open sockets or inter-process communication (IPC) objects, such as pipes or shared memory. Such resources are silently ignored at checkpoint time and are not restored. Applications can arrange to save any necessary information and reacquire such resources at restart time.

Local Area Multicomputer (LAM) [63, 64] is an implementation of the Message Passing Interface (MPI) [65] standard used by applications for low-latency/high-bandwith communication between compute nodes in HPC systems. LAM integrates directly with the BLCR solution and allows transparent checkpoint/restart of MPI-based applications.

Transparent Incremental Checkpointing at Kernel-level

Transparent Incremental Checkpointing at Kernel-level (TICK) [66, 67] is a Linux 2.6.11 kernel module that provides incremental checkpointing support transparently to applications. The general concept of incremental checkpointing allows to take application snapshots and to reduce checkpoint data to a minimum by only saving the difference to the previous checkpoint. This technique may require a local copy of the last full checkpoint, while reduced checkpoint data is saved to a high-performance networked file system.

In case of a restart, all correct nodes roll back using their local copy, while failed nodes reconstruct the last checkpoint from the high-performance networked file system. In order to speed up the restart process, reconstruction of the last checkpoint for every compute node may be performed by the high-performance networked file system itself or by agents on the storage nodes. For incremental checkpointing in diskless HPC systems, checkpoint data reduction may be performed by only saving changed memory pages.

DejaVu

DejaVu [68, 69] is a fault tolerance system for transparent and automatic checkpointing, migration, and recovery of parallel and distributed applications. It provides a transparent parallel checkpointing and recovery mechanism that recovers from any combination of systems failures without any modification to parallel applications or the OS. It uses a new runtime mechanism for transparent incremental checkpointing that captures the least amount of state needed to maintain global consistency and provides a novel communication architecture that enables transparent migration of existing MPI applications without source-code modifications.

The DejaVu fault tolerance system has been integrated into the commercial Evergrid Availability Services product [70].

Diskless Checkpointing

Diskless checkpointing [71–75] saves application state within the memory of dedicated checkpointing nodes, spare compute nodes, service nodes, or compute nodes in use, without relying on stable storage. It eliminates the performance bottleneck of checkpointing to a shared stable storage by (1) utilising memory, which typically has a much lower read/write latency than disks, and by (2) placing checkpoint storage within a HPC system, thus further improving access latency to stored state.

Similarly to stable storage solutions, data redundancy strategies can be deployed to provide high availability for stored state. In contrast to stable storage solutions, diskless checkpointing saves the state of an application only as long as the HPC system is powered on and typically only for its job run, *i.e.*, for the time an application is supposed to run. Diskless checkpointing may be provided by an underlying framework or performed internally by the application itself.

2.2.2 Message Logging

The concept of message logging is based on the idea of capturing all messages of a HPC application. Upon failure of a process, this process is restarted and its messages are played back in order to catch up to the previous state. Message logging may be combined with checkpoint/restart solutions to avoid playback from application start. However, message logging is only applicable to deterministic applications as message playback is used to recover application state.

Message logging solutions have a direct impact on system performance during normal operation as all messages on the network may be doubled. Furthermore, centralised message logging servers are bottlenecks. However, message logging in combination with checkpoint/restart permits uncoordinated checkpointing, *i.e.*, each process of a parallel application is able to checkpoint individually while maintaining a message log to ensure a global recovery line for consistency.

At this moment, there exists only one message logging solution for HPC system compute node fault tolerance, MPICH-V.

MPICH-V

MPICH-V [76–78] is a research effort with theoretical studies, experimental evaluations, and pragmatic implementations aiming to provide a MPI implementation featuring mul-

tiple fault tolerant protocols based on message logging. MPICH-V provides an automatic fault-tolerant MPI library, *i.e.*, a totally unchanged application linked with the MPICH-V library is a fault-tolerant application. Currently, MPICH-V features five different protocols:

- MPICH-V1 (deprecated) is designed for very large-scale HPC systems using heterogeneous networks. It is able to tolerate a very high number of faults, but it requires a large bandwidth for stable components to reach good performance.
- MPICH-V2 (deprecated) is designed for very large-scale HPC systems using homogeneous networks. It only requires a very small number of stable components to reach good performance as it is based on an uncoordinated checkpointing protocol.
- MPICH-VCausal (deprecated) is designed for low-latency dependent applications which must be resilient to a high failure frequency. It combines the advantages of V1 and V2 with direct communication and absence of acknowledgements.
- MPICH-VCL is designed for extra low-latency dependent applications. The Chandy Lamport algorithm [79] used in MPICH-VCL does not introduce any overhead during fault free execution. However, it requires restarting all nodes in the case of a single failure.
- MPICH-PCL features a Blocking Chandy Lamport fault tolerant protocol, which consists of a new communication channel and two components, a checkpoint server and a specific dispatcher, supporting large-scale and heterogeneous applications.

2.2.3 Algorithm-Based Fault Tolerance

The notion of *algorithm-based fault tolerance* re-emerged recently with the trend toward massively parallel computing systems with 100,000 and more processors. The core concept is to design computational algorithms to either ignore failures and still deliver an acceptable result, or to implicitly recover using redundant computation and/or redundant data. A major requirement for algorithm-based fault tolerance is that the underlying system (hardware and software) is capable of detecting and ignoring failures.

Fault-Tolerant Message Passing Interface

Fault-tolerant Message Passing Interface (FT-MPI) [80–82] is a full 1.2 MPI specification implementation that provides process-level fault tolerance at the MPI application programming interface (API) level. FT-MPI is built upon the fault-tolerant Harness runtime

system (Section 2.3.4). It survives the crash of $n-1$ processes in a n-process parallel application, and, if required, can restart failed processes. However, it is still the responsibility of the application to recover data structures and data. FT-MPI is essential for algorithmic fault tolerance as is provides the underlying fault-tolerant software system.

Open Message Passing Interface

Open Message Passing Interface (Open MPI) [83, 84] is a project combining technologies and resources from several other projects in order to build the best MPI library available. A completely new MPI-2 compliant implementation, Open MPI offers advantages for system and software vendors, application developers, and computer science researchers.

Open MPI is founded upon a component architecture that is designed to foster 3^{rd} party collaboration by enabling independent developers to use Open MPI as a production-quality research platform. Specifically, the component architecture was designed to allow small, discrete implementations of major portions of MPI functionality, *e.g.*, point-to-point messaging, collective communication, and runtime environment support.

Ongoing work targets the integration of checkpoint/restart using BLCR (Section 2.2.1), message logging based on the MPICH-V mechanisms (Section 2.2.2), and support for algorithm-based fault tolerance similar to FT-MPI.

Data Redundancy

A recent effort in algorithm-based checkpoint-free fault tolerance for parallel matrix computations [85] takes the approach to encode the data an algorithm works on to contain redundancy. Data redundancy is adjustable by the encoding algorithm, such that a specific number of compute node failures can be tolerated at the same time. A light-weight implicit recovery mechanism using only local computation on the data of correct compute nodes is able to recode the data, such that no data is lost. An application of this technique to matrix-matrix multiplication shows very low performance overhead.

Computational Redundancy

Instead of keeping redundant data or even employing redundant computational processes that operate on the same data, the idea of computational redundancy relies on the algorithm-based relationship between individual data chunks of a parallel application. If a data chunk gets lost due to a compute node failure, the impact on the result may be within the margin of error or may be recoverable by running the surviving nodes a little bit longer. Therefore, a certain number of compute node failures may be tolerated by simply ignoring them.

An example for this technique was implemented by me a couple of years ago as part of the research in cellular architectures [75]. It focused on the development of algorithms that are able to use a 100,000-processor machine efficiently and are capable of adapting to or simply surviving faults. In a first step, a simulator in Java was developed, since a 100,000-processor machine was not available at that time. The prototype was able to emulate up to 500,000 virtual processors on a cluster with 5 real processors solving Laplace's equation and the global maximum problem while ignoring failures.

2.2.4 Proactive Fault Avoidance

Proactive fault avoidance recently emerged as a new research field. In contrast to traditional reactive fault handling, a reliability-aware runtime may provide a new approach for fault tolerance by performing preemptive measures that utilise reliability models based on historical events and current system health status information in order to avoid application faults. For example, a process, task, or virtual machine may be temporarily migrated away from a compute node that displays a behaviour similar to one that is about to fail. Pre-fault indicators, such as a significant increase in heat, an unusual number of network communication errors, or a fan fault, can be used to avoid an imminent application fault through anticipation and reconfiguration.

Early prototypes [86] and simulations [87] suggest that migration can be performed very quickly with minimal performance overhead, while still utilising reactive fault tolerance measures, such as checkpoint/restart, for unanticipated failures. Proactive fault avoidance techniques require highly available head and service nodes as an underlying support framework to perform scalable system monitoring and migration.

2.3 Distributed Systems Solutions

Parallel systems research has always taken the optimistic approach toward fault tolerance, *i.e.*, system components fail rarely and performance has higher priority. In contrast, distributed systems research [5] has always taken the pessimistic approach, *i.e.*, system components fail often and fault tolerance has higher priority. With the advent of extreme-scale HPC systems with 100,000 networked processors and beyond, distributed system solutions may become more attractive as fault recovery in these massively parallel systems becomes more expensive. Moreover, even with small-scale problems, such as HPC system head and service node redundancy, distributed systems solutions may provide for much higher availability with an acceptable performance trade-off.

In the following, the concepts of state-machine replication, process group communi-

cation, virtual synchrony, and distributed control are illustrated and existing solutions are detailed. The main focus is on the primary objective of this thesis to combine high availability efforts for HPC system head and service nodes with high availability efforts for distributed systems to provide the highest level of availability.

2.3.1 State-Machine Replication

The concept of *state-machine replication* [88, 89] has its origins in the early 1980s and is a common technique for providing fault tolerance in distributed systems. Assuming that a service is implemented as a deterministic finite state machine, *i.e.*, all service states are defined and all service state transitions are deterministic, consistent replication may be achieved by guaranteeing the same initial states and a linear history of state transitions for all service replicas. All correct service replicas perform the same state transitions and produce the same service output.

Service replicas are placed on different nodes. Service input needs to be consistently and reliably replicated to service replicas, while replicated service output needs to be consistently and reliably verified and unified. Voting on output correctness may be performed using a reliable quorum algorithm.

The introduction of the state-machine replication concept presented a number of problems for distributed systems with failures and sparked a greater research effort on process group communication, such as *reliable multicast*, *atomic multicast*, *distributed consensus*, and *failure detectors*.

2.3.2 Process Group Communication

In modern computing systems, services are implemented as communicating processes, *i.e.*, as a finite state machine that communicates to other components, such as clients and other services, via a network communication protocol. State-machine replication of a service involves communication among a set of replicated processes, *i.e.*, members of a replicated service group. Algorithms that coordinate the various operating modes of such a group are commonly referred to as *process group communication* algorithms. They typically deal with reliable delivery and consensus issues, such as *liveness*, *i.e.*, the process group eventually makes progress, and *consistency*, *i.e.*, all correct members of the process group eventually agree.

The propably most prominent process group communication algorithm is Lamport‘s Paxos [90] first proposed in 1990, which solves the distributed consensus problem in a network of unreliable processes. Most distributed consensus algorithms are based on the idea that all processes propose their output to either the entire group, a subset, or an

arbiter, to decide which process is correct.

There is a plethora of past research and development on process group communication algorithms and systems [91–94] focusing on semantics, correctness, efficiency, adaptability, and programming models. Group communication systems have been used to provide high availability for financial (stock market) and military applications, but not for head and service nodes in HPC systems.

Total Order Broadcast Algorithms

A total order broadcasting algorithm [93, 94] not only reliably delivers all messages within a process group, but also in the same order to all process group members. This fault-tolerant distributed consensus on message order is essential for state-machine replication and typically provided by a process group communication system.

The agreement on a total message order usually bears a cost of performance: a message is not delivered immediately after being received, until all process group members reach agreement on the total order of delivery. Generally, the cost is measured as latency from the point a new message is ready to be sent by a process group member, to the time it is delivered in total order at all process group members.

The following three approaches are widely used to implement total message ordering: sequencer, privilege-based, and communication history algorithms.

In sequencer total order broadcasting algorithms, one process group member is responsible for ordering and reliably broadcasting messages on behalf of all other process group members. A fail-over mechanism for the sequencer assures fault tolerance. The Amoeba distributed OS [95–98] utilises a sequencer algorithm as well as the Isis process group communication system (Section 2.3.3).

Privilege-based total order broadcasting algorithms rely on the idea that group members can reliably broadcast messages only when they are granted the privilege to do so. In the Totem protocol (Section 2.3.3) for example, a token is rotating among process group members and only the token holder can reliably broadcast messages. A time-out on the token assures fault tolerance and liveness.

In communication history total order broadcasting algorithms, messages can be reliably broadcast by any group member at any time, without prior enforced order, and total message order is ensured by delaying delivery until enough information of communication history has been gathered from other process group members. The Transis protocol (Section 2.3.3) is an example for a communication history algorithm.

These three types of algorithms have both advantages and disadvantages. Sequencer algorithms and privilege-based algorithms provide good performance when a system is relatively idle. However, when multiple process group members are active and constantly

broadcasting messages, the latency is limited by the time to circulate a token or produce total message order via a sequencer.

Communication history algorithms have a post-transmission delay to detect "happend before" relationships between incoming, reliably broadcast messages, *i.e.*, causal message order [99, 100]. The post-transmission delay typically depends on the slowest process group member and is most apparent when the system is relatively idle, since less communication history is produced and a response from all process group members is needed to determine total message order. In the worst case, the delay may be equal to the interval of heartbeat messages from an idle process group member. On the contrary, if all process group members produce messages and the communication in the process group is heavy, the regular messages continuously form a total order, and the algorithm provides the potential for low latency of total order message delivery.

Several studies have been made to reduce the cost. Early delivery algorithms [101, 102] are able to reduce latency by reaching agreement with a subset of the process group. Optimal delivery algorithms [103, 104] deliver messages before total message order is determined, but notify applications and cancel message delivery if the determined total message order is different from the delivered message order.

2.3.3 Virtual Synchrony

The *virtual synchrony* paradigm was first established in the early work on the Isis [105–107] group communication system. It defines the relation between regular-message passing in a process group and control-message passing provided by the system itself, *e.g.*, reports on process group joins or process failures. Process group membership is dynamic, *i.e.*, processes may join and leave the group. Whenever group membership changes, all the processes in the new membership observe a membership change event. Conceptually, the virtual synchrony paradigm guarantees that membership changes within a process group are observed in the same order by all the group members that remain connected. Moreover, membership changes are totally ordered with respect to all regular messages that pass in the system. The *extended virtual synchrony* paradigm [108], which additionally supports crash recoveries and network partitions, has been implemented in the Transis [109, 110] group communication system.

Isis

The Isis [105–107] system implements a collection of techniques for building software for distributed systems that performs well, is robust despite hardware and software failures, and exploits parallelism. The basic approach is to provide a toolkit mechanism for dis-

tributed programming, whereby a distributed system is built by interconnecting fairly conventional nondistributed programs, using tools drawn from the kit. Tools are included for managing replicated data, synchronising distributed computations, automating recovery, and dynamically reconfiguring a system to accommodate changing workloads. Isis has become very successful: companies and universities employed the toolkit in settings ranging from financial trading floors to telecommunications switching systems.

The Isis project has moved from Cornell University to Isis Distributed Systems a subsidiary of Stratus Computer, Inc. It was phased out by the end of 1998.

Horus

The Horus project [111–113] was originally launched as an effort to redesign Isis, but has evolved into a general-purpose communication architecture with advanced support for the development of robust distributed systems in settings for which Isis was unsuitable, such as applications that have special security or real-time requirements. Besides the practical uses of the software, the project has contributed towards the theory of virtual synchrony. At the same time, Horus is much faster and lightweight than the Isis system. Horus can be viewed as a group communication environment rather than as a collection of pre-built groupware solutions. It is UNIX-independent, and permits the use of several programming languages (C, C++, ML, and Python) in a single system. Horus protocols are structured like stacks of Lego-blocks, hence new protocols can be developed by adding new layers or by recombining existing ones. Through dynamic run-time layering, Horus permits an application to adapt protocols to its environment.

Horus can be used for research purposes at no fee, but has restricted commercial rights. Its stacked protocol architecture introduces additional overheads.

Ensemble

Ensemble [111, 114–116] is the next generation of the Horus group communication toolkit. It provides a library of protocols that can be used for quickly building complex distributed applications. An application registers 10 or so event handlers with Ensemble, and then the Ensemble protocols handle the details of reliably sending and receiving messages, transferring state, implementing security, detecting failures, and managing reconfigurations in the system. Ensemble is a highly modular and reconfigurable toolkit. The high-level protocols provided to applications are really stacks of tiny protocol layers. These protocol layers each implement several simple properties: they are composed to provide sets of high-level properties, such as total ordering, security, and virtual synchrony. Individual layers can be modified or rebuilt to experiment with new properties or change perfor-

mance characteristics. The software is partially written in OCaml permitting protocol verification and optimisation. However, the usage of OCaml limits portability.

Transis

Transis [109, 110, 117] is a multicast communication layer that facilitates the development of fault-tolerant distributed applications in a network of machines. It supports reliable group communication for high availability applications. Transis contains a novel protocol for reliable message delivery that optimises the performance for existing network hardware and tolerates network partitioning. Transis provides several forms of group multicast operations: FIFO ordered, causally ordered, totally ordered, and safely delivered. The multicast operations differ in their semantics and in their cost, *i.e.*, latency. The Transis approach to advanced group communication has acquired a wide recognition in the academic community, mainly due to the following desirable properties:

- It employs a highly efficient multicast protocol, based on the Trans protocol, that utilises available hardware multicast.
- It can sustain extremely high communication throughput due to its effective flow control mechanism, and its simple group design.
- It supports partitionable operation, and provides the means for consistently merging components upon recovery.

Totem

The Totem [118–120] system is a set of communication protocols to aid the construction of fault-tolerant distributed systems. The message ordering mechanisms provided by Totem allow an application to maintain consistency of distributed and replicated information in the presence of faults. The features of the Totem system are:

- ordered multicasts to process groups,
- high throughput and low predictable latency,
- highly portable code through the use of standard Unix features,
- rapid reconfiguration to remove failed processes, add recovered and new processes, and to merge a partitioned system, and
- continued operation of all parts of a partitioned system to the greatest possible extent that is consistent with correctness.

The Totem system provides reliable, totally ordered delivery of messages to processes within process groups on a single local-area network, or over multiple local-area networks interconnected by gateways. Superimposed on each local-area network is a logical token-passing ring. The fields of the circulating token provide reliable delivery and total ordering of messages, confirmation that messages have been received by all processors, flow control, and fault detection. Message ordering is consistent across the entire network, despite processor and communication faults, without requiring all processes to deliver all messages.

Totem is a group communication system with a fixed protocol layer. The more flexible Ensemble system provides a Totem protocol layer among others.

Spread

Spread [121, 122] is an open source toolkit that provides a high performance messaging service that is resilient to failures across local and wide area networks. Spread functions as a unified message bus for distributed applications, and provides highly tuned application-level multicast, group communication, and point-to-point support. Spread services range from reliable messaging to fully ordered messages with delivery guarantees. Spread can be used in many distributed applications that require high reliability, high performance, and robust communication among various subsets of members. The toolkit is designed to encapsulate the challenging aspects of asynchronous networks and enable the construction of reliable and scalable distributed applications. Spread consists of a library that user applications are linked with, a binary daemon which runs on each computer that is part of the process group, and various utility and demonstration programs.

While Spread supports multiple messaging protocols, such as Hop and Ring, and configurable message delivery semantics, like *safe* and *agreed*, it does not provide an open platform for group communication algorithms and programming models due to a missing modular architecture.

Object Group Pattern, Orbix+Isis, and Electra

The Object Group Pattern [123] offers programming model support for replicated objects using a group communication system for virtual synchrony. In this design pattern, objects are constructed as state machines and replicated using totally ordered and reliably multicast state changes. 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 efforts [124] focusing on high availability support for the Common Object Request Broker Architecture (CORBA).

2.3.4 Distributed Control

The *distributed control* paradigm emerged from the Harness project [125–127] I was involved in a few years ago. It focused on the design and development of a pluggable lightweight heterogeneous distributed virtual machine (DVM) environment, where clusters of personal computers, workstations, and "big iron" supercomputers can be aggregated to form one giant DVM in the spirit of its widely-used predecessor, Parallel Virtual Machine (PVM). As part of the Harness project, a variety of experiments and system prototypes were developed to explore lightweight pluggable frameworks, adaptive reconfigurable runtime environments, assembly of scientific applications from software modules, parallel plug-in paradigms, highly available DVMs [128–130], FT-MPI (Section 2.2.3), fine-grain security mechanisms, and heterogeneous reconfigurable communication frameworks. Three different Harness system prototypes were developed, two C variants and one Java-based alternative, each concentrating on different research issues.

In order to provide a fully symmetrically replicated DVM runtime environment containing its global state, such as current member processes and loaded software modules at members, the distributed control paradigm [128–130] offers a replicated remote procedure call (RPC) abstraction on top of a ring-based process group communication algorithm. The major difference to other process group communication systems is the utilisation of collective communication for the voting process on the RPC return. This allows the DVM to load a software module on a set of members in parallel and to recover appropriately if loading fails on a subset, such as to unload all successfully loaded software modules or to load software modules at alternate members. Due to the ring architecture, replicated RPC costs are linear with the number of members of a DVM.

2.3.5 Practical Byzantine Fault Tolerance

Recent research and development in providing service redundancy for high availability using group communication mechanisms focused on practical solutions for solving the *Byzantine generals problem* [131], where malicious attacks and software errors result in incorrect process group behaviour. These approaches go beyond the *fail-stop model*, which assumes that system components, such as services, nodes, or links, fail by simply stopping. Appropriate failure detection mechanisms are deployed to verify the correct behaviour of processes. This includes cryptographic techniques to prevent spoofing and replays, and to detect corrupted messages.

Byzantine fault tolerance mechanisms incur a higher performance overhead than fail-stop fault tolerance solutions during failure-free operation due to the additional measures employed to catch incorrect process behaviour. Since HPC is a performance sensitive

application area and typical HPC system failures exhibit fail-stop behaviour, Byzantine fault tolerance solutions are not employed.

Byzantine Fault Tolerance with Abstract Specification Encapsulation

Byzantine Fault Tolerance with Abstract Specification Encapsulation (BASE) [132] is a communication library for state-machine replication. It uses abstraction to reduce the cost of Byzantine fault tolerance and to improve its ability to mask software errors. Using BASE, each replica can be repaired periodically using an abstract view of the state stored by correct replicas. Furthermore, each replica can run distinct or nondeterministic service implementations, which reduces the probability of common mode failures. Prototype implementations for a networked file system [133] and an object-oriented database [134] suggest that the technique can be used in practice with a modest amount of adaptation and with comparable performance results.

Thema

Thema [135] is a middleware system on top of BASE that transparently extends Byzantine fault tolerance to Web Services technologies.

Zyzzyva

Zyzzyva [136] is a protocol that uses speculation to reduce the cost and simplify the design of Byzantine fault-tolerant state-machine replication. In Zyzzyva, replicas respond to a client‘s request without first running an expensive three-phase commit protocol to reach agreement on the order in which the request must be processed. Instead, they optimistically adopt the order proposed by the primary and respond immediately to the client. Replicas can thus become temporarily inconsistent with one another, but clients detect inconsistencies, help correct replicas converge on a single total ordering of requests, and only rely on responses that are consistent with this total order. This approach allows Zyzzyva to reduce replication overheads to near their theoretical minima.

Low-Overhead Byzantine Fault-Tolerant Storage

Related recent research [137] also produced an erasure-coded Byzantine fault-tolerant block storage protocol that is nearly as efficient as protocols that tolerate only crashes. Previous Byzantine fault-tolerant block storage protocols have either relied upon replication, which is inefficient for large blocks of data when tolerating multiple faults, or a combination of additional servers, extra computation, and versioned storage. To avoid these expensive techniques, this new protocol employs novel mechanisms to optimise for

the common case when faults and concurrency are rare. In the common case, a write operation completes in two rounds of communication and a read completes in one round. The protocol requires a short checksum comprised of cryptographic hashes and homomorphic fingerprints. It achieves throughput within 10% of the crash-tolerant protocol for writes and reads in failure-free runs when configured to tolerate up to 6 faulty servers and any number of faulty clients.

2.4 Information Technology and Telecommunication Industry Solutions

The IT and telecommunication industry has dealt with service-level high availability for some time. Individual solutions range from simple active/standby mechanisms to sophisticated enterprise products supporting a variety of replication strategies (Table 2.2). However, only very few efforts employ state-machine replication or virtual synchrony mechanisms in the form of active replication and dual modular redundancy (DMR) due to the complexity of communication protocols. In the following, individual existing solutions are examined and ongoing research and development efforts are described.

2.4.1 Hewlett-Packard Serviceguard

Hewlett-Packard (HP) Serviceguard [138] is a high availability solution for services that offers active/standby with cascading fail-over when using more than two service nodes, crosswise active/standby, and $n+1$ and $n+m$ high availability clustering configurations, all using a shared storage device (Sections 2.1.1 and 2.1.3). HP Serviceguard utilises a heartbeat mechanism over a network connection and needs a shared Small Computer System Interface (SCSI), Fibre Channel, or software mirrored storage subsystem with RAID capability. While the RAID capability provides data high availability within a single location, network-based storage mirroring allows robust data replication across the world. The HP Serviceguard Quorum Server [139] is able to arbitrate in case of service node partitions to ensure that there is only one active service group.

HP Serviceguard is one of the primary solutions for service high availability in the IT sector and part of HP‘s *non-stop* solutions [140] with support for HP-UX and Linux.

2.4.2 RSF-1

RSF-1 [141] from High-Availability.com is a service high availability product that provides active/standby with optional cascading fail-over, crosswise active/standby, and $n+1$

	Max. Service Nodes	Shared Storage	Storage Mirroring	Quorum Server	Active/Standby	Cascading Active/Standby	Crosswise Active/Standby	$n+1$ High Availability Clustering	$n+m$ High Availability Clustering	Active Replication	Dual Modular Redundancy	Linux	AIX	HP-UX	Solaris	Windows	MacOS	DRBD	SCSI	Fibre Channel	NAS	NFS
HP Serviceguard	?	×	×	×	×	×	×	×	×			×	×						×	×		
RSF-1	64	×			×	×	×	×	×			×	×	×	×		×					
IBM HAC	32	×	×	×	×	×	×	×	×				×						×	×	×	
Veritas Cluster Server	?	×	×	×	×	×	×	×	×			×	×	×	×	×			×	×	×	
Solaris Cluster	16	×	×	×	×	×	×	×	×						×				×	?	?	
Microsoft Cluster Server	8	×		×	×	?	×	?	?							×			×	×	?	
Red Hat Cluster Suite	128	×			×	×	×	×	×			×						?	×	×	×	×
LifeKeeper	32	×	×		×	×	×	×	×			×				×		?	×	×	×	
Linux FailSafe	16	×			×	×	×	×	×			×						?	×	×		
Linuxha.net	2	×	×		×		×					×						×	×	×	×	
Kimberlite	2	×			×		×					×						?	×	×		
T2CP-AR	2				×					×		×										
Stratus Cont. Processing	2									×	×	×				×						

×, Requirement or available feature
?, Insufficient documentation

Table 2.2: Requirements and features comparison between information technology and telecommunication industry service high availability products

and $n + m$ high availability clustering configurations, all using a shared storage device (Sections 2.1.1 and 2.1.3). It uses a heartbeat mechanism over a network connection and is based on a shared network attached storage (NAS), SCSI, or Fibre Channel storage subsystem with RAID capability for data high availability. RSF-1 scales up to 64 service nodes and supports Linux, HP-UX, Solaris, MacOS X (Darwin), and AIX.

RSF-1 is a primary service high availability IT solution. Customers include the University of Salford [142], the German WestLB Bank [143], and the China Postal Service [144].

2.4.3 IBM High Availability Cluster Multiprocessing

IBM High Availability Cluster Multiprocessing [145, 146] is a service high availability solution for up to 32 nodes running AIX that offers active/standby with optional cascading fail-over, crosswise active/standby, and $n + 1$ and $n + m$ high availability clustering configurations, all using a shared storage device (Sections 2.1.1 and 2.1.3). It relies on a heartbeat mechanism over a network connection or shared storage device, and requires a shared NAS, SCSI, or Fibre Channel storage subsystem with RAID capability for data high availability. While RAID technology is employed locally, a network-based storage mirroring mechanism is used for robust data replication across the world to assure disaster recovery. A quorum service for arbitrating service node partitions is supplied as well.

IBM High Availability Cluster Multiprocessing is one of the primary solutions for service high availability in the IT sector.

Note that IBM uses the term *cluster* to describe its configuration, which refers to high availability clustering (Section 2.1.3). IBM also offers cluster computing products for use in parallel and distributed computing scenarios.

2.4.4 Veritas Cluster Server

Symantec's Veritas Cluster Server [147, 148] is a high availability solution for services that offers active/standby with optional cascading fail-over, crosswise active/standby, and $n + 1$ and $n + m$ high availability clustering configurations, all using a shared storage device (Sections 2.1.1 and 2.1.3). It relies on a heartbeat mechanism over a network connection and requires a shared NAS, SCSI, or Fibre Channel storage subsystem with RAID capability for data high availability. Veritas Cluster Server supports Solaris, HP-UX, AIX, Linux, Windows. It offers a quorum service to arbitrate service node partitions as well as a data replication service for wide-area redundancy.

Symantec's Veritas Cluster Server is one of the primary solutions for service high availability in the IT sector.

Note that Symantec uses the term *cluster server* to describe its configuration, which refers to high availability clustering (Section 2.1.3). Veritas Cluster Server uses a process group communication system (Section 2.3.2) to determine *node membership*, *i.e.*, to detect service node failures and to notify its software components to perform fail-over. The product uses state-machine replication for its cluster configuration database, but not for service high availability.

2.4.5 Solaris Cluster

Sun's Solaris Cluster [149–151] is a high availability product for services that offers active/standby with optional cascading fail-over, crosswise active/standby, and $n + 1$ and $n + m$ high availability clustering configurations, all using a shared storage device (Sections 2.1.1 and 2.1.3). It relies on a heartbeat mechanism over a network connection and requires a shared SCSI storage subsystem with RAID capability for data high availability. The documentation is unclear about support for a NAS or Fibre Channel storage subsystem. Solaris Cluster runs on up to 16 Solaris service nodes and supports a quorum service to arbitrate service node partitions. Sun also offers a data replication in conjunction with its Metro Cluster and World Wide Cluster products for wide-area redundancy. Open High Availability Cluster (OHAC) [152] is the open-source code base of Solaris Cluster.

Note that Solaris uses the term *cluster* to describe its configuration, which refers to high availability clustering (Section 2.1.3).

2.4.6 Microsoft Cluster Server

Microsoft Cluster Server [153, 154] is a Windows-based service high availability variant that provides active/standby and crosswise active/standby, both using a shared storage device (Sections 2.1.1 and 2.1.3). The documentation is unclear about support for active/standby with optional cascading fail-over, and $n + 1$ and $n + m$ high availability clustering configurations. Microsoft Cluster Server uses a heartbeat mechanism over a network connection and requires a shared SCSI or Fibre Channel storage subsystem with RAID capability for data high availability. The documentation is also unclear about support for a NAS storage subsystem. The product supports up to 8 service nodes and a quorum service to arbitrate service node partitions.

Note that Microsoft uses the term *cluster server* to describe its configuration, which refers to high availability clustering (Section 2.1.3). Microsoft recently started offering cluster computing products for use in parallel and distributed computing scenarios.

2.4.7 Red Hat Cluster Suite

The Red Hat Cluster Suite [155, 156] is a service high availability product based on Red Hat Enterprise Linux [157] and offers active/standby with optional cascading fail-over, crosswise active/standby, and $n+1$ and $n+m$ high availability clustering configurations, all using a shared storage device (Sections 2.1.1 and 2.1.3). It relies on a heartbeat mechanism over a network connection and requires a shared network file system (NFS), NAS, SCSI, or Fibre Channel storage subsystem with RAID capability.

For high-volume open source applications, such as NFS, Samba, and Apache, Red Hat Cluster Suite provides complete ready-to-use solutions. For other applications, customers can create custom fail-over scripts using provided templates. Red Hat Professional Services can provide custom Red Hat Cluster Suite deployment services where required.

Note that Red Hat uses the term *cluster* to describe its configuration, which refers to high availability clustering (Section 2.1.3). Many 3^{rd} party vendors offer parallel and distributed cluster computing products based on Red Hat Enterprise Linux.

2.4.8 LifeKeeper

SteelEye's LifeKeeper [158] is a high availability product for services that offers active/standby with optional cascading fail-over, crosswise active/standby, and $n + 1$ high availability clustering configurations, all using a shared storage device (Sections 2.1.1 and 2.1.3). It relies on a heartbeat mechanism over a network connection and requires a shared NAS, SCSI, or Fibre Channel storage subsystem with RAID capability for data high availability. LifeKeeper runs on Linux and Windows. It supports up to 32 service nodes and offers a variety of recovery kits for services. SteelEye also offers a data replication product in conjunction with LifeKeeper for wide-area redundancy.

2.4.9 Linux FailSafe

Linux FailSafe by Silicon Graphics, Inc [159] is a service high availability solution that provides active/standby with optional cascading fail-over, crosswise active/standby, and $n + 1$ and $n + m$ high availability clustering configurations, all using a shared storage device (Sections 2.1.1 and 2.1.3). It uses a heartbeat mechanism over a serial and/or network connection and requires a shared SCSI or Fibre Channel storage subsystem with RAID capability for data high availability. Linux FailSafe is architected to scale up to 16 service nodes and offers plug-ins for service-specific interfaces. It is intended to be Linux distribution agnostic. However, there are dependencies on libraries and paths to system files that do vary from one distribution to the next.

Note that the Linux FailSafe documentation uses the term *cluster* to describe its configuration, which refers to high availability clustering (Section 2.1.3). It also uses the terms *process membership* and *process group*, different from the usage in virtual synchrony (Section 2.3.3). Linux FailSafe uses a process group communication system (Section 2.3.2 to determine *node membership*, *i.e.*, to detect service node failures and to notify its software components to perform fail-over. The product uses state-machine replication for its cluster configuration database, but not for service high availability.

2.4.10 Linuxha.net

Linuxha.net [160] is a high availability solution for services that features active/standby and crosswise active/standby limited for two service nodes, all using a shared storage device (Sections 2.1.1 and 2.1.3). It utilises a heartbeat mechanism over a network connection and a DRBD-based replicated storage subsystem or a NAS, SCSI, or Fibre Channel shared storage subsystem. In contrast to its competitors, an expensive NAS, SCSI, or Fibre Channel RAID is not necessarily required due to the network-based consistent data replication of DRBD.

Note that the Linuxha.net documentation uses the term *cluster* to describe its configuration, which refers to high availability clustering (Section 2.1.3). It also uses the terms *idle standby* for an active/standby configuration and *active standby* for the crosswise active/standby configuration.

2.4.11 Kimberlite

Kimberlite from Mission Critical Linux [161, 162] is a high availability variant that supports active/standby and crosswise active/standby redundancy for two service nodes using a shared storage device (Section 2.1.1). It utilises a heartbeat mechanism over a serial and/or network connection, and requires a shared SCSI or Fibre Channel storage subsystem with RAID capability for data high availability. Kimberlite runs on Linux. Since it is primarily composed of user-level daemons, Kimberlite is Linux distribution agnostic and runs on a great diversity of commodity hardware.

Note that the Kimberlite documentation uses the term *cluster* to describe its configuration and *cluster system* for each individual service node, both refer to high availability clustering (Section 2.1.3).

2.4.12 Transparent Transmission Control Protocol Active Replication

The Transmission Control Protocol (TCP) is one of the core protocols of the Internet. It provides reliable, in-order delivery of a stream of bytes. Transparent Transmission Control Protocol Active Replication (T2CP-AR) [163] provides replication support for TCP-based services by allowing backup services to intercept communication between clients and the primary service and by performing seamless connection fail-over.

The core concept of T2CP-AR relies on keeping consistent TCP-related connection state information at the backup service. T2CP-AR is a communication protocol for active/standby redundancy on two service nodes using active replication, *i.e.*, the backup service receives the same incoming traffic as the primary, performs all state transitions

in virtual synchrony, and takes over upon primary failure. The protocol was mainly developed for telecommunication services. A recently developed proof-of-concept prototype showcases a stateful fault-tolerant firewall (FT-FW) [164] in Linux.

2.4.13 Stratus Continuous Processing

Stratus [165] offers standard DMR, which uses two CPU-memory assemblies (motherboards). These systems deliver levels of availability unrivalled by competitive high availability cluster systems. DMR systems are designed for 99.999% of availability. All motherboards run in a lockstep manner from a single system clock source. The fault-detection and isolation logic compares I/O output from all motherboards; any miscompare indicates an error. DMR systems rely on fault-detection logic on each motherboard to determine which board is in error. If no motherboard error is signalled, a software algorithm determines which board to remove from service.

Stratus enables Continuous Processing capabilities through three fundamental elements: lockstep technology, failsafe software, and an active service architecture. While the lockstep technology uses replicated, fault-tolerant hardware components that process the same instructions at the same time, the failsafe software works in concert with lockstep technology to prevent many software errors from escalating into outages. The active service architecture enables built-in serviceability by providing remote Stratus support capabilities. The Continuous Processing technology is available in Stratus' ftServer product [166] for Red Hat Enterprise Linux 4 and Microsoft Windows Server 2003.

2.4.14 Open Application Interface Specification Standards Based Cluster Framework

The Open Application Interface Specification (OpenAIS) Standards Based Cluster Framework [167] is an Open Source implementation of the Service Availability Forum Application Interface Specification (AIS) [168]. AIS is a software application programming interface API and policies, which are used to develop applications that maintain service during faults. Restarting and failover of applications is also targeted for those deploying applications which may not be modified.

The OpenAIS software is designed to allow any third party to implement plug-in cluster services using the infrastructure provided. Among others, AIS includes APIs and policies for: naming, messaging, membership, event notification, time synchronisation, locking, logging, cluster management, and checkpointing. OpenAIS utilises process group communication semantics (Section 2.3.2) on compute nodes to deal with fault-tolerant node membership management, but not process membership management. Specifically,

the membership service relies on the virtual synchrony (Section 2.3.3) provided by the Totem process group communication system for node failure detection and triggering high availability cluster re-configuration.

OpenAIS is a pure development effort at this moment. It is based on existing mechanisms for service availability and fault tolerance. It supports Linux, Solaris, BSD, and MacOS X. OpenAIS does not support specific service high availability configurations. Instead, it is an effort to provide an open source solution for the underlying mechanisms used in service high availability.

2.4.15 System-level Virtualization

System-level virtualisation using hypervisors has been a research topic since the 1970s [169], but regained popularity during the past few years because of the availability of efficient solutions, such as Xen [170, 171], and the implementation of hardware support in commodity processors, *e.g.*, Intel-VT and AMD-V.

The IT industry has recently taken an interest in exploiting the benefits of system-level virtualisation in enterprise computing scenarios for server consolidation and for high availability. While server consolidation focuses on moving entire operating system environments of multiple underutilised servers to a single server, the high availability aspect similarly targets redundancy strategies for operating system environments across multiple server resources. While currently the IT and telecommunication industry typically deploys exactly the same high availability solutions in virtualised systems as in non-virtualised systems, recent research and development efforts focus on exploiting the additional features of system-level virtualisation, such as live migration and transparent checkpoint/restart, for providing high availability.

Xen

Xen [170, 171] is an open source hypervisor for IA-32, x86-64, IA-64, and PowerPC architectures. Its type-I system-level virtualisation allows one to run several virtual machines (VMs or guest operating systems) in an unprivileged domain (DomU) on top of the hypervisor on the same computer hardware at the same time using a host operating system running in a privileged domain (Dom0) for virtual machine management and hardware drivers. Several modified operating systems, such as FreeBSD, Linux, NetBSD, and Plan 9, may be employed as guest systems using paravirtualisation, *i.e.*, by modifying the guest operating system for adaptation to the hypervisor interface. Using hardware support for virtualisation in processors, such as Intel VT and AMD-V, the most recent release of Xen is able to run unmodified guest operating systems inside virtual machines.

Xen originated as a research project at the University of Cambridge, led by Ian Pratt, senior lecturer at Cambridge and founder of XenSource, Inc. This company now supports the development of the open source project and also sells enterprise versions of the software. The first public release of Xen was made available in 2003.

Xen supports live migration, *i.e.*, moving entire operating system environments from one physical resource to another while keeping it running. This capability is heavily used in enterprise computing scenarios for load balancing, scheduled maintenance, and proactive fault tolerance (Section 2.2.4).

VMware

VMware Inc. [172] offers a wide range of system-level virtualisation solutions, including the free VMware player and VMware Server (formerly VMware GSX Server). While the mentioned free products and the non-free VMware Workstation employ type-II virtualisation, *i.e.*, virtual machines are actual processes inside the host operating system, VMware Inc. also provides a non-free type-I system-level virtualisation solution, VMware ESX Server, based on hypervisor technology. The company further distributes an infrastructure solution (VMware Infrastructure) and various data-centre products for deployment of system-level virtualisation in enterprise businesses, such as for server consolidation.

VMware products support suspending and resuming running operating system environments, a transparent checkpoint/restart technique (Section 2.2.1). Using the suspend/resume facility, operating system environments can be migrated from one physical resource to another using the stop-and-copy approach. VMware products have also the notion of "virtual appliances", where a service is deployed together with an operating system environment certified for this specific service.

High Availability in a Virtualised Environment

High Availability in a Virtualised Environment (HAVEN) [173] is a classification approach for high availability configurations in virtualised systems based on operational virtual machine states and the respective life cycle. This classification encompasses several checkpoint/restart schemes, including virtual machine states and state transitions, based on the suspend/resume facility of system-level virtualisation solutions.

It is important to note that the IT and telecommunication industry typically deploys exactly the same high availability solutions in virtualised systems as in non-virtualised systems at the service-level, *i.e.*, service-level replication protocols are used. This is primarily due to the fact that system-level replication protocols, such as replicating an entire virtual machine, have to deal with certain issues regarding possible non-deterministic be-

haviour of the hypervisor and the operating system sitting on top of it. Replayability, *i.e.*, feeding a replica with a sequence of input messages to archive a specific state, is often a requirement of active/hot-standby solutions and a must for any sort of state-machine replication variant, such as dual-modular redundancy.

2.5 Summary

This Chapter evaluated previous work within the research context of this thesis. Detailed past and ongoing research and development for HPC head and service node high availability included active/standby configurations using shared storage, active/standby configurations using software state replication, and high availability clustering. Certain pitfalls involving active/standby configurations using shared storage, such as backup corruption during failure, have been pointed out. There was no existing solution for HPC head and service node high availability using symmetric active/active replication, *i.e.*, state-machine replication.

Described techniques for HPC compute node high availability included checkpoint/restart, message logging, algorithm-based fault tolerance, and proactive fault avoidance. The underlying software layer often relied on HPC head and service node high availability for coordination and reconfiguration after a failure.

Examined distributed systems efforts focused on state-machine replication, process group communication, virtual synchrony, distributed control, and practical Byzantine fault tolerance. Many of the distributed systems mechanisms were quite advanced in terms of communication protocol correctness and provided availability. There has been much less emphasis on high performance.

Detailed IT and telecommunication industry solutions covered a wide range of high availability configurations. The only two solutions using some variant of state-machine replication were T2CP-AR and Stratus Continuous Processing, where T2CP-AR targets TCP-based telecommunication services and Stratus Continuous Processing offers DMR with hardware-supported instruction-level replication.

3 Taxonomy, Architecture, and Methods

This Chapter lays the theoretical ground work of this thesis by defining a modern service-level high availability taxonomy, describing existing availability deficiencies in HPC system architectures using a generalised model, and detailing methods for providing high availability for services running on HPC system head and service nodes.

3.1 High 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 [174, 175]. As part of this research effort, it has been adopted to the complexity of HPC system architectures, enhanced with more precise definitions for active/standby replication, and extended with definitions for active/active replication.

For completeness, the text of Section 3.1.1 has been copied from [174] with minor corrections and added definitions for soft errors and hard errors. Section 3.1.3 also contains greater parts from [174], while it extends on the original definitions for active/standby replication, adds definitions for active/active replication, and omits the terms passive and active replication.

3.1.1 Faults, Failures, and Outages

Conceptually, a *fault* can be described as an unexpected behaviour 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 pre-empt or react to faults it already expects and can detect, *i.e.*, faults that are visible

either directly as *abnormal* system behaviour 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 behaviour. 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.
- *Pre-emptive, 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.
- *Other causes* for failures exist, such as race conditions between processes, distributed deadlocks, and network transmission errors.
- Further causes are *catastrophic events*, such as earthquake, 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* (Section 2.3.5) [131] 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* (or *downtime*) is used to describe any kind of deviation from specified system behaviour, whether it is expected or unexpected. All faults and failures can

be categorised as *unplanned outages*, while intentional prevention of delivering specified system functionality, such as to perform maintenance operations, software upgrades, etc., are *planned outages*.

3.1.2 Reliability Metrics

Reliability is the property of a component that defines its probability to perform its intended function during a specified period of time, *i.e.*, it provides information about the failure-free time period. In contrast to *availability* (Sections 3.1.3 and 3.1.4), reliability does not provide information about failure recovery, *i.e.*, how well a component or system deals with failures. Reliability metrics, such as *failure rate* λ and $MTTF$, are essential aspects of availability metrics (Section 3.1.4).

The failure rate λ is the frequency at which an individual component experiences faults. Given a large enough population of identical components, a failure rate for this population can be observed that displays the "bathtub curve" (Figure 3.1) [175]. While the initial usage period is characterised by a high but rapidly decreasing failure rate (infant mortality phase), a roughly constant failure rate can be observed for a prolonged period of time. Finally, the failure rate begins to increase again (wear-out phase).

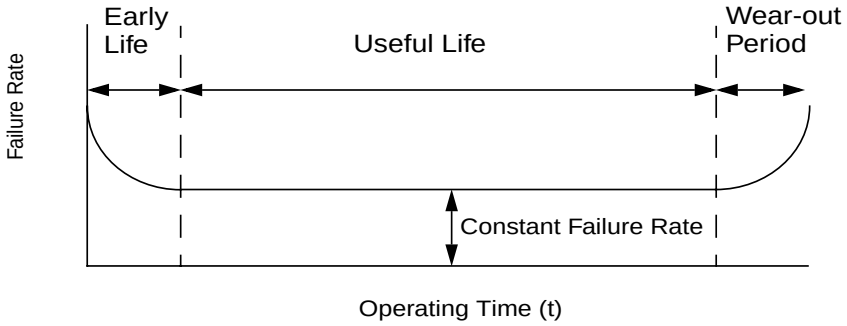


Figure 3.1: The "bathtub curve": Failure rate for a given large enough population of identical components [175]

Manufacturers often subject components to a "burn-in" phase before delivery to provide customers with products that have passed their infant mortality phase. Customers requiring high availability also typically replace components before the wear-out phase begins using manufacturer recommendations on component lifetime. Both are very common procedures in the IT and telecommunication industry as well as in HPC. High availability solutions mostly need to deal with the prolonged period of time where a constant failure rate can be observed.

The reliability function $R(t)$ of an individual component is the probability that it will perform its intended function during a specified period of time $0 \leq \tau \leq t$ (Equation 3.1),

while its *failure distribution* function $F(t)$ is the probability that it will fail during a specified period of time $0 \leq \tau \leq t$ (Equation 3.2) [176].

$$R(t) = \int_t^\infty f(\tau)d\tau, \quad t \leq \tau \leq \infty \tag{3.1}$$

$$F(t) = \int_0^t f(\tau)d\tau, \quad 0 \leq \tau \leq t \tag{3.2}$$

The *probability distribution* of both functions is defined by $f(\tau)$, a *probability density function* (PDF). The reliability function $R(t)$ also defines a component's failure rate function $\lambda(t)$ (Equation 3.1), and its $MTTF$ (Equation 3.4) [176].

$$\lambda(t) = \frac{f(t)}{R(t)} = \frac{f(t)}{\int_t^\infty f(\tau)d\tau}, \quad 0 \leq \tau \leq t \tag{3.3}$$

$$MTTF = \int_0^\infty R(t)dt \tag{3.4}$$

A normalized exponential PDF of $\lambda e^{-\lambda t}$ with a constant failure rate λ is typically assumed for the prolonged centre period of the "bathtub curve", which is not an exact match, but close to empirical failure data. All these functions can be simplified to the following terms [176]:

$$R(t) = e^{-\lambda t} \tag{3.5}$$

$$F(t) = 1 - e^{-\lambda t} \tag{3.6}$$

$$\lambda(t) = \lambda \tag{3.7}$$

$$MTTF = \frac{1}{\lambda} \tag{3.8}$$

Individual components within a system may directly depend on each other, resulting in a series subsystem. Individual components also may directly provide redundancy for each other in form of a parallel subsystem. The reliability function $R(t)$ of a n-series or n-parallel subsystem is as follows [176]:

$$R(t)_{series} = \prod_{i=1}^{n} R_i(t) \tag{3.9}$$

$$R(t)_{parallel} = 1 - \prod_{i=1}^{n}(1 - R_i(t)) \tag{3.10}$$

Assuming an exponential PDF for component i of $\lambda e^{-\lambda_i t}$ with constant individual component failure rates λ_i, these functions can be simplified to the following terms [176]:

$$R(t)_{series} = e^{-\lambda_s t}, \quad \lambda_s = \sum_{i=1}^{n} \lambda_i \tag{3.11}$$

$$R(t)_{parallel} = 1 - \prod_{i=1}^{n}(1 - e^{-\lambda_i t}) \tag{3.12}$$

The failure rate of a series subsystem is constant as the entire system stops functioning upon a single failure. The failure rate of a parallel subsystem changes with each failure of a redundant component as the system continues functioning. The $MTTF$ of a n-series subsystem with individual constant component failure rates λ_i and of a n-parallel subsystem with an equal constant individual component failure rate λ is [176]:

$$MTTF_{series} = \frac{1}{\lambda_s}, \quad \lambda_s = \sum_{i=1}^{n}\lambda_i = \sum_{i=1}^{n}\frac{1}{MTTF_i} \tag{3.13}$$

$$MTTF_{parallel} = \frac{1}{\lambda}\sum_{i=1}^{n}\frac{1}{i} = MTTF_{component}\sum_{i=1}^{n}\frac{1}{i} \tag{3.14}$$

While $MTTF$ decreases with every component in a series subsystem due to dependency, it increases with every component in a parallel subsystem due to redundancy. The $MTTF$ in a series subsystem can be close to 0 in a worst-case scenario, while it can be close to $2 \cdot MTTF_{component}$ in the best-case scenario for a parallel subsystem.

Note that these reliability metrics do not include any information about how well a component or system deals with failures. A system's $MTTF$ can be very low, yet its availability can be very high due to efficient recovery.

3.1.3 Availability Domains and Configurations

Availability is a property that provides information about how well a component or system deals with outages. It can be generalised into the following three distinctive domains.

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.

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 further 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, *e.g.*, military, financial, healthcare, 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 behaviour, masking a fault means that no deviations (or only precisely defined deviations) from specified behaviour 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. 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, 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. Asymmetric active/active provides high availability with improved throughput performance. While it is heavily used in the telecommunication sector for stateless components ($n + 1$ *and* $n + m$ *high availability clustering*, Sections 2.1.3 and 2.4), 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 maintain a common global component state using *state-machine replication* (Section 2.3.1), *virtual synchrony* (Section 2.3.3), or *distributed control* (Section 2.3.4), *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.

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, *e.g.*, in the military, financial, healthcare, and telecommunication sectors. Employed technologies range from hot standby with transparent fail-over and multiple standbys to active/active.

3.1.4 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}:

$$A = \frac{t_{up}}{t_{up} + t_{down}} \tag{3.15}$$

Availability can be calculated (Equation 3.16) based on a systems $MTTF$ and $MTTR$. 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 $MTBF$, which is the average interval of time in which any failure occurs again. A systems $MTBF$ covers both, $MTTF$ and $MTTR$ (Equation 3.17). The estimated annual unplanned downtime of a system in terms of hours t_{down} can be calculated using its availability (Equation 3.18). Planned outages may be included by respectively adjusting $MTTF$ and $MTTR$.

$$A = \frac{MTTF}{MTTF + MTTR} = \frac{1}{1 + \frac{MTTR}{MTTF}} \tag{3.16}$$

$$MTBF = MTTF + MTTR \tag{3.17}$$

$$t_{down} = 365 \cdot 24 \cdot (1 - A) \tag{3.18}$$

A system is often rated by the number of 9s in its availability figure (Table 3.1). For example, a system that has a five-nines availability rating, has 99.999% availability and an annual downtime of 5 minutes and 15.4 seconds. This rating is commonly used for mission critical systems, *e.g.*, military, financial, healthcare, and telecommunication.

9s	Availability	Annual Downtime
1	90%	36 days, 12 hours
2	99%	87 hours, 36 minutes
3	99.9%	8 hours, 45.6 minutes
4	99.99%	52 minutes, 33.6 seconds
5	99.999%	5 minutes, 15.4 seconds
6	99.9999%	31.5 seconds

Table 3.1: Availability measured by the "nines"

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 *entirely* redundant to component 4. The availability of n-series or n-parallel component compositions is as follows:

$$A_{series} = \prod_{i=1}^{n} A_i \tag{3.19}$$

$$A_{parallel} = 1 - \prod_{i=1}^{n}(1 - A_i) \tag{3.20}$$

The availability of n-series or n-parallel component compositions with equal individual component availability $A_{component}$ is as follows:

$$A_{series} = A_{component}^{n} \tag{3.21}$$

$$A_{parallel} = 1 - (1 - A_{component})^{n} \tag{3.22}$$

The more dependent, *i.e.*, serial, components a system has, the less availability it provides. The more redundant, *i.e.*, parallel, components a system has, the more availability it offers. High availability systems are build upon adding redundant components to increase overall system availability. Automatic redundancy solutions, such as active/standby (Section 3.1.3), employ parallel subsystems with equal component failure rates (Section 3.1.2) in series with a fault detection and reconfiguration *software subsystem* residing on one or more components.

Assuming an equal individual component $MTTF_{component}$ based on observed failure rate, an equal individual component $MTTR_{component}$ based on the time it takes to manually replace a component, and an equal individual component $MTTR_{recovery}$ based on the time it takes to automatically recover from a component failure, the availability $A_{redundancy}$ of an automatic n-redundancy solution can be calculated based on a parallel component composition for the n redundant components in series with the fault detection and reconfiguration software subsystem residing on m components as follows:

$$A_{redundancy} = [1 - (1 - A_{component})^{n}]A_{reconfiguration} \tag{3.23}$$

$$A_{component} = \frac{MTTF_{component}}{MTTF_{component} + MTTR_{component}} = \frac{1}{1 + \frac{MTTR_{component}}{MTTF_{component}}} \tag{3.24}$$

$$A_{reconfiguration} = \frac{\frac{1}{m}MTTF_{component}}{\frac{1}{m}MTTF_{component} + MTTR_{recovery}} = \frac{1}{1 + m\frac{MTTR_{recovery}}{MTTF_{component}}} \tag{3.25}$$

The distinction between n and m is a result of the fact that the $MTTF$ of the software subsystem depends on the series component composition (Equation 3.13) for which $MTTR_{recovery}$ is observed. $m = 1$ for active/warm-standby, while $m = n$ for active/hot-standby and symmetric active/active due to consistency requirements. $MTTR_{recovery}$ is the primary quality of service metric for a high availability method with an automatic recovery procedure. The faster recovery takes place, the higher availability is provided. Efficient redundancy strategies focus on a low $MTTR_{recovery}$. They also target a low replication overhead during failure-free operation as a secondary quality of service metric.

3.1.5 Fail-Stop

The *failure model* is an important aspect of high availability as it defines the scope of failures that are masked.

The *fail-stop* model assumes that system components, such as individual services, nodes, and communication links, fail by simply stopping. Employed failure detection and recovery mechanisms only work correctly in case of hard errors or catastrophic soft errors, *i.e.*, benign failures (Section 3.1.1). 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 work presented in this thesis is entirely based on the fail-stop model, since HPC system nodes typically exhibit this behaviour. Furthermore, HPC is a performance sensitive application area and *Byzantine* fault tolerance mechanisms (Section 2.3.5) that handle all kinds of failures, including malicious failures and non-catastrophic soft errors (Section 3.1.1), incur a higher performance overhead during normal operation due to the extra measures employed to catch incorrect process behaviour. Fail-stop behaviour can also be enforced to a certain degree by immediately fencing off failed or incorrect system components (Section 2.1.4) to avoid further disruption.

3.2 System Architecture

In the following, the design and properties of current HPC system architectures are examined and generalised. In this context, availability deficiencies are identified and classified.

3.2.1 HPC System Architectures

The emergence of cluster computing in the late 90's made scientific computing not only affordable to everyone using commercial off-the-shelf (COTS) hardware, it also introduced the Beowulf cluster system architecture (Figure 3.2) [177–180] with its single head node controlling a set of dedicated compute nodes. In this architecture, head node, compute nodes, and interconnects can be customised to their specific purpose in order to improve efficiency, scalability, and reliability. Due to its simplicity and flexibility, many supercomputing vendors adopted the Beowulf architecture either completely in the form of HPC Beowulf clusters or in part by developing hybrid HPC solutions.

Most architectures of today's HPC systems have been influenced by the Beowulf cluster system architecture. While they are designed based on fundamentally different system architectures, such as vector, massively parallel processing (MPP), and single-system

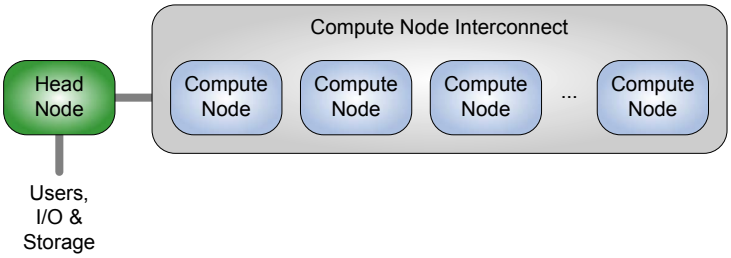


Figure 3.2: Traditional Beowulf cluster system architecture

image (SSI), the Beowulf cluster computing trend has led to a generalised architecture for HPC systems.

In this generalised HPC system architecture (Figure 3.3), a number of compute nodes perform the actual parallel computation, while a head node controls the system and acts as a gateway to users and external resources. Optional service nodes may offload specific head node responsibilities in order to improve performance and scalability. For further improvement, the set of compute nodes may be partitioned (Figure 3.4), tying individual service nodes to specific compute node partitions. However, a system's architectural footprint is still defined by its compute node hardware and software configuration as well as the compute node interconnect.

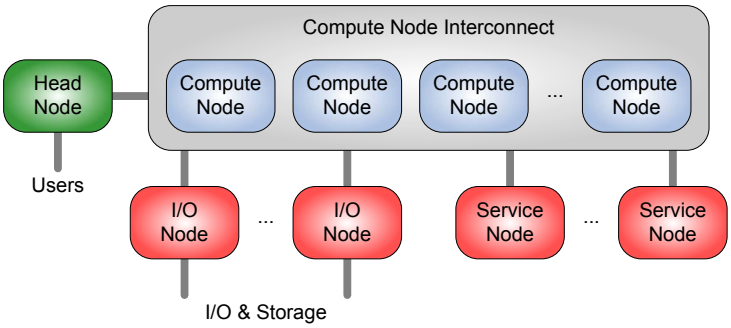


Figure 3.3: Generic modern HPC system architecture

System software, such as OS and middleware, has been influenced by this trend as well, but also by the need for customisation and performance improvement. Similar to the Beowulf cluster system architecture, system-wide management and gateway services are provided by head and service nodes.

However, in contrast to the original Beowulf cluster system architecture with its "fat" compute nodes running a full OS and a number of middleware services, today's HPC systems typically employ "lean" compute nodes (Figure 3.5) with a basic OS and only a small amount of middleware services, if any middleware at all. Certain OS parts and middleware services are provided by service nodes instead.

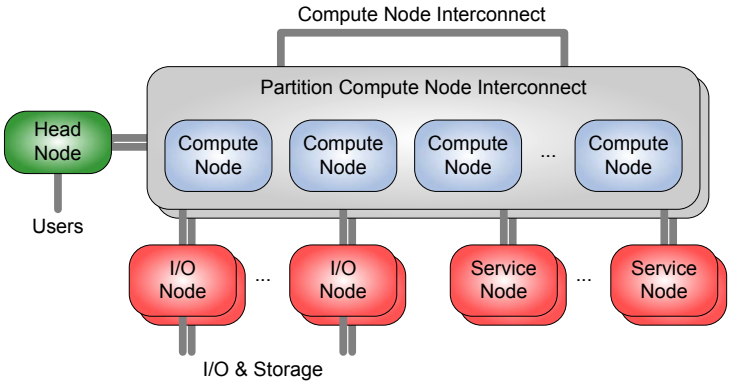
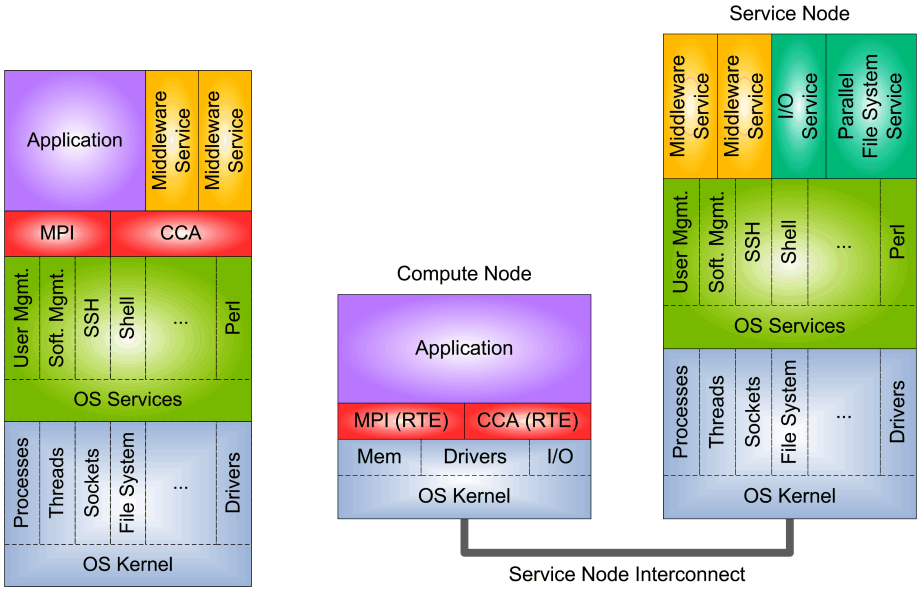


Figure 3.4: Generic modern HPC system architecture with compute node partitions



(a) Fat (b) Lean

Figure 3.5: Traditional fat vs. modern lean compute node software architecture

The following paragraph is an overview description from the Cray XT4 documentation [181] that illustrates this recent trend in HPC system architectures:

The XT4 is the current flagship MPP system of Cray. Its design builds upon a single processor node, or processing element (PE). Each PE is comprised of one AMD microprocessor (single, dual, or quad core) coupled with its own memory (1-8 GB) and dedicated communication resource. The system incorporates two types of processing elements: compute PEs and service PEs. Compute PEs run a lightweight OS kernel, Catamount [182–184], that is optimised for application performance. Service PEs run standard SUSE Linux [185] and can be configured for I/O, login, network, or system functions. The I/O system uses the highly scalable Lustre [41–43] parallel file system. Each compute

blade includes four compute PEs for high scalability in a small footprint. Service blades include two service PEs and provide direct I/O connectivity. Each processor is directly connected to the interconnect via its Cray SeaStar2 routing and communications chip over a 6.4 GB/s HyperTransport path. The router in the Cray SeaStar2 chip provides six high-bandwidth, low-latency network links to connect to six neighbours in the 3D torus topology. The Cray XT4 hardware and software architecture is designed to scale steadily from 200 to 120,000 processor cores.

The Cray XT4 system architecture with its lean compute nodes is not an isolated case. For example, the IBM Blue Gene [33, 34] solution also uses a lightweight compute node OS together with service nodes. In fact, the Compute Node Kernel (CNK) on the IBM Blue Gene forwards most Portable Operating System Interface (POSIX) [186] calls to the service node for execution using a lightweight RPC [187].

System software solutions for modern HPC architectures, as exemplified by the Cray XT4 and the IBM Blue Gene, need to deal with certain architectural limitations. For example, the compute node OS of the Cray XT4, Catamount, is a non-POSIX lightweight OS, *i.e.*, it does not provide multiprocessing, sockets, and other POSIX features. Furthermore, compute nodes do not have direct attached storage (DAS), instead they access networked file system solutions via I/O service nodes.

3.2.2 Availability Deficiencies

Due to the fact that a HPC system depends on each of its nodes and on the network to function properly, *i.e.*, all HPC system components are interdependent (*serially coupled*, Section 3.1.4), single node or network failures trigger system-wide failures. The availability of a HPC system may be calculated (Equation 3.26) using the availability of its individual components, such as of its nodes n, network links l, and network routers r. However, this model is simplistic as it does not include any external dependencies, such as electrical power, thermal cooling, and network access.

$$A = \prod_{n=1}^{N} A_n \prod_{l=1}^{L} A_l \prod_{r=1}^{R} A_r \tag{3.26}$$

Individual availability deficiencies, *i.e.*, deficient components, of any type of system can be categorised by their system-wide impact into: *single points of failure* and *single points of control*. A failure at a single point of failure interrupts the entire system. However, the system is able to continue to partially function properly after reconfiguration (recovery) into a *degraded operating mode*. Such reconfiguration may involve a full or partial restart of the system. A failure at a single point of control interrupts the entire system and

additionally renders the system useless until the failure has been repaired.

Partial or complete loss of state may occur in case of any failure. For HPC systems, there is a distinction between *system state* and *application state*. While system state consists of the states of all system services including related OS state, application state comprises of the process states of a parallel application running on a HPC system, including dependent system state, such as communication buffers (in-flight messages).

In the following, individual single points of failure and single points of control of HPC systems are described. The notion of critical system services is introduced and their impact on system availability is discussed. The role of individual HPC system nodes is examined in more detail

Critical System Services

HPC systems run *critical* and *non-critical system services* on head, service, and compute nodes, such as job and resource management, communication services, and file system metadata services, permitting applications to perform computational jobs in parallel on the compute nodes using the interconnect for messaging.

A service is *critical* to its system if it can't operate without it. Any such critical system service is a single point of failure and control for the entire HPC system. As long as one of them is down, the entire HPC system is not available. Critical system services 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. Dependent 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 HPC system.

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

A *system partition service* is critical to its system partition if it can't operate without it. Any such service is a single point of failure and control for the respective partition it belongs to. If the system is not capable of operating in a degraded mode, any such critical system partition service is also a critical system service. However, if the system is capable of operating in a degraded mode, any such critical system partition service is also a non-critical system service.

Typical critical HPC system services are: user login, network file system (I/O forward-

ing, metadata, and storage), job and resource management, communication services, and in some cases the OS or parts of the OS itself, *e.g.*, for SSI systems. User management, software management, and programming environment are usually non-critical system services, while network file system I/O forwarding and communication services are typical critical system partition services.

Head Node

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 HPC system. As long as it is down, the entire HPC 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 metadata and storage. 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, *e.g.*, IBM MareNostrum [188], vector machines, *e.g.*, Cray X1 [189], MPP systems, *e.g.*, Cray XT4 [181], and SSI solutions, *e.g.*, SGI Altix [190].

Service Nodes

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 HPC system. As long as one of them is down, the entire HPC 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 [189], Cray XT4 [181], IBM Blue Gene [33, 34], and IBM MareNostrum [188].

Partition Service Nodes

If a system has *partition service nodes running critical system partition services*, any such partition service node is a single point of failure and control for the respective HPC system partition it belongs to. As long as any such partition service node is down, the respective HPC system partition of the system is not available. Similar to a service node failure,

a partition service node failure may cause a loss of system and application state. If the system is not capable of operating in a degraded mode, any such partition service node is a single point of failure and control for the entire HPC system. As long as any one of them is down, the entire HPC system is not available.

Partition service nodes typically offload critical head/service node system services, but not non-critical system services. They can be found in more advanced large-scale cluster and MPP systems, *e.g.*, Cray XT4 [181] and IBM Blue Gene [33, 34]. Furthermore, federated cluster solutions use head nodes of individual clusters as partition service nodes, *e.g.*, NASAs SGI Altix Supercluster Columbia [191].

Compute Nodes

Each *compute node running critical system services* is a single point of failure and control for the entire HPC system. As long as any such compute node is down, the entire HPC system is not available. A failure of a compute node running critical system services may cause a loss of system and application state. Each *compute node not running critical system services* is still a single point of failure for the entire HPC system. If the system is not capable of operating in a degraded mode, any such compute node is a single point of failure and control for the entire HPC system. As long as one of them is down, the entire HPC system is not available. A failure of a compute node not running critical system services may cause a loss of application state, but not a loss of system state.

Compute nodes that do run critical system services may be found in some HPC systems. Sometimes communication services and in some cases the OS are critical system services running on compute nodes, *e.g.*, in SSI systems, like SGI's Altix [190]. Compute nodes that do not run critical system services can be found in most HPC systems, similar to partition compute nodes that do not run critical system services (following Section).

Partition Compute Nodes

Each *partition compute node running critical system partition services* is a single point of failure and control for the respective HPC system partition it belongs to. As long as any such partition compute node is down, the respective partition of the HPC system is not available. Similar to a failure of a compute node, a failure of a partition compute node running critical system services may cause a loss of system and application state. If the system is not capable of operating in a degraded mode, any partition compute node is a single point of failure and control for the entire HPC system. As long as any one of them is down, the entire HPC system is not available.

Each *partition compute node not running critical system services* is still a single point of

failure for the respective partition it belongs to. If the system is not capable of operating in a degraded mode, any such compute node is a single point of failure and control for the entire system. A failure of a partition compute node not running critical system services may cause a loss of application state, but not a loss of system state.

Partition compute nodes may run the same critical system services that run on normal compute nodes. Partition compute nodes that do run critical system services can be found in federated SSI solutions, *e.g.*, NASAs SGI Altix Supercluster Columbia [191], where each partition is a SSI system. Partition compute nodes that do not run critical system services can be found in more advanced large-scale MPP systems, *e.g.*, Cray XT4 [181] and IBM Blue Gene [33, 34].

System Scale

Theory (Equation 3.26) and recent empirical studies [2, 3] show that the availability of a system decreases proportionally with its number of dependent (serial) components. Additionally, the MTTR of a system grows with the number of its components, if the recovery involves reconfiguration of the entire system and not just of the single failed component.

The more nodes a HPC system consists of, the more frequent is the occurrence of failures and the more time is needed for a system-wide recovery, such as restart from a checkpoint (Section 2.2.1). Moreover, the total number of nodes in a HPC system also negatively influences the efficiency of recovery strategies during normal operation. Checkpointing a HPC system needs to be counted as planned outage and will take longer the more nodes are involved.

Section 1.2 already referred to two studies [20, 21] performed at Los Alamos National Laboratory that estimate MTBF and recovery overhead on next-generation large-scale HPC systems. Extrapolating from current HPC system performance, scale, and overall system MTTF, the first study suggests that the MTBF will fall to only 1.25 hours of useful computation on a 1 PFlop/s next-generation supercomputer. It also estimates that more than 60% of the processing power (and investment) may be lost due to the overhead of dealing with reliability issues. The second study estimates a MTTF of 1.5 hours for non-recoverable radiation-induced soft errors (double-bit memory or single-bit logic error) on a Cray XD1 system [22] with the same number of processors as the ASCI Q system [23], with almost 18,000 FPGAs and 16,500 GB of ECC memory.

3.3 High Availability Methods

Previous work on providing high availability for HPC head and service nodes (Section 2.1) relies on service-level solutions and customised replication environments, resulting in insufficient reuse of code. This causes not only rather extensive development efforts for each new high availability method and service, but also makes their correctness validation an unnecessary repetitive task. Furthermore, replication techniques can't be seamlessly interchanged for a specific service in order to find the most appropriate solution based on quality of service requirements.

In the following, a more generic approach toward service-level high availability for HPC head and service nodes is presented to alleviate and eventually eliminate these issues. In a first step, a conceptual service model is defined. In a second step, various service-level high availability methods and their properties are described based on this model. In a third step, individual service-level high availability methods are compared with each other with regards to their performance overhead and provided availability. Similarities and differences in their programming interfaces are examined as well.

3.3.1 Service Model

A more generic approach toward service-level high availability relies on the definition of a service as a communicating process (Figure 3.6), which interacts with other local or remote clients, services, and users via an input/output interface, such as network connection(s), command line interface(es), and/or other forms of inter-process 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 clients, services, and users in a request/response fashion, or to other clients, services, and users in a trigger/forward fashion.

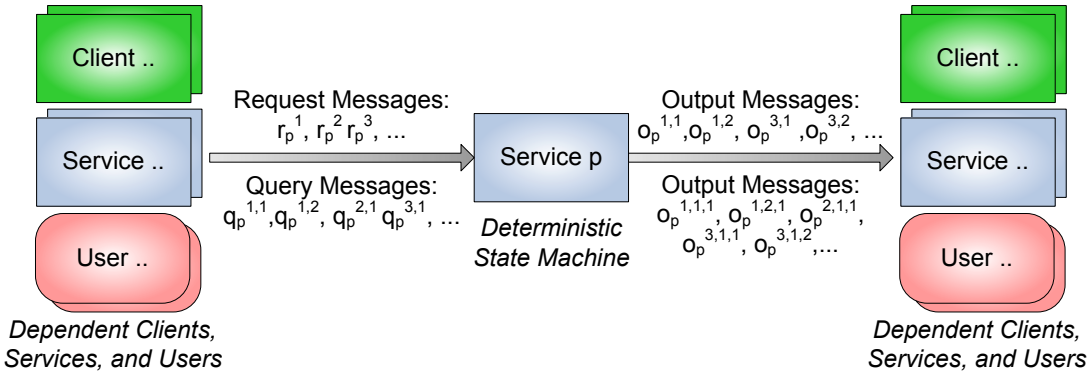


Figure 3.6: Generic service model

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 service behaviour may be caused by non-deterministic service input, such as by a system timer sending input messages signalling a specific timeout or time. If a stateless or stateful service relies on such a non-deterministic component invisible to the outside world it is considered non-deterministic. Examples for sources of non-determinism are:

- *unsynchronised* random number generators,
- *unsynchronised* timers,
- *uncoordinated* multiplexed I/O, *e.g.*, `select` with multiple file descriptors, and
- *uncoordinated* multithreading, *e.g.*, uncoordinated `read`/`write` in multiple threads.

Non-deterministic service behaviour has serious consequences for service-level replication techniques for stateful services as deterministic replay using the same input messages can't be guaranteed, *i.e.*, using the same input messages two identical services may reach different state and/or may produce different output.

Sources of non-determinism may be eliminated by forcing determinism onto them, such as by synchronising timers and random number generators across nodes or by coordinating I/O accesses and threads within a service.

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

The state of service p at step t is defined as S_p^t, and its initial state as S_p^0. A state transition from S_p^{t-1} to S_p^t is triggered by request message r_p^t 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 processing the x^{th} query message $q_p^{t,x}$ by service p at step t. Multiple 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 ..., $o_p^{t,x,y-1}$, $o_p^{t,x,y}$.

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 needed for service state replication.

The failure mode of a service is fail-stop (Section 3.1.4), *i.e.*, the service itself, its node, or its communication links, fail by simply stopping. Appropriate failure detection mechanisms may be deployed to assure fail-stop behaviour in certain cases, such as to detect incomplete, garbled, or otherwise erroneous messages.

The availability $A_{component}$ of a service can be calculated based on its $MTTF_{component}$ and $MTTR_{component}$ (Equation 3.16 in Section 3.1.4). The availability $A_{redundancy}$ of a service redundancy solution can be calculated based on $MTTF_{component}$, $MTTR_{component}$, $MTTR_{recovery}$, n, and m (Equation 3.23 in Section 3.1.4). $MTTR_{recovery}$, *i.e.*, the time it takes to automatically recover from a component failure, is the primary quality of service metric. The goal of the following service replication mechanisms is to provide for service redundancy with a low $MTTR_{recovery}$. The availability of a service approaches 1, *i.e.*, 100%, if $MTTR_{recovery}$ approaches 0. A low replication overhead during failure-free operation is a secondary quality of service metric.

3.3.2 Active/Standby Replication

In the active/standby replication method 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 (node fencing, Section 2.1.4).

Service-level active/standby replication (Figure 3.7) 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.

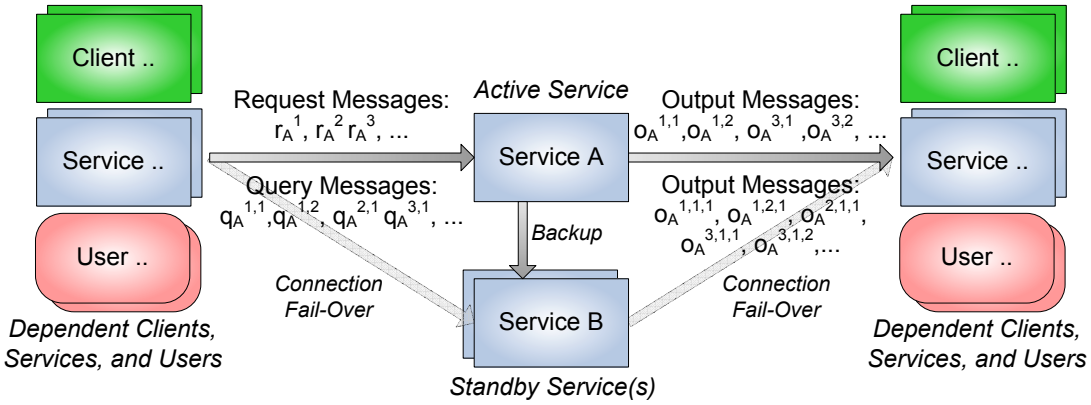


Figure 3.7: Active/standby method

In the warm-standby replication method, 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. 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 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 method does assure a linear history of state transitions only up to the last replicated service state, all dependent clients, services, and users need to be notified about a possible service state rollback. However, the new primary service A' is unable to verify by itself if a rollback has occurred.

In the hot-standby method, 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 acknowledgement 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 acknowledgement $o_B^{t,1}$ 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 to the standby service B, which becomes the new primary service A' based on the current state. In contrast to the warm-standby replication method, the hot-standby method 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 ..., $o_A^{t,y-1}$, $o_A^{t,y}$ for an already processed request message r_A^t. Furthermore, the primary service A may have failed before sending all output messages ..., $o_A^{t,x-1,y-1}$, $o_A^{t,x-1,y}$, $o_A^{t,x,y-1}$, $o_A^{t,x,y}$ for previously processed query messages ..., $q_A^{t,x-1}$, $q_A^{t,x}$. All dependent clients, services, and users need to be notified that a fail-over has occurred. The new primary service A' resends all output messages ..., $o_{A'}^{t,y-1}$, $o_{A'}^{t,y}$ related to the previously processed request message $r_{A'}^t$, while all dependent clients, services, and users ignore duplicated output messages. Unanswered query messages ..., $q_A^{t,x-1}$, $q_A^{t,x}$ are reissued to the new primary service A' as query messages ..., $q_{A'}^{t,x-1}$, $q_{A'}^{t,x}$ by dependent clients, services, and users.

The active/standby method always requires to notify dependent clients, services, and users about the fail-over. 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 $MTTR_{recovery}$ of the active/warm-standby method depends on the time to detect a failure, the time needed for fail-over, time to reconfigure client connections, and the time needed by clients and service to catch up based on the last replicated service state. The $MTTR_{recovery}$ of the active/hot-standby method only depends on the time to detect a failure, the time needed for fail-over, and the time to reconfigure client connections.

The only performance impact of the active/warm-standby method is the atomic service state snapshot, which briefly interrupts the primary service. The active/hot-standby method adds communication latency to every request message, as forwarding to the secondary node and waiting for the respective acknowledgement is needed. This 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 clients, 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 $n-1$ standby services B, C, ... requires fault tolerant multicast capability, *i.e.*, service group membership management and reliable multicast (Section 2.3.2). The introduced overhead directly depends on the protocol and may be $O(log_2(n))$ in the best case. Furthermore, a priority-based fail-over policy, *e.g.*, A to B to C to ..., is needed as well.

3.3.3 Asymmetric Active/Active Replication

In the asymmetric active/active replication method (Figure 3.8), two or more active services A, B, ... provide essentially the same service capability at tandem without state coordination, while optional standby services α, β, ... in an $n+1$ or $n+m$ configuration may replace failing active services. There is no synchronisation or replication between the active services.

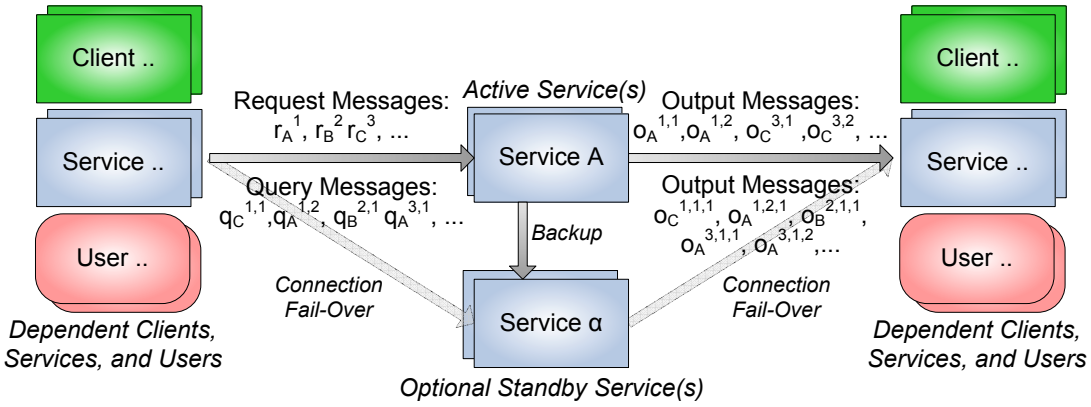


Figure 3.8: Asymmetric active/active method

Service state replication is only performed from the active services A, B, ... to the optional standby services α, β, ... 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 than standby services ($n > m$), and load balancing for using the active services 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 active services. 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 a dependency to the system, which requires its own redundancy strategy.

Similar to the active/standby method, the asymmetric active/active replication method requires notification of dependent clients, 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 $MTTR_{recovery}$ 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 method 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 under high-load conditions due to the availability of more resources and load balancing.

3.3.4 Symmetric Active/Active Replication

In the symmetric active/active replication method for service-level high availability (Figure 3.9), two or more active services A, B, ... offer the same capabilities and maintain a common global service state.

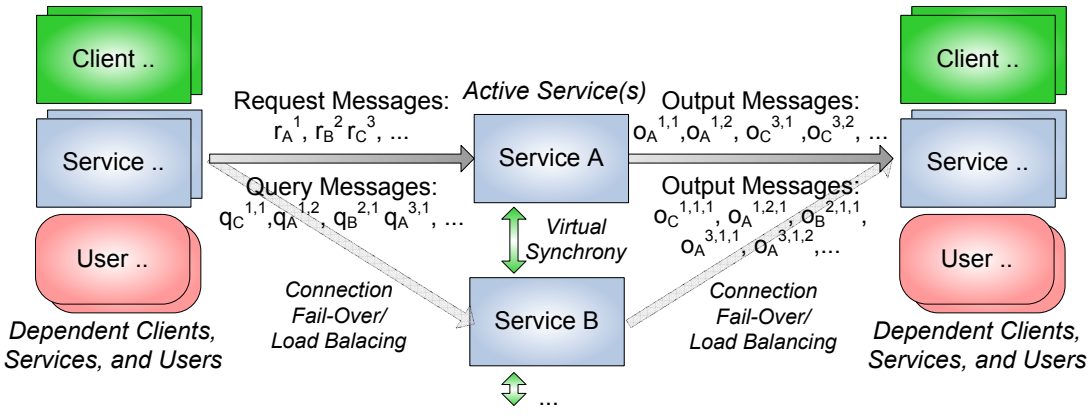


Figure 3.9: Symmetric active/active method

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 (Section 2.3.3). 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, ... A process group communication system (Section 2.3.2) 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 process group communication system for a distributed mutual exclusion. The latter is required if duplicated messages can't be simply ignored by dependent clients, services, and 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 $r^t_{A,B,...}$ through the process 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 clients, 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 a state transition. A distributed mutual exclusion implies a high overhead lock-step mechanism for exact-once output delivery. It should be avoided whenever possible. Message duplication and respective filters at the receiving end should be used instead.

The number of active services is variable at runtime and can be changed by either forcing an active service to leave the active 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 in the order they occur or based on a predetermined priority. New services join the service group by atomically overwriting their service state with the service state snapshot from a service group member, *e.g.*, S^t_A, before receiving following request messages, *e.g.*, r^{t+1}_A, r^{t+2}_A, ...

Query messages $q^{t,x}_A$, $q^{t,x}_B$, ... may be send directly to respective active services through the process group communication system to assure total order with conflicting request messages, but without replicating them across all service group members. Related output messages ..., $o^{t,x,y-1}_A$, $o^{t,x,y}_A$, $o^{t,x,y-1}_B$, $o^{t,x,y}_B$ are sent directly to the dependent service or user. In case of a failure, unanswered query messages, *e.g.*, ..., $q^{t,x-1}_A$, $q^{t,x}_A$, are reissued by dependent clients, services, and users to a different active service, *e.g.*, B, in the service group as query messages, *e.g.*, ..., $q^{t,x-1}_B$, $q^{t,x}_B$.

The symmetric active/active method also always requires to notify dependent clients, services, and users about failures in order to reconfigure their access to the service group through the process 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 for extensive reconfiguration. The $MTTR_{recovery}$ only depends on the time to reconfigure client connections. However, the process group communication system introduces a latency overhead, which increases with the number of active nodes n. This overhead may be in the best case $2l_{A,B}$ for two active services and $O(log_2(n))$ for more.

3.3.5 Comparison

All presented service-level high availability methods show certain interface and behaviour similarities. They all require to notify dependent clients, 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 method, fail-over and rollback scenarios, message duplication, and unanswered query messages.

The requirements for service interfaces in these service-level high availability methods 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 method, 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 symmetric active/active service-level high availability methods are equivalent for reliably delivering request messages in total order to all standby or active services. In fact, the active/hot-standby service-level high availability method uses a centralised process group communication commit protocol with fail-over capability for the central message sequencer. However, the overhead for active/hot-standby is typically lower as only one service is active.

Based on the presented conceptual service model and service-level high availability methods, active/warm-standby provides the lowest runtime overhead in a failure-free environment and the highest recovery impact in case of a failure (Table 3.2). Conversely, symmetric active/active together with active/hot-standby offer the lowest recovery impact, while symmetric active/active incurs the highest runtime overhead. However, this comparison highly depends on actual implementation details and possible performance/redundancy tradeoffs.

Method	$MTTR_{recovery}$	Latency Overhead
Warm-Standby	$T_d + T_f + T_r + T_c$	0
Hot-Standby	$T_d + T_f + T_r$	$2l_{A,B}$, $O(log_2(n))$, or worse
Asymmetric with Warm-Standby	$T_d + T_f + T_r + T_c$	0
Asymmetric with Hot-Standby	$T_d + T_f + T_r$	$2l_{A,\alpha}$, $O(log_2(n))$, or worse
Symmetric	$T_d + T_f + T_r$	$2l_{A,B}$, $O(log_2(n))$, or worse

T_d, time between failure occurrence and detection
T_f, time between failure detection and fail-over
T_c, time to recover from checkpoint to previous state
T_r, time to reconfigure client connections
$l_{A,B}$ and $l_{A,\alpha}$, communication latency between A and B, and A and α

Table 3.2: Comparison of replication methods

It is also noted that symmetric active/active and active/hot-standby have a n-times lower $MTTF$ for the software subsystem for which $MTTR_{recovery}$ is observed (Equation 3.23 in Section 3.1.4). When comparing the $\frac{MTTR_{recovery}}{MTTF_{component}}$ ratio, symmetric active/active and active/hot-standby have a n-times higher ratio than active/warm-standby.

Since previous work on providing high availability for HPC head and service nodes (Section 2.1) focused primarily on active/standby replication (Sections 2.1.1 and 2.1.2) and secondarily on asymmetric active/active replication (Section 2.1.3), this thesis targets symmetric active/active replication.

3.4 Summary

This Chapter provided the theoretical ground work of the research presented in this thesis. An extended generic taxonomy for service-level high availability has been presented that introduced new terms, such as asymmetric active/active and symmetric active/active, to resolve existing ambiguities of terms, such as active/active. The taxonomy also clearly defined the various configurations for achieving high availability of service and relevant metrics for measuring service availability. This extended taxonomy represented a major contribution to service availability research and development with respect to incorporating state-machine replication theory and resolving ambiguities of terms.

Current HPC system architectures were examined in detail and a more generalised HPC system architecture abstraction was introduced to allow identification of architectural availability deficiencies. HPC system services were categorised into critical and non-critical to describe their impact on overall system availability, while HPC system nodes were categorised into single points of failure and single points of control to pinpoint their involvement in system failures, to describe their impact on overall system availability, and to identify their individual need for a high availability solution. This analysis of architectural availability deficiencies of HPC systems represented a major contribution to the understanding of high availability aspects in the context of HPC environments.

Using the taxonomy and a conceptual service model, various methods for providing service-level high availability were defined and their mechanisms and properties were described in detail. A theoretical comparison of these methods with regards to their performance overhead and provided availability was presented. This comparison represented a major contribution to service availability research and development with respect to incorporating state-machine replication theory. It clearly showed that symmetric active/active replication provides the highest form of availability, while its performance impact highly depends on the employed process group communication protocol.

4 Prototypes

Chapter 3 identified individual critical system services running on HPC system head and service nodes that represent single points of failure and single points of control as major availability deficiencies, such as the job and resource management system service and the parallel file system metadata system service. Furthermore, it clearly showed that symmetric active/active replication provides the highest form of availability.

This Chapter details objectives, technical approach, architecture, design, test results, and conclusions for each of the following developed proof-of-concept prototypes:

- external symmetric active/active replication for the critical HPC job and resource management system service,
- internal symmetric active/active replication for the critical HPC parallel file system metadata system service,
- transparent symmetric active/active replication framework for services, and
- transparent symmetric active/active replication framework for dependent services.

The developed proof-of-concept prototypes centre around a multi-layered symmetric active/active high availability framework concept that coordinates individual solutions with regards to their respective field, and offers a modular approach that allows for adaptation to system properties and application needs. In the following, the symmetric active/active high availability framework concept is explained in more detail and the overarching approach for the developed proof-of-concept prototypes is explained.

4.1 Symmetric Active/Active High Availability Framework Concept

The symmetric active/active high availability framework concept (Figure 4.1) consists of four layers: communication drivers, group communication system, virtual synchrony runtime environment and applications/services. At the lowest layer, communication drivers

provide reliable point-to-point and multicast messaging capability. The group communication system additionally offers process group membership management and total order multicast. The virtual synchrony runtime environment offers adaptation of the group communication system capabilities to the virtual synchrony approach for service-level high availability in the form of easy-to-use interfaces for applications/services.

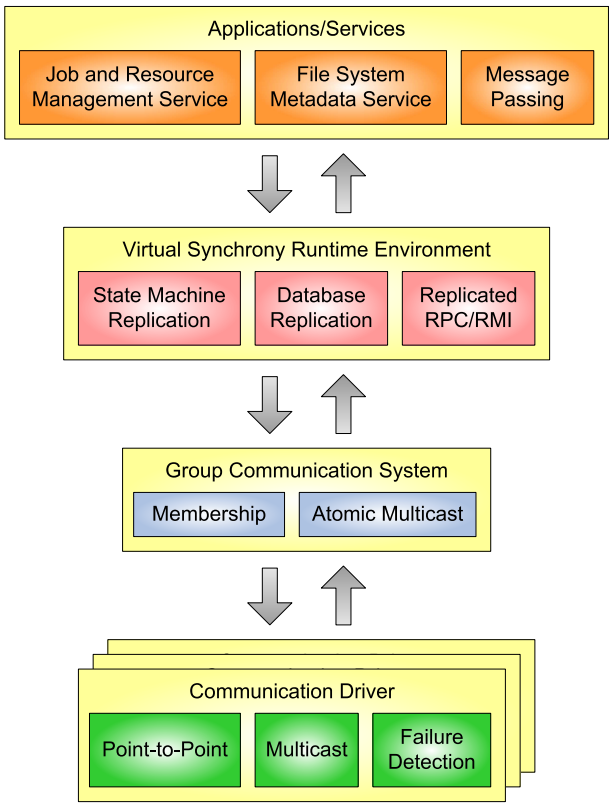


Figure 4.1: Symmetric active/active high availability framework concept

4.1.1 Communication Drivers

Today's HPC system architectures come with a variety of communication technologies, such as Myrinet, Infiniband, Quadrics Elan4, Cray portals, shared memory, and Ethernet. The symmetric active/active high availability framework concept is capable of supporting vendor supplied high-performance network technologies as well as established standards, such as IP, 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 can be found in some existing solutions. For example, Open MPI (Section 2.2.3) uses a component-based framework that encapsulates communication drivers using interchangeable and interoperable components.

4.1.2 Group Communication System

The group communication system contains all essential protocols and services to run virtual synchronous services for symmetric active/active high availability. Many (60+) group communication algorithms can be found in literature (Section 2.3.2). A pluggable component-based 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. Pluggable component-based frameworks for group communication mechanisms can be found in some existing solutions, such as in Horus (Section 2.3.3).

4.1.3 Virtual Synchrony Runtime Environment

The supported APIs at the virtual synchrony runtime environment are based on application properties. Deterministic and fully symmetrically replicated applications may use replication interfaces for state-machines and databases. Nondeterministic or asymmetrically replicated applications may use more advanced replication interfaces for replicated RPCs or remote method invocations (RMIs). This adaptation of group communication system capabilities to the virtual synchrony approach for service-level high availability is needed due to the group communication system‘s limited knowledge about individual application/service properties.

4.1.4 Applications/Services

There are many, very different, applications for the symmetric active/active high availability framework concept. This thesis primarily focuses on critical HPC system services that represent a single point of failure and control, such as the job and resource management service typically located on the head node or the parallel file system metadata service typically located on a dedicated service node. However, symmetric active/active replication can be used in any service-oriented architecture (SOA) to re-enforce critical services with appropriate redundancy strategies.

4.1.5 Approach

Since this thesis research targets efficient software state replication mechanisms for redundancy of services running on HPC head and service nodes, the developed proof-of-concept prototypes described in the following Sections 4.2-4.5 primarily focus on the virtual synchrony runtime environment within this symmetric active/active high availability framework concept. The implementations rely on an existing IP-based process

group communication system, since the lower layers of the symmetric active/active high availability framework concept can be found in various already existing solutions, such as Open MPI (Section 2.2.3) with its pluggable runtime environment and high-performance network components, or the Horus system (Section 2.3.3) with its stackable component approach for group communication services. Future work may combine these existing solutions with the virtual synchrony runtime environment presented in this thesis.

The first two proof-of-concept prototypes (Sections 4.2 and 4.3) demonstrate two new approaches for providing symmetric active/active high availability for two different HPC system services, the HPC job and resource management service and the HPC parallel file system metadata service. The two proof-of-concept prototypes offer completely different interfaces between service and group communication system. The first utilises the *external* service interface that is visible to clients (Section 4.2), while the second tightly integrates with the *internal* service interface (Section 4.3). Advantages and shortcomings of both proof-of-concept prototypes are examined.

The next two preliminary proof-of-concept prototypes (Sections 4.4 and 4.5) demonstrate mechanisms within the virtual synchrony runtime environment and abstractions at the virtual synchrony runtime environment interface to provide *transparent symmetric active/active replication* in *client-service* scenarios (Section 4.4) as well as in scenarios with with *dependent services* (Section 4.5). Accomplishments and limitations of both preliminary proof-of-concept prototypes are discussed.

4.2 External Symmetric Active/Active Replication for the HPC Job and Resource Management Service

One of the most important HPC system services running on the head node is the job and resource management service, also commonly referred to as batch job scheduler or simply the scheduler. If this critical HPC system service goes down, all currently running jobs, *i.e.*, applications running on the HPC system, loose the service they report back to, *i.e.*, their logical parent. They typically have to be restarted once the HPC job and resource management service is up and running again after the head node has been repaired.

Previous research and development efforts offered active/standby and asymmetric active/active high availability solutions (Sections 2.1.2 and 2.1.3) for various HPC job and resource management services, such as for OpenPBS [44–46] and for Moab [50, 51].

The research and development effort presented in this Section targets symmetric active/active high availability for the HPC job and resource management service. As part of this effort, the fully functional JOSHUA [192–194] proof-of-concept prototype has been

developed in C on Linux to provide symmetric active/active high availability for the Terascale Open-Source Resource and QUEue Manager (TORQUE) [195, 196] using the external replication approach. TORQUE is a fork of the original OpenPBS version 2.3.12 [47] that has been significantly enhanced. It is maintained and commercially supported by Cluster Resources Inc.. TORQUE is used in many small-to-mid size HPC systems.

While the symmetric active/active high availability concept and the external replication approach was developed by me [192, 197, 198], the detailed software design and the actual implementation was carried out under my supervision by Kai Uhlemann [193, 194] during his internship at Oak Ridge National Laboratory, USA, for his Master of Science (MSc) thesis at the University of Reading, UK. His thesis [194], titled "High Availability for High-End Scientific Computing", primarily focuses on the developed JOSHUA proof-of-concept prototype. It appropriately references my prior work in symmetric active/active high availability [129, 198, 199]. The work performed under my supervision has been published in a co-authored paper [193].

4.2.1 Objectives

Since no previous solution for providing symmetric active/active high availability for a HPC system service existed, the development of this proof-of-concept prototype had the following primary objectives:

- development of the first fully functional symmetric active/active prototype,
- development of an external symmetric active/active replication infrastructure,
- measuring the introduced symmetric active/active replication overhead, and
- gaining experience with symmetric active/active replication in HPC environments.

The HPC job and resource management service was chosen for this proof-of-concept prototype as it is the most important HPC system service running on the head node and it is not response-latency sensitive, *i.e.*, a higher response latency of service requests, such as for adding a job to the job queue or reporting job statistics back to the user, is acceptable to a certain degree if this performance trade-off results in higher availability.

TORQUE was chosen for this proof-of-concept prototype as it is widely used, open source, and supported by Cluster Resources Inc. as well as the open source community. A TORQUE-based proof-of-concept prototype can be easily adapted to other HPC job and resource management services as it supports the widely-used Portable Batch System (PBS) [47] interface standard.

4.2.2 Technical Approach

The concept of external symmetric active/active replication (Figure 4.2) 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 process group communication system that mimics the service interface using separate event handler routines.

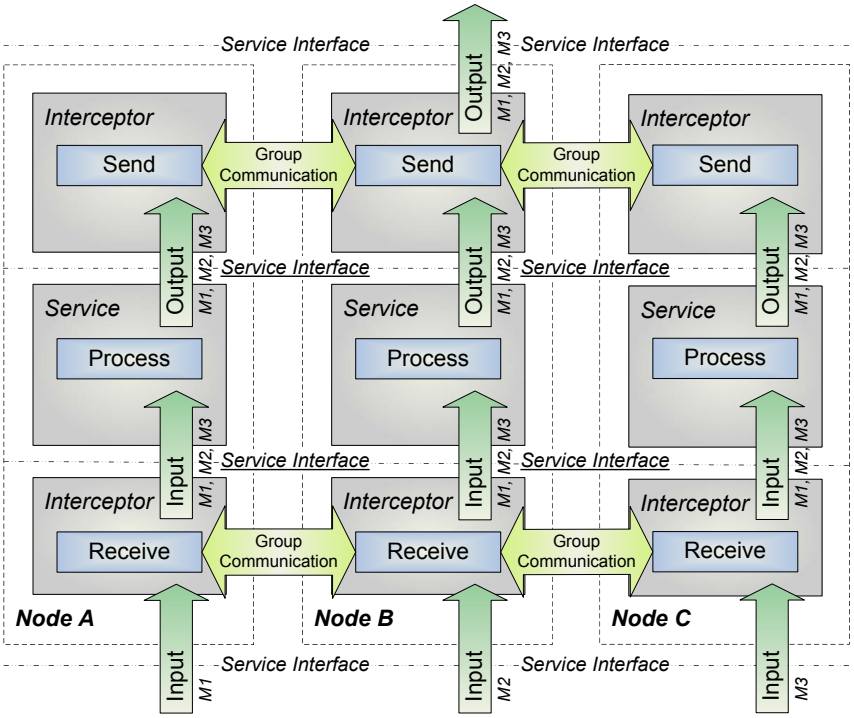


Figure 4.2: Symmetric active/active replication architecture using external replication by service interface utilisation

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 process group. Once totally ordered and reliably delivered, each interceptor process group member calls the original command to perform operations at each replicated service. Service output may be routed through the interceptor process group for at-most-once delivery if dependent clients, services, or users can't handle duplicated messages (Section 3.3.4).

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 process group communication system may lead to performance degradation as external replication implies coarse-grain synchronisation of state changes at the service interface level, *i.e.*, lock-step execution of service queries and requests.

The technical approach for providing symmetric active/active high availability for the HPC job and resource management service focuses on the external replication method, since HPC job and resource management implementations are very complex and typically support the PBS [47] service interface. While modification of service code is prohibitively time consuming and error prone, the PBS service interface is a widely supported standard. The HPC job and resource management service TORQUE [195, 196] supports the PBS service interface as well as OpenPBS [44–46], Moab [50, 51], and Sun Grid Engine [35].

4.2.3 Architecture and Design

The JOSHUA solution is a generic approach for offering 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 4.3) providing symmetric active/active high availability without any interruption of service and without any loss of state.

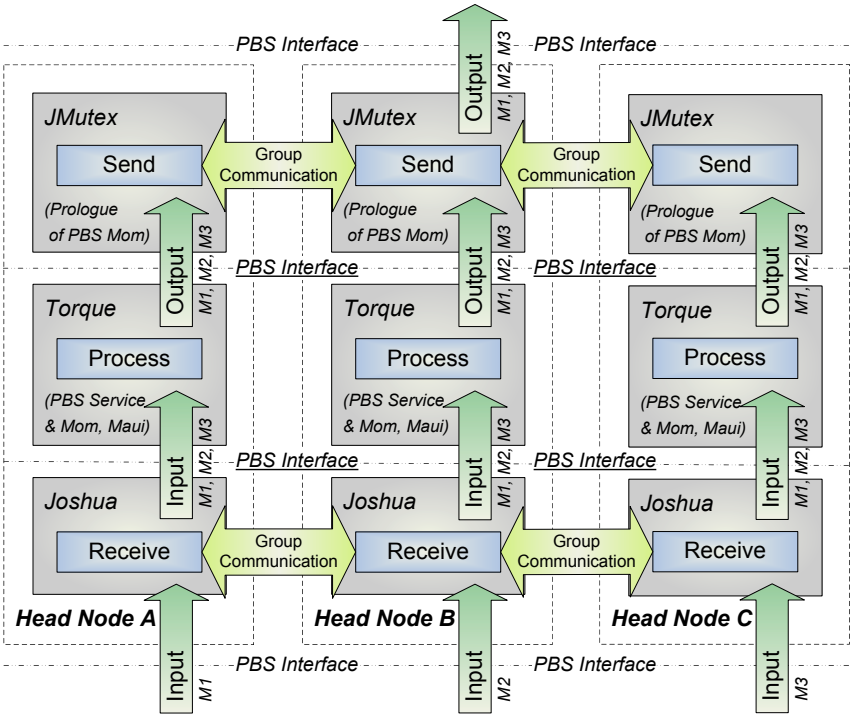


Figure 4.3: External replication architecture of the symmetric active/active HPC job and resource management service

A process group communication system is used for total ordering of incoming messages and for at-most-once job launch, *i.e.*, for unifying output to the PBS job start mechanism. Consistently produced output to PBS clients is delivered from every active head node. Clients filter duplicated messages based on a simple message sequencing scheme.

Group Communication System

In HPC systems, multiple concurrent queries and requests are expected to arrive simultaneously. In the particular case of the HPC job and resource management service, two users may want to schedule a job at the same time. A communication history algorithm (Section 2.3.2) is preferred to order queries and requests in a symmetric active/active head node scenario, since it performs well under heavy load with concurrent messages. However, for relatively light-load scenarios, the post-transmission delay is high.

The JOSHUA proof-of-concept prototype implementation uses the Transis process group communication system (Section 2.3.3) for reliable message delivery, total message order, and membership management, since Tansis uses a communication history algorithm, provides a sophisticated extended virtual synchrony mechanism, offers an easy-to-use programming interface, and is easily deployable in POSIX-compliant OSs [186], such as Linux. The post-transmission delay issue is not as important, since the HPC job and resource management service is not response-latency sensitive.

Software Design

Conceptually, the JOSHUA prototype implementation software design (Figure 4.4) consists of the following major parts:

- the JOSHUA client commands, `jsub`, `jdel`, and `jstat`, reflecting PBS compliant behaviour to the user, equivalent to the PBS client commands, `sub`, `del`, and `stat`;
- the Transis process group communication system with its extended virtual synchrony implementation for reliable, totally ordered message delivery and fault-tolerant process group membership management, a service running on each active head node;
- the JOSHUA service running on each active head node and imitating PBS compliant client behaviour by locally invoking the original PBS client commands: `sub`, `del`, and `stat`;
- the original PBS service on running each active head node and the original PBS Mom service running on a single compute node, a subset, or all compute nodes;
- the distributed mutual exclusion client scripts, jmutex and jdone, needed for at-most-once job launch on the compute nodes; and
- a watchdog service running each active head node to guard the other services and to ensure fail-stop behaviour by shutting down all local services of a active head node if one of them fails, a self-fencing technique (Section 2.1.4).

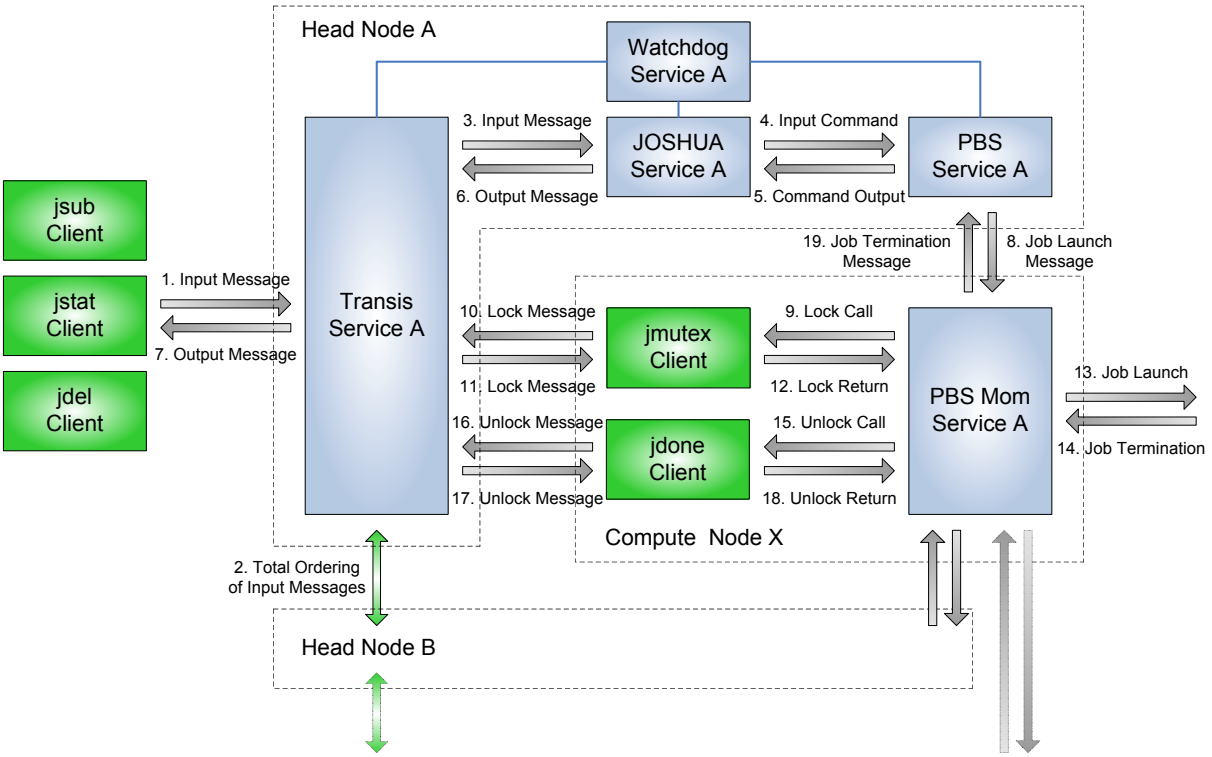


Figure 4.4: External replication design of the symmetric active/active HPC job and resource management service

Note that the JOSHUA prototype implementation does not provide a JOSHUA client command for signalling an executing batch job, *i.e.*, a `qsig` equivalent, as this operation is asynchronous in nature and does not change the state of the HPC job and resource management service. The original PBS client command may be executed on any replicated head node independently of the deployed replication infrastructure.

Failure-Free Operation

The proof-of-concept implementation prototype is based on the PBS compliant TORQUE HPC job and resource management system that employs the TORQUE PBS service together with the Maui [200] scheduler on each active head node and a single or a set of PBS mom services on compute nodes. Each PBS mom service is capable of communicating to each TORQUE PBS service on every active head node (TORQUE v2.0p1 feature), which allows the reuse of PBS mom services. However, this is not a requirement of the JOSHUA solution. Dedicated PBS mom services may be used for each TORQUE PBS service instance. The Maui scheduling policy is set to FIFO (default) to produce deterministic scheduling behaviour on all active head nodes.

During normal operation, the JOSHUA client commands, `jsub`, `jdel`, and `jstat`, per-

form job submission, deletion and statistics retrieval by connecting to the JOSHUA service group, issuing the respective command, `sub`, `del`, and `stat`, locally at all active head nodes, and relaying the output back to the user. Fundamentally, the JOSHUA client commands and service act in concert as an interceptor for PBS user commands to provide global ordering of user input for virtual synchrony on all active head nodes. The JOSHUA client commands may be invoked on any of the active head nodes or from a separate login service node as the commands contact the JOSHUA service group via the network. The JOSHUA client commands may even replace the original PBS commands in the user context using a shell alias, *e.g.*, `'alias qsub=jsub'`, in order to offer 100% PBS service interface compliance at the user level.

Once a submitted job is first in the TORQUE job queue and there is an appropriate amount of resources available, each TORQUE service connects to one PBS mom service on the compute nodes to start the job. The JOSHUA distributed mutual exclusion client scripts are part of the job start prologue and perform a distributed mutual exclusion using the Transis process group communication system to ensure that the job gets started only once, and to emulate the job start for all other attempts for this particular job. Once the job has finished, the distributed mutual exclusion is released and all TORQUE services receive the respective job statistics report.

Failure Handling

Upon failure of an active head node, Transis informs all JOSHUA services to exclude the failed head node from any further communication. The PBS mom services simply ignore the failed head node when sending job statistics reports, while the distributed mutual exclusion performed by the JOSHUA distributed mutual exclusion client scripts rely on the Transis process group membership management for releasing any locks.

The JOSHUA client commands receive duplicated output from all active head nodes and are not affected by a head node failure. Transis delivers all messages even if a client needs to reconnect to the service group via a different service group member, a Transis specific caching feature for temporary absences of clients.

There is no service failover necessary as the healthy active head nodes continue to provide the service and the system parts on the compute nodes are able to adapt. Since Transis is able to deal with multiple simultaneous failures in the same way it deals with multiple sequential failures, the HPC job and resource management service is provided transparently as long as one head node survives.

Head node failures degrade the overall availability of the system by reducing the number of redundant components. Replacement of head nodes that are about to fail, *e.g.*, due to a recent fan fault, allows to sustain and guarantee a certain availability. The JOSHUA

solution permits head nodes to join and leave using the Transis process group communication system for coordination. Leaving the active service group is actually handled as a forced failure by causing the JOSHUA service to shutdown via a signal. Joining the active service group involves copying the current state of an active service over to the joining head node.

The current JOSHUA proof-of-concept prototype implementation uses configuration file modification and PBS command replay to copy the state of one TORQUE service over to another. This is due to the fact that the PBS interface does not provide a solution for starting up a replica. As a consequence, it also is impossible to support holding and releasing jobs as the PBS command replay causes inconsistencies in the job queue of the joining TORQUE service when holding jobs.

4.2.4 Test Results

The fully functional 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 450MHz processors with 512MB of memory and 8GB of disk space. All nodes were connected via a single Fast Ethernet (100MBit/s full duplex) hub. Debian GNU/Linux 3.1 has been used as OS in conjunction with Transis v1.03, TORQUE v2.0p5, and Maui v3.2.6p13. Performance results are averages over 100 test runs. Failures were simulated by unplugging network cables and by forcibly shutting down individual processes.

The job submission latency overhead (Figure 4.5 or Section A.1.1) introduced by the network communication between the JOSHUA client commands, the Transis service and the JOSHUA service is in an acceptable range. The latency overhead between TORQUE and TORQUE+JOSHUA on a single head node, 36ms or 37%, can be attributed to additional communication on the head node between Transis and the JOSHUA service, and to the delegated execution of the TORQUE client commands by the JOSHUA service. The significant latency overhead on two head nodes, 163ms or 170%, can be explained by the Transis process group communication protocol overhead. Overall a job submission latency overhead of only 251ms or 256% for a four head node system is still acceptable for a HPC system.

The job submission throughput overhead (Figure 4.6 or Section A.1.1) reflects 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 33.32s or at 31% of original job submission throughput performance for a four head node system is also acceptable.

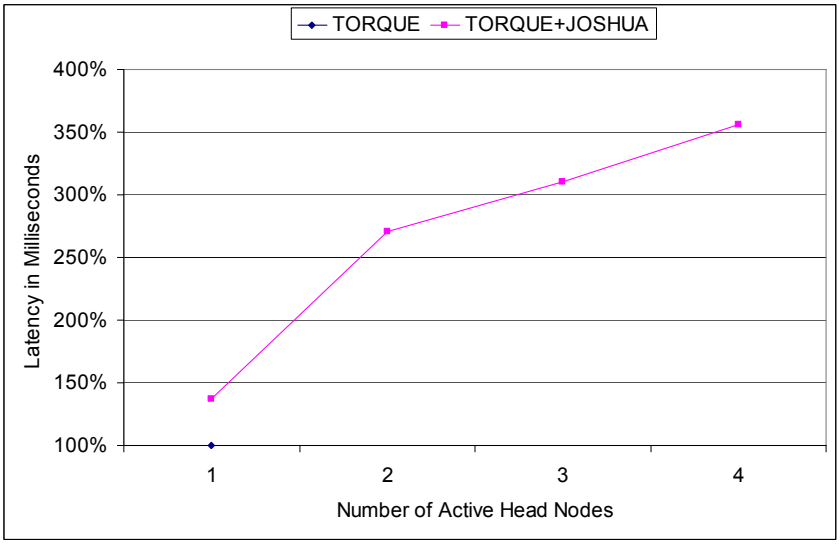


Figure 4.5: Normalised job submission latency performance of the symmetric active/active HPC job and resource management service prototype (averages over 100 tests)

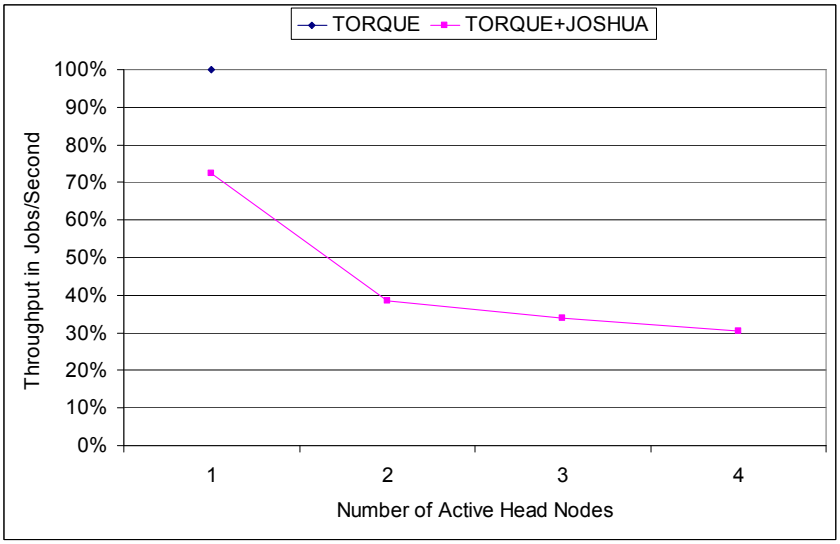


Figure 4.6: Normalised job submission throughput performance of the symmetric active/active HPC job and resource management service prototype (100 submissions, averages over 100 tests)

Extensive functional testing revealed correct behaviour during normal system operation and in case of single and multiple simultaneous failures. Head nodes were able to join the service group, leave it voluntary, and fail, while job and resource management state was maintained consistently at all head nodes and service was provided transparently to applications and users. However, the PBS mom service and the JOSHUA distributed mutual exclusion client scripts run on compute nodes. The developed proof-of-concept prototype is not capable of tolerating failures of these compute nodes.

The experienced $MTTR_{recovery}$ was dominated by the heartbeat interval of the Transis process group communication system, *i.e.*, by the communication timeout between group members, which is configurable at startup and was set to 500 milliseconds. The developed proof-of-concept prototype provided an extraordinary service availability (Figure 4.7 or Section A.1.1) due to its low $MTTR_{recovery}$.

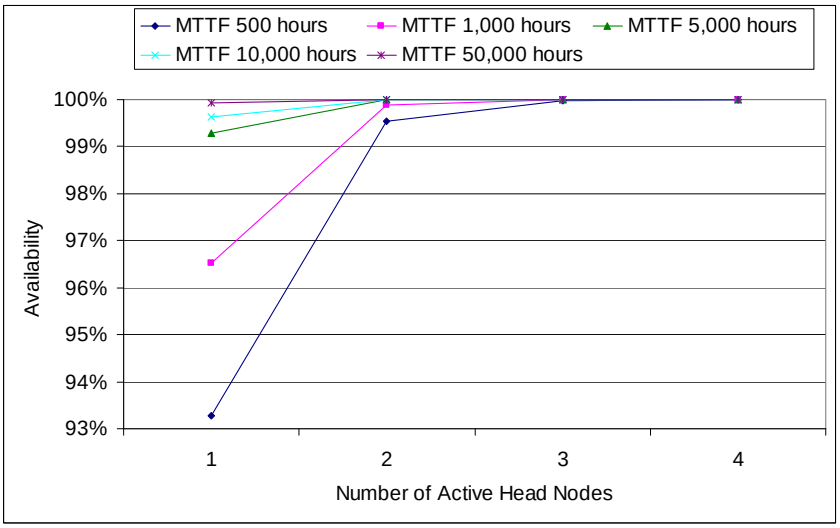


Figure 4.7: Availability of the symmetric active/active HPC job and resource management service prototype

Using Equation 3.23 in Section 3.1.4 and a $MTTR_{component}$ of 36 hours, service availability could be improved for a $MTTF_{component}$ of 1,000 hours from 96.248% to 99.998% in a four-node system. With a $MTTF_{component}$ of 5,000 hours, service availability could be improved from 99.285% to 99.995% in a two-node system, an increase from a two-nines to a four-nines rating just by using a second node (Table 3.1 in Section 3.1.4). Adding another node would increase service availability to 99.99996%, a six-nines rating.

It has to be noted that when approaching the 100% availability mark with this solution, certain catastrophic failures, such as a site-wide power outage, are not masked due to a missing multi-site redundancy strategy. A local redundancy strategy of 2-3 symmetric active/active head nodes should be sufficient to cover common failures.

4.2.5 Conclusions

With the JOSHUA proof-of-concept prototype, the first fully functional solution for providing symmetric active/active high availability for a HPC system service was developed using the external replication approach that wraps an existing service into a virtually synchronous environment. The prototype performed correctly and offered an acceptable job submission latency and throughput performance. Significant experience was gained with respect to architecture, design, and performance challenges for symmetric active/active replication in HPC environments.

The developed proof-of-concept prototype provided an extraordinary availability. Assuming a node MTTF of 5,000 hours, service availability improved from 99.285% to 99.995% in a two-node system, and to 99.99996% with three nodes. A change from a two-nines rating to four nines in a two-node system, and to six nines with three nodes.

The reliance of the PBS service on the PBS mom service on the compute nodes revealed the existence of more complex interdependencies between individual system services on head, service and compute nodes. Providing high availability for a critical system service also needs to deal with dependent critical system services due to the serial availability coupling of dependent system components (Equations 3.19 and 3.21 in Section 3.1.4).

Although the job submission latency overhead of 251ms on four symmetric active/active head nodes is in an acceptable range for the HPC job and resource management, this may not be true for more latency-sensitive services, such as the HPC parallel file system metadata service. Significant performance improvements may be necessary.

4.3 Internal Symmetric Active/Active Replication for the HPC Parallel File System Metadata Service

One of the second most important HPC system services running on the head node of small-scale HPC systems and one of the most important HPC system services running on a dedicated service node in large-scale HPC systems is the metadata service (MDS) of the parallel file system. Since the MDS keeps the records of all directories and files located on storage services of the parallel file system, a failure of this critical HPC system service results in an inability for applications running on the HPC system to access and store their data, and may result in file system corruption and loss of stored data.

Previous research and development efforts offered active/standby high availability solutions using a shared storage device (Section 2.1.1) for various MDSs, such as for the parallel Virtual File System (PVFS) [38–40] and for the Lustre cluster file system [41–43], with questionable correctness and quality of service.

The research and development effort presented in this Section targets a symmetric active/active high availability solution for the MDS of a parallel file system. As part of this effort, a fully functional proof-of-concept prototype [201] has been developed in C on Linux to provide symmetric active/active high availability for the MDS of PVFS using the internal replication approach.

While the symmetric active/active high availability concept and the internal replication approach was developed by me [192, 197, 198], the detailed software design and the actual implementation was carried out under my supervision by Li Ou [201–204] during his internship at Oak Ridge National Laboratory, USA, for his Doctor of Philosophy (PhD) thesis at the Tennessee Technological University, USA. His thesis [204], titled "Design of a High-Performance and High-Availability Distributed Storage System", primarily focuses on various performance aspects of distributed storage systems and secondarily on the high availability aspect. It appropriately references my prior work in symmetric active/active high availability [197–199]. The work performed under my supervision has been published in two co-authored papers [201, 203]. Another co-authored paper [202] has been submitted to a journal and is currently under review.

4.3.1 Objectives

Since the first proof-of-concept prototype for providing symmetric active/active high availability for a HPC system service (Section 4.2) focused on the less response-latency sensitive HPC job and resource management service and on an external symmetric active/active replication infrastructure, the development of this proof-of-concept prototype had the following primary objectives:

- development of a fully functional symmetric active/active prototype for a response-latency sensitive critical HPC system service,
- development of an internal symmetric active/active replication infrastructure,
- measuring the introduced symmetric active/active replication overhead, and
- gaining further experience with symmetric active/active replication in HPC environments, especially with regards to possible performance enhancements.

The MDS of a parallel file system was chosen for this prototype as it is one of the second most important HPC system services running on the head node of small-scale HPC systems and one of the most important HPC system services running on a dedicated service node in large-scale HPC systems. It is very response-latency sensitive, *i.e.*, a higher response latency of service requests, such as for opening the MPI executable or creating

a checkpoint file for each compute node, is *not* acceptable if this performance trade-off results in a significant slowdown of such common file system operations.

PVFS was chosen for this proof-of-concept prototype as it is widely used in HPC, open source, and supported by the PVFS development team as well as the open source community. A PVFS-based proof-of-concept prototype can be easily adapted to other parallel file systems as it has a modular architecture and is mostly in user space. In contrast to PVFS, the Lustre cluster file system is entirely in kernel space, which offers better performance but significantly complicates development, *e.g.*, a crash of Lustre clients or services results in a kernel module crash requirering a reboot.

4.3.2 Technical Approach

The concept of internal symmetric active/active replication (Figure 4.8) allows each active service of a replicated service group to accept query and request messages from external clients individually, while using a process group communication system for total state change message order and reliable state change 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. Consistently produced service group output may be routed through the process group communication system for at-most-once delivery if dependent clients, services, and users can‘t handle duplicated messages (Section 3.3.4)

For example, a networked service that changes its state based on RPCs, such as the MDS of a parallel file system, 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. RPCs and respective state changes are decoupled and executed by separate event handler routines. RPC return messages may be unified via the process group communication system, delivered by every process group member, or delivered by only one process group member and temporarily cached by others.

This method requires modification of existing service 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 process group communication system may lead to performance enhancements. Furthermore, internal replication allows fine-grain synchronisation of state changes, such as pipelining, due to the decoupling of incoming requests and respective state changes.

The technical approach for providing symmetric active/active high availability for the MDS of a parallel file system focuses on the internal replication method, since the MDS is very response-latency sensitive.

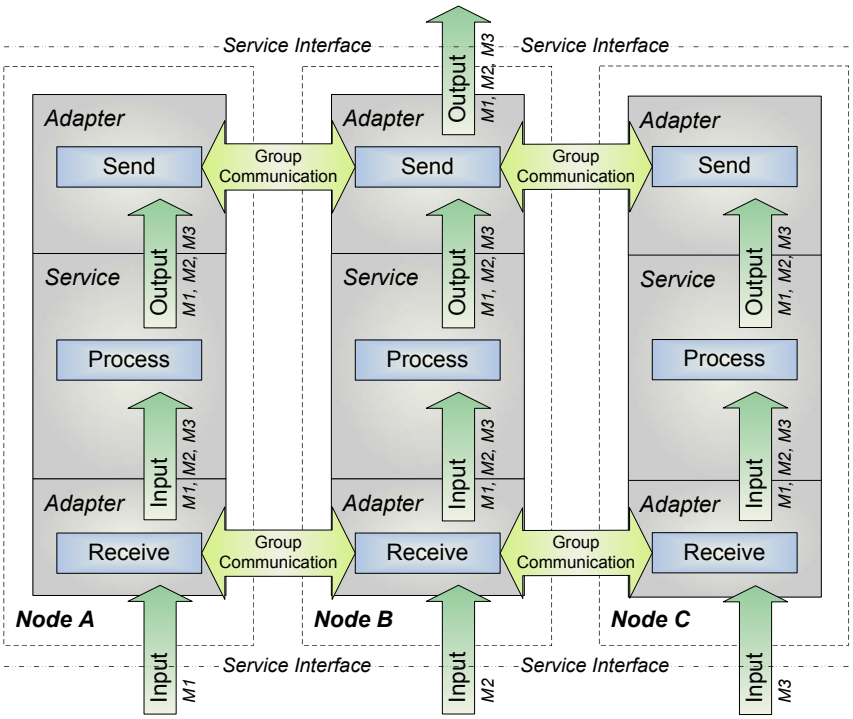


Figure 4.8: Symmetric active/active replication architecture using internal replication by service modification/adaptation

4.3.3 Architecture and Design

The developed proof-of-concept prototype is a customised implementation for offering symmetric active/active high availability for the MDS of PVFS. It is based on the internal RPC and state change mechanisms of PVFS and utilises adaptors as part of the internal replication architecture (Figure 4.9) to provide symmetric active/active high availability without any interruption of service and without any loss of state.

The process group communication system is only used for total ordering of state changes. Consistently produced MDS output to clients is delivered by only one active service node, the service node the client is directly connected to. The last produced output to a not directly connected client is cached to allow seamless connection fail-over. An incoming message from a client serves as an acknowledgement for a previously produced output message, since clients can't have outstanding queries or requests while issuing new ones, *i.e.*, file system operations are atomic at the client, a POSIX consistency requirement for file system access [186].

Group Communication System

The reliable, atomic multicast protocol of the Transis process group communication system has been improved to provide for lower latency overhead, especially in situations

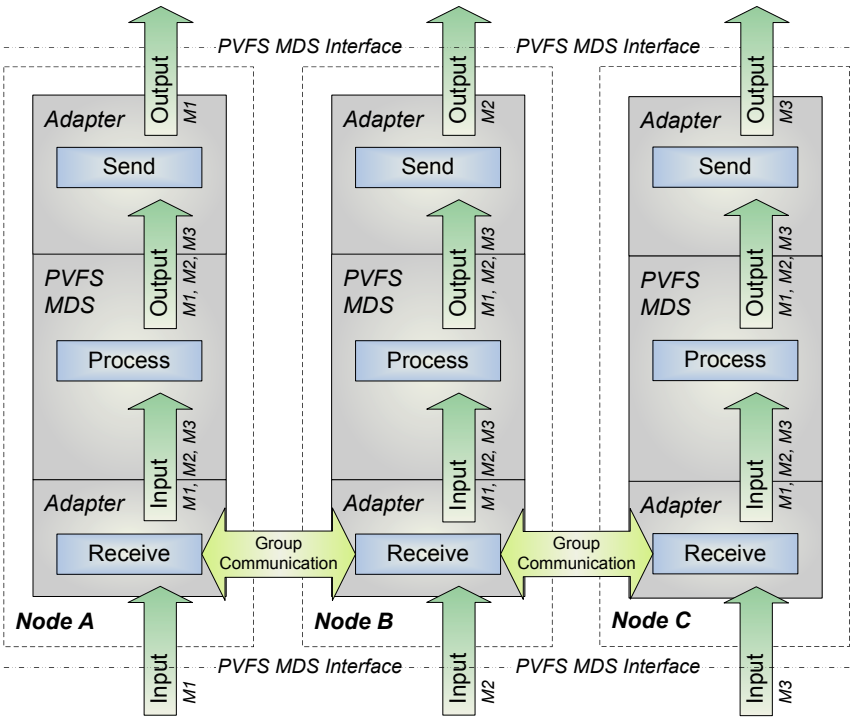


Figure 4.9: Internal replication architecture of the symmetric active/active HPC parallel file system metadata service

where the network is underutilised. The new *fast delivery* protocol optimises the total message ordering process by waiting for messages only from a subset of the group, and by *fast acknowledging* messages on behalf of other machines. This new feature drastically reduces the post-transmission delay of the communication history algorithm for total message order in Transis.

Total Order Broadcast in Transis The original Transis process group communication algorithm for total order broadcast uses message sequencing by senders in conjunction with a logical clock for the sequence counter at each sender to assure causal order broadcast at a lower layer. In the layer above, total order broadcast is achieved using the message sequence number, *i.e.*, causal order, and using sender Identifier (Id) order in case of messages with equal sequence numbers, *i.e.*, a predetermined order for independent simultaneous messages based on origin.

Transis uses a negative acknowledgement scheme, *i.e.*, retransmissions are requested if there is a missing message. A heartbeat mechanism generates periodic acknowledgements by each process group member to assure progress of the communication history algorithm in light-load scenarios, and for failure detection. The heartbeat interval is configurable at startup and represents a trade-off between latency in heavy- and light-load scenarios. Frequent heartbeats result in faster delivery, but also in more network traffic.

The developed fast delivery protocol separates the communication history algorithm progress responsibility from the heartbeat mechanism by allowing idle processes to acknowledge on behalf of others. This leads to a faster delivery of totally ordered messages as well as to an adaptation of the acknowledgement scheme to current network usage.

Notation and Definition A partition P consists of a group of processes $\{p_1, p_2, \cdots, p_N\}$. Each process in the group P has a distinct Id. For a process p, function $id(p)$ returns its Id. If the number of processes in the primary group P is N,

$$\forall p, q \in P, \quad id(p), id(q) \in \{1, 2, \cdots, N\}, \quad id(p) \neq id(q) \tag{4.1}$$

Each process $p \in P$ is associated with the functions $prefix$ and $suffix$:

$$prefix(p) = \{q | \forall q \in P, id(q) < id(p)\} \tag{4.2}$$

$$suffix(p) = \{q | \forall q \in P, id(q) > id(p)\} \tag{4.3}$$

The input to the fast delivery protocol is a stream of causally ordered messages from the underlying Transis broadcasting service. The i^{th} message sent by process q is denoted as $m_{q,i}$, where function $sender(m_{q,i}) = q$. If message $m_{q,i}$ is delivered before $m_{k,j}$ in process p, then function $deliver(m_{q,i}) < deliver(m_{k,j})$.

A *pending message* is a received, causally ordered message that has not been totally ordered, *i.e.*, delivered. A *candidate message* is a pending message that follows a message that has been delivered. The set of concurrent candidate messages is called the *candidate set* $M_p = \{m_1, m_2, \cdots, m_k\}$, which is the set of messages that are considered for the next slot in the total message order. It is associated with the function $senders$:

$$senders(M_p) = \{sender(m_i) | \forall m_i \in M_p\} \tag{4.4}$$

The *deliver set* Md_p is the set of messages ready to be delivered, *i.e.*, totally ordered, at process p, where $Md_p \subseteq M_p$.

Fast Delivery Protocol The fast delivery protocol (Figure 4.10) forms the total message order by waiting for messages only from a subset of the process group. Assuming a candidate message m is in candidate set M_p, the following delivery criterion is used to define which messages a process has to wait for before delivering m:

1. Add m into deliver set Md_p when:

$$prefix(sender(m)) \subseteq senders(M_p) \tag{4.5}$$

2. Deliver the messages in the Md_p with the following order:

$$\forall m_i, m_j \in Md_p, id(sender(m_i)) < id(sender(m_j)) \longrightarrow deliver(m_i) < deliver(m_j) \tag{4.6}$$

When receiving a regular message m in machine p:

```
if (m is a new candidate message) {
      add m into candidate set M_p
}

if (M_p ≠ φ) {
   for all (m_i ∈ M_p) {
      if (prefix(sender(m_i)) ⊆ senders(M_p)) {
         add m_i into delivery set Md_p
      }
   }
}

if (Md_p ≠ φ) {
   deliver all messages m_j in Md_p in the order of id(sender(m_j))
}

if (sender(m) ∈ suffix(p)) and
   (message m is a total order message) and
   (no messages are waiting to be broadcast from p) and
   (∄m_i ∈ M_p, id(sender(m_i)) = p) {
   fast acknowledge m
}
```

Figure 4.10: Fast delivery protocol for the Transis group communication system

To speedup the delivery of m, idle processes should immediately acknowledge m on behalf of other processes. If a process q receives a message m, and q is idle, q broadcasts a fast acknowledgement when:

$$sender(m) \in suffix(q) \tag{4.7}$$

Fast acknowledgement reduces the latency of message delivery, however, it injects more network traffic. If communication is heavy, fast acknowledgement may burden network and processes, thus increase delivery latency. To reduce the cost of fast acknowledgement,

the following acknowledgement criterion is defined. Fast acknowledge a message m from a process q when:

1. message m is a total order message, and
2. there is no message waiting to be sent from the process q, and
3. there is no message from the process q in the candidate set M_p, *i.e.*,

$$\nexists m_j \in M_p, id(sender(m_j)) = q \tag{4.8}$$

There are almost no additional acknowledgements injected into the network when communication in the process group is heavy, since conditions 2 and 3 are very unlikely to be satisfied simultaneously.

Software Design

Conceptually, the symmetric active/active PVFS MDS prototype implementation software design (Figure 4.11) consists of the following major parts:

- modified PVFS MDS clients running on compute nodes;
- the modified PVFS MDS service running on each active head or service node;
- the Transis process group communication system with the fast delivery protocol for total message order, a service running on each active head or service node; and
- a watchdog service running each active head node to guard the other services and to ensure fail-stop behaviour.

Note that PVFS storage services are not included in the symmetric active/active PVFS MDS architecture and design as PVFS clients access storage services independently and directly, after receiving proper authorisation from the MDS. Data redundancy strategies for PVFS storage services, such as distributed RAID, are ongoing research and development efforts [205–207] and outside the scope of this thesis.

Failure-Free Operation

To balance workloads among multiple active MDSs, a client either randomly chooses one MDS out of the active MDS group to send a message or uses a preferred active MDS associated to its specific HPC system partition.

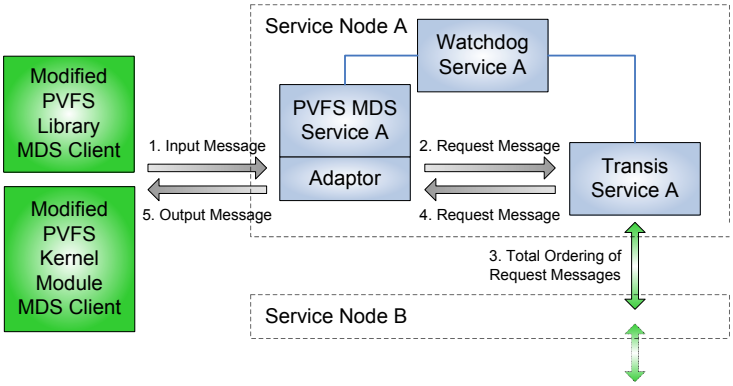
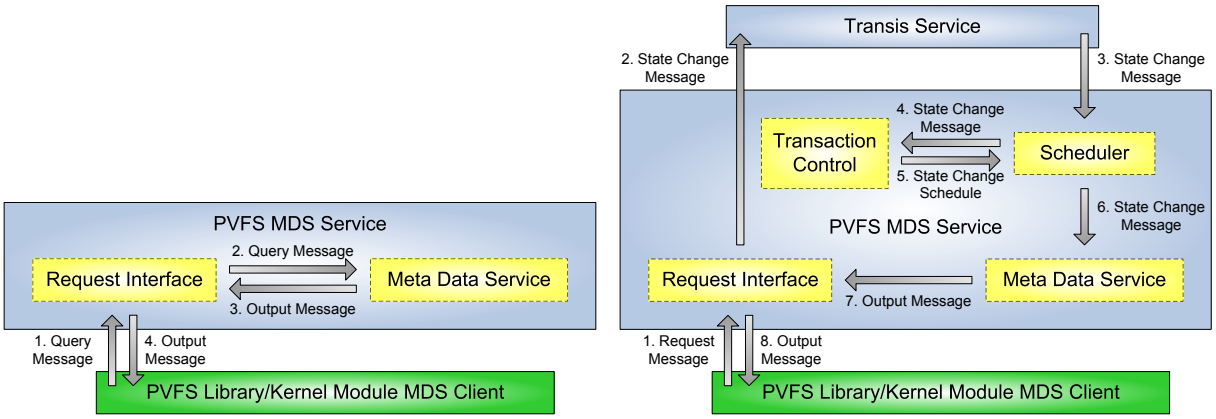


Figure 4.11: Internal replication design of the symmetric active/active HPC parallel file system metadata service



(a) Queries (b) Requests

Figure 4.12: Query and request handling of the symmetric active/active HPC parallel file system metadata service

All incoming messages are received and pre-processed by the MDS request interface module (Figure 4.12), which interprets the message content and detects if the message requests a state change.

Query messages do not request state changes and are immediately processed by the MDS module (Figure 4.12a). Respective output messages are returned through the MDS request interface module directly to the client.

Request messages do result in state changes and respective state change messages are forwarded to the Transis process group communication system after an initial preparation of the expected state change (Figure 4.12b). Once ordered, Transis delivers state change messages to the MDS scheduler module that manages a queue of totally ordered state change messages. The MDS scheduler module decouples incoming, totally ordered state change messages from their execution to allow in-order execution of concurrent state

changes and out-of-order execution of non-concurrent state changes. The transaction control module provides the necessary locking mechanisms to hold concurrent state changes until they become non-concurrent. The execution of state changes is performed by the MDS module and respective output messages are returned through the MDS request interface module directly to the client.

The MDS scheduler module also decouples the receiving of totally ordered state change messages via Transis from scheduling, execution, and output delivery using a separate thread to further improve throughput performance.

The MDS module performs all necessary state lookups, such as getting a file attribute, and state changes, such as creation of a new file or directory. Some request messages trigger a series of state lookups and changes as part of their state change. For example, the request to create a new file involves the following three MDS module operations: (1) reading the directory to make sure that no other object has the same name, (2) creation of the file object, and (3) adding the file object handle to the parent directory object. All operations are atomic and directly depend on each others‘ result. The concurrency control of the transaction module is used to allow other, independent operations to interleave with such dependent series of operations (Table 4.1).

MDS Operation Type	Read	Update	Write
Read	Execute	Execute	Queue
Update	Execute	Queue	Queue
Write	Queue	Queue	Queue

Table 4.1: Transaction control module locking table of the symmetric active/active HPC parallel file system metadata service

Note that since all incoming state change requests from Transis are totally ordered and all MDS modules, including the scheduler module and the transaction control module, are deterministic, virtual synchrony is provided for all request messages. Query messages are *not* executed in virtual synchrony with request messages, since a single client always needs to wait for a completion of a request before issuing another query, a POSIX consistency requirement for file system access [186].

Failure Handling

Upon failure of an active service node, the Transis process group communication system informs all MDSs to exclude the failed service node from any further communication. Only those clients that are directly connected to the failed service node are affected and initiate a connection fail-over. After reconnecting to an active service group member, the

previously produced output message cached at the service group member is resent and may be ignored by the client if duplicated.

There is no service failover necessary as the healthy active service nodes continue to provide the service. Since Transis is able to deal with multiple simultaneous failures in the same way it deals with multiple sequential failures, the parallel file system metadata service is provided transparently as long as one service node survives.

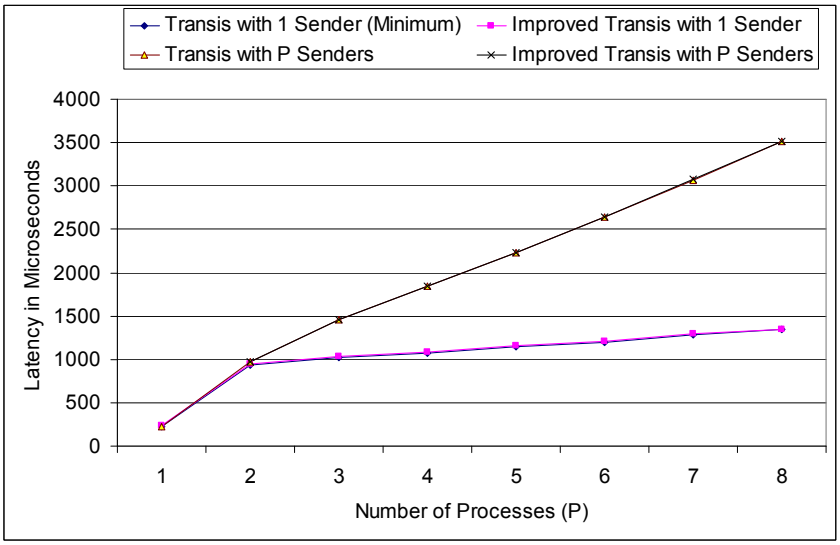
Similarly to the JOSHUA solution (Section 4.2), service nodes may be forcibly removed from the active service group for replacement and new service nodes may be added to the active service group. State transfer from an active service group member to a new service is performed by a membership management module inside the MDS.

4.3.4 Test Results

The fully functional proof-of-concept prototype implementation has been deployed on a dedicated Linux cluster for functional and performance testing. Each node contained dual Intel Pentium IV 2GHz processors with 768MB of memory and 40GB of disk space. All nodes were connected via a single Fast Ethernet (100MBit/s full duplex) hub. Federa Core 5 has been used as OS in conjunction with the modified Transis v1.03 and the modified PVFS v1.3.2. A MPI-based benchmark is used to perform and measure total message ordering by Transis using the fast delivery protocol, 5,000 messages per group member per test run. A second MPI-based benchmark is used to perform and measure concurrent MDS queries and requests from multiple clients, 5,000 MDS queries or requests per client per test run. Performance results are averages over 100 test runs with disabled client and service caches. Failures were simulated by unplugging network cables and by forcibly shutting down individual processes.

Fast Delivery Protocol

The improved Transis total message order latency performance (Figure 4.13 or Section A.1.2) shows an excellent performance in a testbed of 1-8 processes with 235-1,348μs for a single sender and 971-3,514μs when all group members send messages. The single sender tests resemble the idle case of the fast delivery protocol, while the all group members tests resemble scenarios with heavy load. The latency is consistent in scaling for both idle and busy systems. The fast acknowledgement algorithm aggressively acknowledges total order messages to reduce the latency of idle systems when only a single process is active. The protocol is smart enough to hold its acknowledgements when the network communication is heavy because more processes are involved.



Maximum of Transis 1 sender = heartbeat interval, *e.g.*, ≈500,000 microseconds

Figure 4.13: Latency performance of the fast delivery protocol (averages over 100 tests)

Compared to the original communication history algorithm of the Transis process group communication system, the post-transmission delay impact is apparent. The original Transis total message order latency performance is not consistent and its maximum is equivalent to the heartbeat interval in the worst case scenario. The heartbeat interval of Transis can be adjusted, but is typically in the order of several hundred milliseconds. The latency of the fast delivery protocol is almost exactly the same as the minimum measured with the original Transis. The fast delivery protocol provides total message order with consistent optimal performance.

Symmetric Active/Active Metadata Service

The MDS request latency performance (Figure 4.14 or Section A.1.2) is remarkably good. The overhead introduced by the network communication between the PVFS service and the Transis process group communication service is within 2-26ms when comparing all tested configurations. The significant latency performance overhead reduction to the previous JOSHUA solution (Section 4.2) can be attributed to the internal replication approach, the improved Transis process group communication algorithm using the fast delivery protocol, and to the fine grain synchronisation of state changes performed by the scheduler and transaction control modules of the symmetric active/active PVFS MDS proof-of-concept prototype.

It is also noted that the request latency performance scales quite well to 32 clients with an only slightly increasing absolute overhead, a drastically decreasing relative overhead.

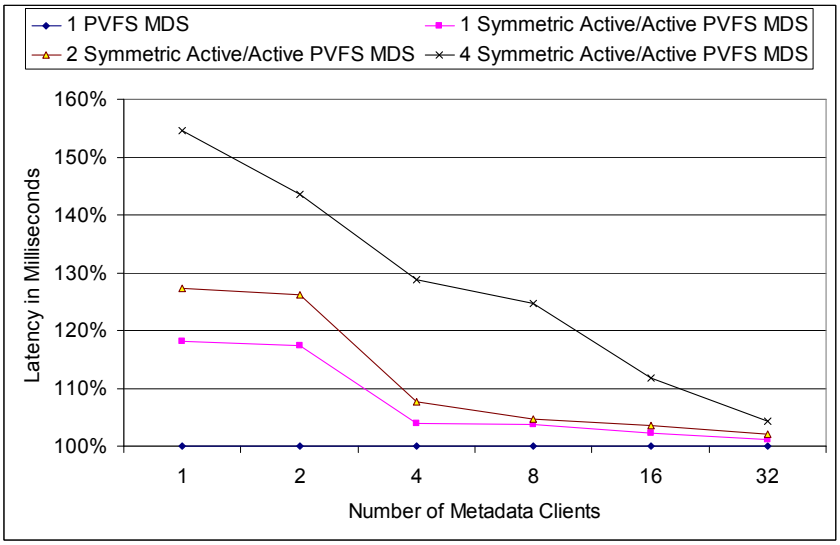


Figure 4.14: Normalised request latency performance of the symmetric active/active HPC parallel file system metadata service (averages over 100 tests)

The MDS query latency performance was not tested as queries are processed by the the symmetric active/active PVFS MDS proof-of-concept prototype almost exactly as by the baseline PVFS MDS. The query throughput performance tests (Figure 4.15 or Section A.1.2) not only verify this argument, but also show the throughput performance of 300-380% of PVFS MDS baseline performance in a four service node system due to load balancing and local processing of queries. Note that the query throughput does not scale linear with the number of active service nodes due to the sharing of the network.

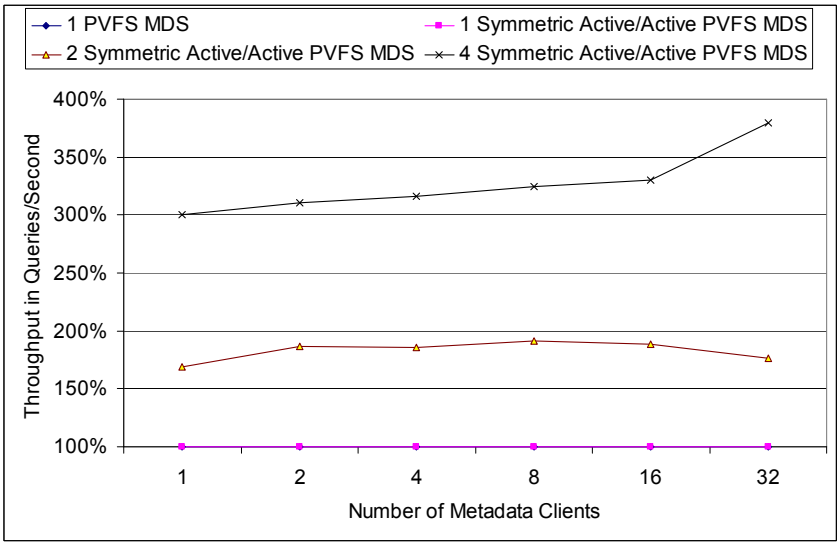


Figure 4.15: Normalised query throughput performance of the symmetric active/active HPC parallel file system metadata service (averages over 100 tests)

The MDS request throughput performance (Figure 4.16 or Section A.1.2) shows the impact of the Transis process group communication system. Throughput drops down to 79% of PVFS MDS baseline performance in a two service node system, while it drops down to 73% in a four service node system. However, the relative difference between all configurations becomes much smaller the more PVFS MDS clients are involved.

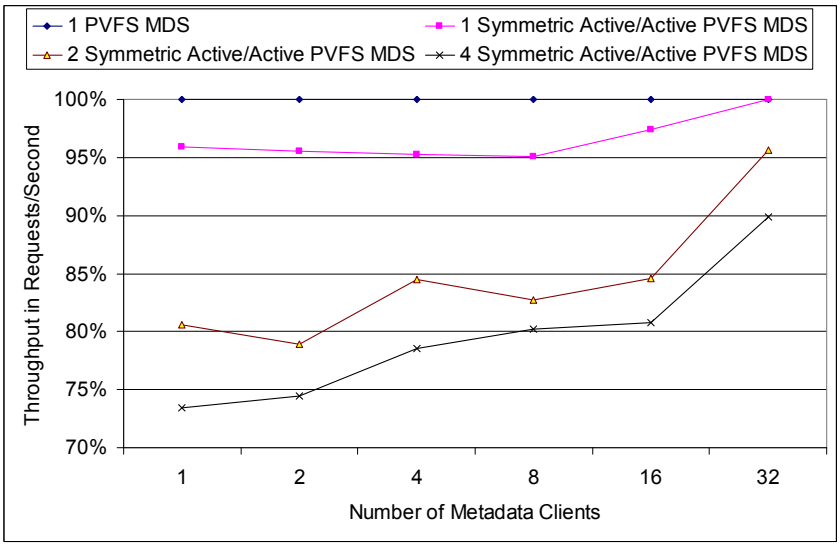


Figure 4.16: Normalised request throughput performance of the symmetric active/active HPC parallel file system metadata service (averages over 100 tests)

Extensive functional testing revealed correct behaviour during normal system operation and in case of single and multiple simultaneous failures. Service nodes were able to join the service group, leave it voluntary, and fail, while MDS state was maintained consistently at all service nodes and service was provided transparently to applications.

Although the acknowledgement scheme for total message ordering within the Transis process group communication system has been changed, the heartbeat interval is still responsible for detecting member failures through a communication timeout. The $MTTR_{recovery}$ of the previous proof-of-concept prototype (Section 4.2, 500 milliseconds) remained unchanged with this proof-of-concept prototype. Similarly, the provided availability remained unchanged with this proof-of-concept prototype as well.

4.3.5 Conclusions

With the symmetric active/active PVFS MDS proof-of-concept prototype, a second fully functional solution for providing symmetric active/active high availability for a HPC system service was developed. This is the first prototype that uses the internal replication approach, which modifies an existing service to inferface it with a process group commu-

nication system for virtual synchrony. The prototype performed correctly and offered a remarkable latency and throughput performance. Significant experience was added with respect to architecture, design, and performance challenges for symmetric active/active replication in HPC environments. The developed proof-of-concept prototype provided an extraordinary availability.

Since the MDS of a parallel file system is very response-latency sensitive, the latency performance results are not only convincing, but also encouraging for widely deploying symmetric active/active replication infrastructures in HPC systems to re-enforce critical HPC system services with high-performance redundancy. The developed fast delivery protocol with its low-latency total order messaging performance was instrumental to the success for this proof-of-concept prototype.

While the internal replication approach provides very good performance, it requires modification of existing services. The PVFS MDS is an easy-to-modify solution due to the small size and complexity of its clients and service. This may not be true with other services, such as the MDS of the Lustre cluster file system (Section 2.1.1).

4.4 Transparent Symmetric Active/Active Replication Framework for Services

The two developed symmetric active/active replication proof-of-concept prototypes, for the HPC job and resource management service (Section 4.2) and for the HPC parallel file system metadata service (Section 4.3), are fully functional solutions, provide adequate performance, and offer extraordinary service availability. However, there is an insufficient reuse of code between individual prototype implementations. Each service requires a customised symmetric active/active environment, either externally using interceptors or internally using adaptors.

The research and development effort presented in this Section targets a transparent symmetric active/active replication software framework for service-level high availability. As part of this effort, a preliminary proof-of-concept prototype [208] has been developed in C on Linux to provide symmetric active/active high availability transparently to services with a minimum amount of adaptation and performance overhead.

4.4.1 Objectives

With the exception of the Transis group communication system, there is an insufficient reuse of code between the two developed symmetric active/active replication proof-of-

concept prototypes (Sections 4.2 and 4.3). The development of this preliminary proof-of-concept prototype had the following primary objectives:

- development of a symmetric active/active replication framework that provides for more reuse of code between individual implementations,
- development of interfaces and mechanisms to transparently provide symmetric active/active replication to services with a minimum amount of adaptation,
- measuring any additional overhead introduced by the developed symmetric active/active replication framework and its mechanisms, and
- gaining further experience with service-level symmetric active/active replication.

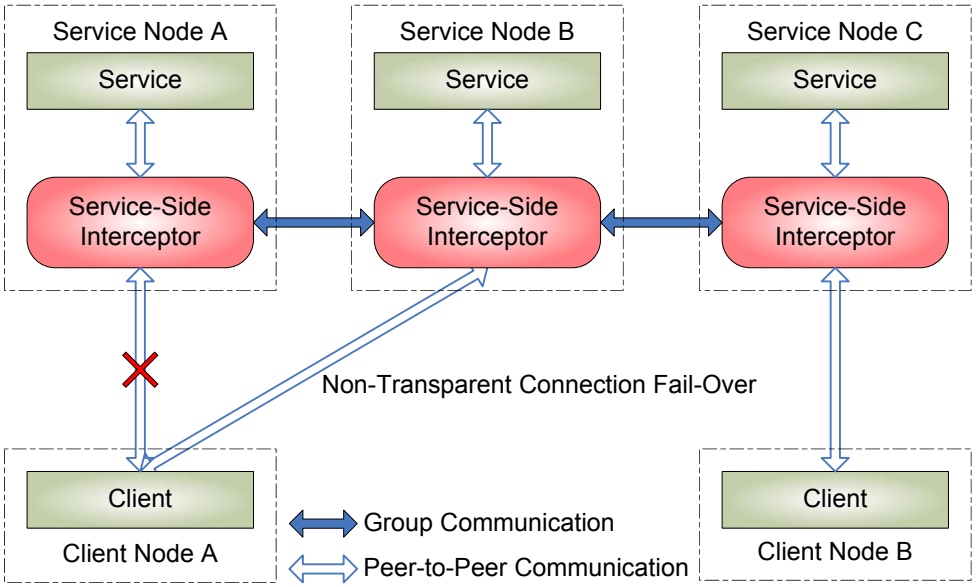
This proof-of-concept prototype focuses on the underlying symmetric active/active replication framework and not on a specific service. With the experience from the previously developed prototypes (Sections 4.2 and 4.3), interactions of the symmetric active/active replication framework with services are generalised.

4.4.2 Technical Approach

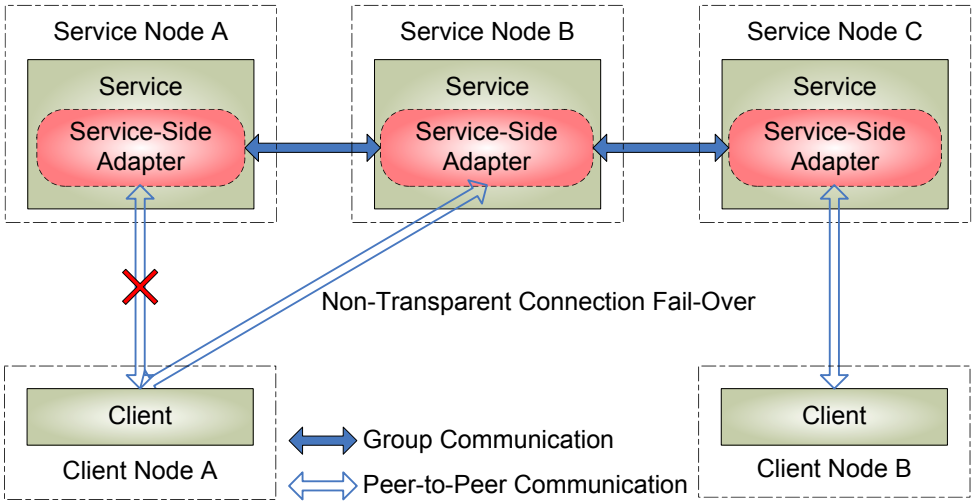
In the external or internal symmetric active/active replication software architecture, a service-side interceptor or adaptor component deals with receiving incoming request messages and routing them or related state changes through the process group communication system for total message order and reliable message delivery (Figure 4.17).

The service-side interceptor or adaptor also routes output back to the client, which interacts with the service-side interceptor or adaptor component instead with the original service. In case of a failure, clients need to be reconfigured in order to interact with the service-side interceptor or adaptor component of another member of the active service group. This requires clients to be informed about service group membership and to perform a consistent connection fail-over in case of a failure. Clients need to be made aware of the service-level symmetric active/active replication technique and need to be modified for external or internal service-side replication.

The main idea behind the service-side interceptor/adaptor concept of the external/internal symmetric active/active replication software architecture is to hide the interaction of the service with the group communication system from the service. While the internal replication method tightly integrates the service with the service-side adaptor, the external replication method utilises the service interface. In both cases, the client interacts with the service-side interceptor/adaptor.



(a) External



(b) Internal

Figure 4.17: Symmetric active/active replication software architecture with non-transparent client connection fail-over

The transparent symmetric active/active replication software framework accommodates both replication methods, external and internal, by using a virtual communication layer (VCL). The original external and internal symmetric active/active replication methods are refined to utilise service- and client-side interceptors/adaptors in order to provide total transparency. Adaptation of these interceptors/adaptors to clients and services is only needed with regards to the used communication protocols and its semantics. Clients and services are unaware of the symmetric active/active replication infrastructure as it provides all necessary mechanisms internally.

4.4.3 Architecture and Design

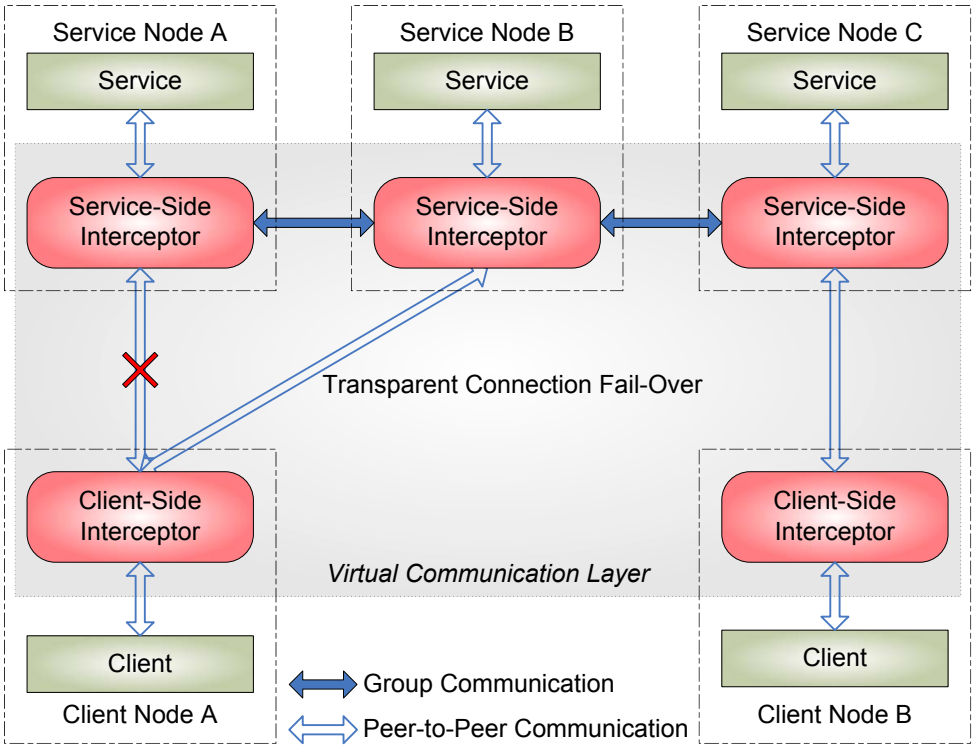
In the refined symmetric active/active replication software architecture with transparent client connection fail-over (Figure 4.18), an additional client-side interceptor/adaptor hides the interaction of the client with the service-side interceptor/adaptor from the client in the same fashion the service-side interceptor/adaptor hides the interaction of the service with the service-side interceptor/adaptor.

Similar to the service-side interceptor/adaptor, the client-side interceptor may be implemented externally by utilising the service interface at the client-side, and the client-side adaptor may be implemented internally by tightly integrating it with the client. In both cases, the client recognises the client-side interceptor/adaptor as the service.

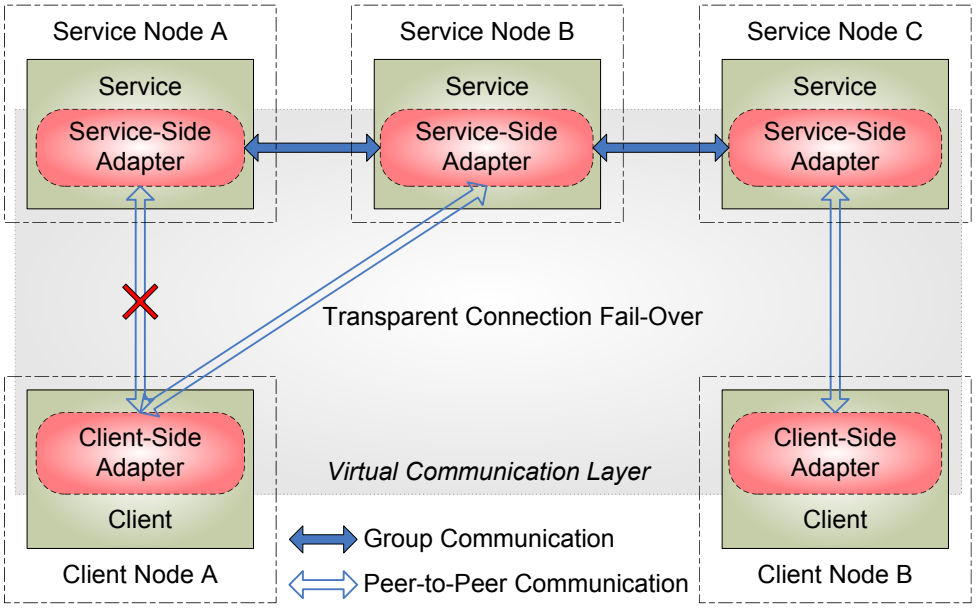
The client- and service-side interceptors/adaptors maintain a VCL, which client and service are unaware of. In fact, the client is only aware of a connection to a local service represented by the client-side interceptor/adaptor, while the service is only aware of a connection to a local client represented by the service-side interceptor/adaptor.

External interceptors offer transparency at the networking layer, *i.e.*, the client-side interceptor process establishes a local interceptor service at a certain network port that acts like the replicated service and the service-side interceptor process acts as a network client for the replicated service. Internal adapter libraries, however, provide a certain level of transparency at the messaging layer, *i.e.*, calls to network functions are intercepted/replaced with calls to adaptor library functions mimicking network function behaviour at the client and service side.

The VCL enforces the needed process group communication semantics at the service side as well as at the client-side based on the existing symmetric active/active replication software architecture. In addition to maintaining the previous process group communication role of service-side interceptors/adaptors of performing total message ordering, the VCL assures that client-side interceptors/adaptors are informed about service group membership and perform a consistent connection fail-over in case of a failure.



(a) External



(b) Internal

Figure 4.18: Symmetric active/active replication software architecture with transparent client connection fail-over

Figure 4.19 shows an example that explains how the VCL would provide full transparency to clients and services in the external replication design of the symmetric active/active HPC job and resource management service proof-of-concept prototype (for the original non-transparent design see Figure 4.4 in Section 4.2). The interceptor process on client nodes together with the Transis group communication service, the JOSHUA service, and the distributed mutual exclusion client scripts on each redundant head node form the VCL. Full transparency is provided as clients recognise the interceptor process on client nodes as their local HPC job and resource management service, while the HPC job and resource management service on each redundant head node recognises the JOSHUA service as their local client.

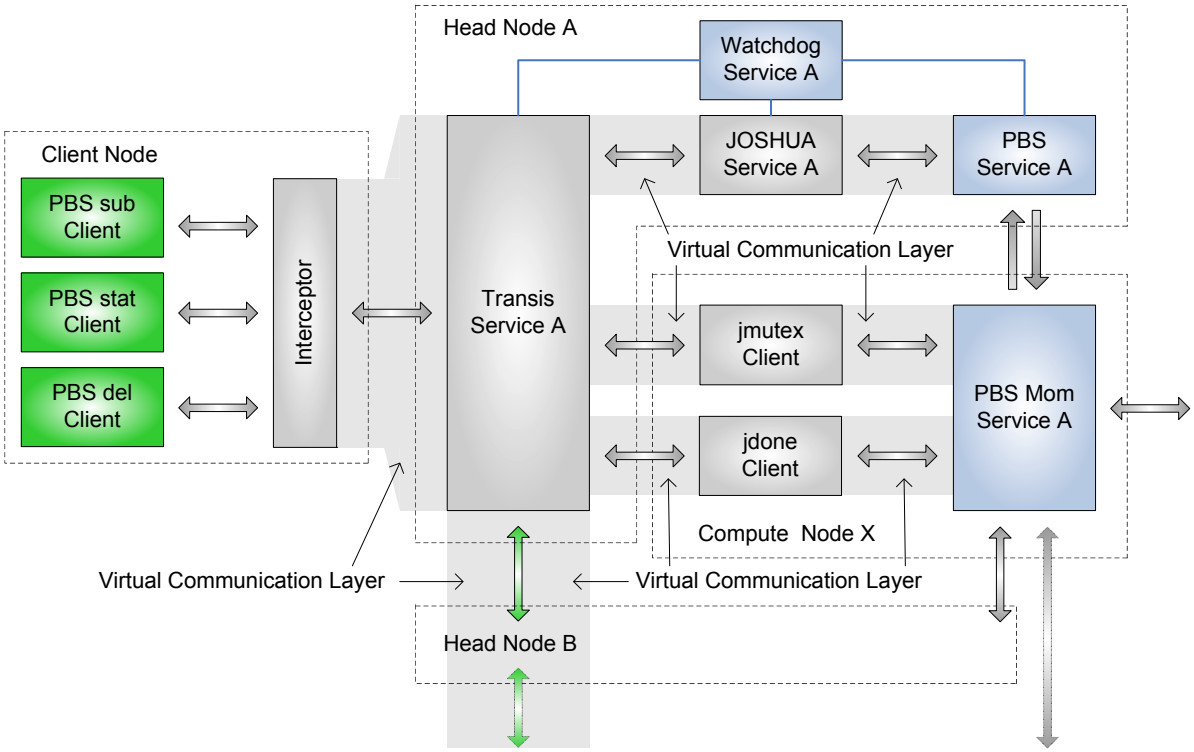


Figure 4.19: Example: Transparent external replication design of the symmetric active/active HPC job and resource management service

Figure 4.20 shows an example that explains how the VCL would provide partial transparency to clients and services in the internal replication design of the symmetric active/active HPC parallel file system metadata system service proof-of-concept prototype (for the original non-transparent design see Figure 4.11 in Section 4.3). The adaptors for the PVFS metadata clients on client nodes together with the Transis group communication service and the adaptor for the PVFS metadata service on each redundant head node form the VCL. Transparency is provided as clients recognise their adaptor on client

nodes as their local PVFS metadata service, while the PVFS metadata service on each redundant head node recognises its adaptor as local client. However, transparency is only partial as client and service need to be modified to interact with their adaptors.

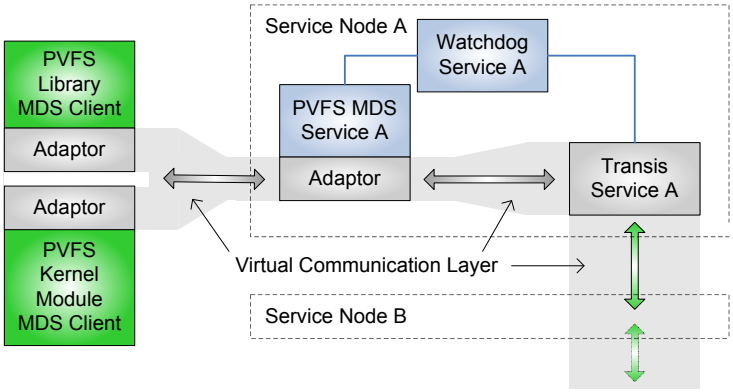


Figure 4.20: Example: Transparent internal replication design of the symmetric active/active HPC parallel file system metadata service

Failure Handling

In addition to the fault-tolerant group communication mechanisms handled at the service side (Section 3.3.4), the VCL provides communication fail-over for client connections in a transparent fashion, *i.e.*, clients are unaware of the failure of a service.

Upon initial connection to a service-side interceptor/adaptor, the client-side interceptor/adaptor receives the current list of group members. All client-side interceptors/adaptors are notified about membership changes by the service-side interceptor/adaptor they are connected to after the service group members agree.

A client-side interceptor/adaptor that detects a failure of its service-side interceptor/adaptor performs a connection fail-over to another service-side interceptor/adaptor based on its current list of group members. After reconnection, a recovery protocol retrieves any undelivered messages cached at the service group.

The connection between client- and service-side interceptors/adaptors uses message sequencing and acknowledgements in both directions to assure reliable message delivery. While the message sequencing assures that already received messages can be ignored, acknowledgements are used in certain intervals to clear cached messages at the service group. However, acknowledgements from the client-side interceptor/adaptor are interleaved with request messages, reliably multicast, and totally ordered, such that each service-side interceptor/adaptor is able to maintain a consistent message cache for service-side output messages in order to perform the connection fail-over transparently.

In case of a connection fail-over, all cached messages are resent by the new service-side interceptor/adaptor in the same order and duplicated messages are ignored accordingly to the sequence number of the last received message before the failure.

4.4.4 Test Results

Introducing external client-side interceptors into the communication path of a client-service system inherently results in a certain performance degradation. The introduction of internal client-side adaptors does not significantly affect performance.

A test mockup of the preliminary proof-of-concept prototype implementation has been deployed on a dedicated Linux cluster for performance testing. Each node contained dual Intel Pentium IV 2GHz processors with 768MB of memory and 40GB of disk space. All nodes were connected via a single Fast Ethernet (100MBit/s full duplex) hub. Federa Core 5 has been used as OS. A simple benchmark is used to perform and measure emulated RPC patterns by sending a payload to the service and waiting for its return. The RPCs are symmetric, *i.e.*, requests and responses have the same payload.

The tests do not include any process group communication system as its performance impact is service dependent, *e.g.*, on message size, and has been studied in the previous proof-of-concept prototype for providing symmetric active/active replication for the MDS of the parallel file system (Section 4.3). The process group communication system latency of 235-3,514μs for 1-8 members can be simply added to the measured RPC latency for a generic comparison, since the tests are performed in the same testbed and interceptor communication and process group communication do not interfere with each other. Performance results are averages over 100 test runs.

The message ping-pong, *i.e.*, emulated RPC, latency performance tests (Figure 4.21 or Section A.1.3) of the transparent symmetric active/active replication framework proof-of-concept prototype using external replication show that performance decreases with more interceptors in the communication path. The performance penalty for small payloads (0.1kB) for using client- and service-side interceptors can be as high as 22% in comparison to an unmodified client/service system, in contrast to the penalty of 11% when using service-side interceptors only. However, the performance impact dramatically decreases with increasing payload.

With a latency performance of 150-178μs for 0-2 interceptors using small payloads (0.1kB), the previously measured process group communication system latency of 235-3,514μs for 1-8 members is clearly the dominant factor. However, with a latency performance of 1900-2000μs for 0-2 interceptors using bigger payloads (10kB), the process group communication system latency becomes equal or less important.

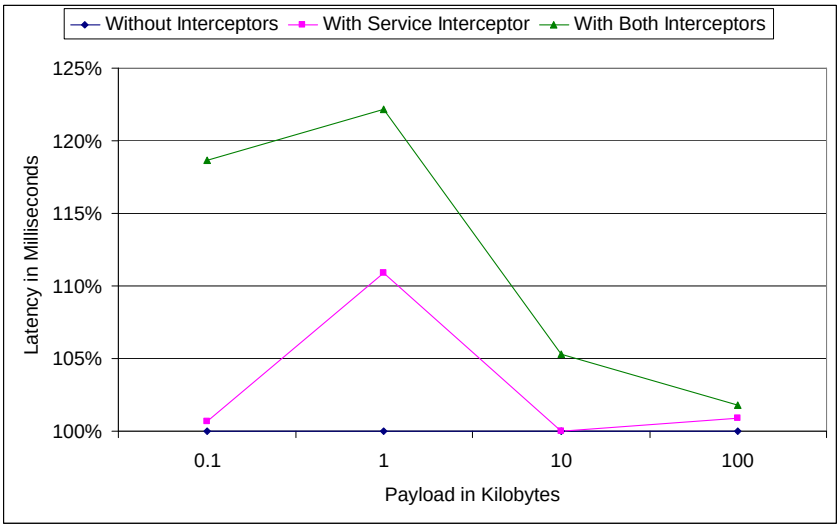


Figure 4.21: Normalised message ping-pong (emulated remote procedure call) latency performance of the transparent symmetric active/active replication framework using external replication (averages over 100 tests)

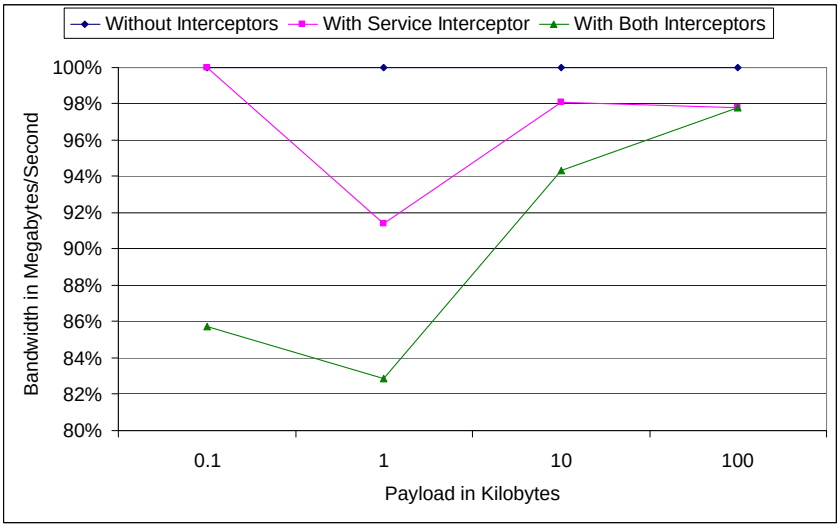


Figure 4.22: Normalised message ping-pong (emulated emulated remote procedure call) bandwidth performance of the transparent symmetric active/active replication framework using external replication (averages over 100 tests)

The message ping-pong bandwidth performance tests (Figure 4.22 or Section A.1.3) also show that performance decreases with more interceptors in the client-service communication path. The performance for small payloads (0.1kB) for using client- and service-side interceptors can be as low as 83% of an unmodified client/service system, in contrast to the 91% when using service-side interceptors only. However, the performance impact also dramatically decreases with increasing payload.

Both, latency and bandwidth performance are also highly dependent on the request/query processing latency of a service. The performed tests assume a request/query processing latency of 0μs, which is unrealistic but represents the worst-case scenario for the symmetric active/active replication infrastructure. The higher the request/query processing latency, the lesser is the impact of the latency and bandwidth performance of the symmetric active/active replication infrastructure.

4.4.5 Conclusions

With this preliminary proof-of-concept prototype, a symmetric active/active replication software architecture has been designed that uses a VCL to accommodate both replication methods, internal and external, and to allow for transparent client connection fail-over as well as for more reuse of code between individual service-level replication implementations.

With the introduction of client-side interceptors/adaptors in addition to those on the service side, the developed solution additionally hides the interaction of the client with the service-side interceptor/adaptor from the client in the same fashion the service-side interceptor/adaptor hides the interaction of the service with the service-side interceptor/adaptor. Adaptation of interceptors/adaptors to clients and services is only needed with regards to the used communication protocol and its semantics. Clients and services are unaware of the symmetric active/active replication infrastructure as it provides all necessary mechanisms internally via the VCL.

The developed symmetric active/active replication prototype is able to transparently provide service-level high availability using the symmetric active/active replication approach for services in parallel and distributed computing systems. It is applicable to any service-oriented or service-dependent architecture.

The transparency provided by the VCL also hides any communication across administrative domains, *i.e.*, communication appears to be local. This has two consequences. First, client and service still need to perform any necessary authentication and authorisation using the client- and service-side interceptors/adaptors as virtual protocol routers. Second, the VCL itself may need to perform similar mechanisms to assure its own integrity across administrative domains.

The experience gained with this preliminary proof-of-concept prototype is important with respect to applying symmetric active/active replication to other HPC system services. The performance results can be used as a guideline for choosing between the internal and external replication method depending on the performance requirements and architectural complexity of a specific service.

4.5 Transparent Symmetric Active/Active Replication Framework for Dependent Services

The previously developed software framework (Section 4.4) offers transparent symmetric active/active replication. However, this preliminary proof-of-concept prototype is limited to client-service scenarios. While they are quite common, as exemplified with the symmetric active/active replication proof-of-concept prototype for the HPC parallel file system metadata service (Section 4.3), dependencies between critical HPC system services exist.

The Lustre cluster file system (Section 2.1.1), for example, employs a MDS as well as object storage services (OSSs). File system drivers on compute nodes communicate in a client-service fashion with these services. However, in contrast to PVFS (Sections 2.1.1 and 4.3), MDS and OSSs communicate with each other in a service-to-service fashion incompatible with the current client-service replication architecture.

The previously developed symmetric active/active replication proof-of-concept prototype for the HPC job and resource management service (Section 4.2) also showed exactly this deficiency, as the job management service on the head node of a HPC system, *i.e.*, the PBS service, and the parallel job start and process monitoring service on the compute nodes, *i.e.*, the PBS mom service, depend on each other to function correctly.

The developed preliminary proof-of-concept prototype of a transparent symmetric active/active replication framework does not clearly address dependent services. The research and development effort presented in this Section targets the extension of the developed preliminary proof-of-concept prototype to replication scenarios with dependent services using its already existing mechanisms and features. As part of this effort, a second preliminary proof-of-concept prototype [209] has been developed in C on Linux to provide symmetric active/active high availability transparently to dependent services.

This preliminary proof-of-concept prototype was primarily motivated by the experience gained from an attempt to apply the previously developed software framework to the Lustre cluster file system metadata service. A preliminary proof-of-concept prototype implementation in C on Linux was carried out under my supervision by Matthias Weber during his internship at Oak Ridge National Laboratory, USA, for his MSc thesis at the

University of Reading, UK. His thesis [210], titled "High Availability for the Lustre File System", revealed the incompatibility of the client-service oriented transparent symmetric active/active replication framework with the service interdependencies in Lustre. The following approach for a transparent symmetric active/active replication framework for dependent services was a direct result of this experienced limitation.

4.5.1 Objectives

The developed transparent symmetric active/active replication software framework only addresses client-service scenarios, but more complex service-service relationships between critical HPC system services exist. The development of this preliminary proof-of-concept prototype had the following primary objectives:

- extension of the developed symmetric active/active replication framework to provide high availability to dependent services,
- utilisation of existing mechanisms and features to avoid an unnecessary increase in framework size and complexity,
- measuring any additional overhead introduced by the extended symmetric active/active replication framework in scenarios with dependent services, and
- gaining further experience with service-level symmetric active/active replication.

This proof-of-concept prototype focuses on the underlying transparent symmetric active/active replication framework and not on a specific service. With the experience from the previously developed prototype for the HPC job and resource management service (Section 4.2) and based on the architecture of the Lustre cluster file system (Section 2.1.1), interactions of the symmetric active/active replication framework in scenarios with dependent services are generalised.

4.5.2 Technical Approach

Two networked services depend on each other, when one service is a client of the other service or when both services are clients of each other. More than two services may depend on each other through a composition of such service-to-service dependencies.

In the previously introduced example of the Lustre cluster file system architecture, MDS and OSSs are not only services for the file system driver client on the compute nodes of a HPC system, but also clients for each other. Assuming n compute, 1 MDS and m OSS nodes, this architecture consists of a n-to-1 dependency for file system driver and MDS,

a n-to-m dependency for file system driver and OSSs, a m-to-1 dependency for OSSs and MDS, and a 1-to-m dependency for MDS and OSSs.

In order to deal with such dependencies between services, the existing transparent symmetric active/active replication framework is extended to service-service scenarios by using its already existing mechanisms and features for client-service systems. While each interdependency between two services is decomposed into two respective orthogonal client-service dependencies, services utilise separate client-side interceptors/adapters for communication to services they depend on.

4.5.3 Architecture and Design

A high-level abstraction of the existing transparent symmetric active/active replication framework architecture helps to illustrate dependencies between clients and services, and to decompose dependencies between services into respective client-service dependencies. With the help of the VCL that hides the replication infrastructure as much as possible, the framework architecture can be simplified into five components: nodes, clients, services, VCLs, and connections (Figure 4.23). This high-level abstraction is independent of the replication method, internal or external, while it clearly identifies clients, services, replicas, client-service dependencies, and decomposed service-to-service dependencies.

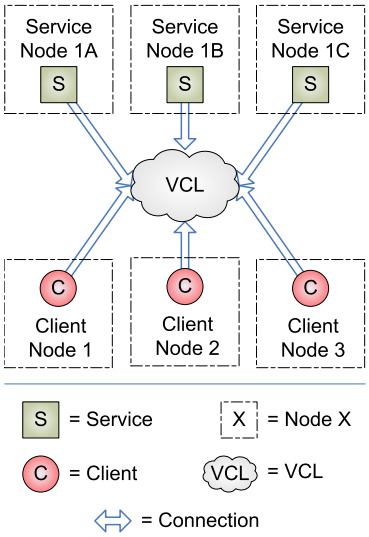


Figure 4.23: Transparent symmetric active/active replication framework architecture for client/service scenarios

In this abstraction, a node represents some specific service state. Each node may host only one service and multiple clients, where clients on a node that hosts a service belong to that service. Each VCL belongs to one group of replicated services and their clients. Any client or service may belong to only one VCL.

The notion of node in this abstraction may not directly fit to real-world applications as a single physical service node may host multiple services, like the head node in a HPC system. This may be noted in the abstraction by using a node for each service that runs independently on a physical node. Multiple services that do depend on each other and run on the same physical node are considered as one service, since they are replicated as one single deterministic state machine.

Service-independent clients may be noted by using a node for each client. Independent clients may not be grouped together on a node in the abstraction, even if they reside on the same physical node. This allows to differentiate between the states of clients.

For the respective VCL they are connected to, clients utilise their client-side interceptor/adaptor and services utilise their service-side interceptor/adaptor. The client-side interceptors/adaptors of a replicated service form a single virtual client in the VCL layer, thus allowing client-side interceptors/adaptors of clients or of non-replicated services to connect to the same VCL.

Based on this high-level abstraction, a variety of scenarios can be expressed using the existing transparent symmetric active/active replication framework.

Figure 4.24 depicts a scenario with a group of clients, client nodes 1-3, accessing a replicated service group, service nodes 1A-C, which itself relies on another replicated service group, service nodes 2X-Z. The service nodes 1A-C provide a replicated service to client nodes 1-3, while they are clients for service nodes 2X-Z. Each of the two VCLs performs the necessary replication mechanisms for its replicated service.

The scenario depicted in Figure 4.24 solves the issue of the HPC job and resource management service. Services 1A-C represent the replicated job management service process running on HPC system head nodes and services 2X-Z represent the replicated parallel job start and process monitoring service process running on HPC system compute nodes. Both services can be fully and transparently replicated by using serial VCLs to hide the replication infrastructure from both services.

Figure 4.25a shows a group of clients, client nodes 1-3, communicating with two different replicated service groups, service nodes 1A-B and 2Y-Z. The client nodes 1-3 host two clients, each for a different VCL belonging to a different replicated service group.

Figure 4.25b illustrates a scenario with two interdependent replicated service groups. The service nodes 1A-B provide a replicated service and are clients of 2Y-Z, while the service nodes 2Y-Z provide a replicated service and are clients of 1A-B.

The presented high-level abstraction can be used to guide the deployment of the transparent symmetric active/active replication framework in more complex scenarios with dependent HPC system services using one of the presented scenarios or a combination of the presented scenarios.

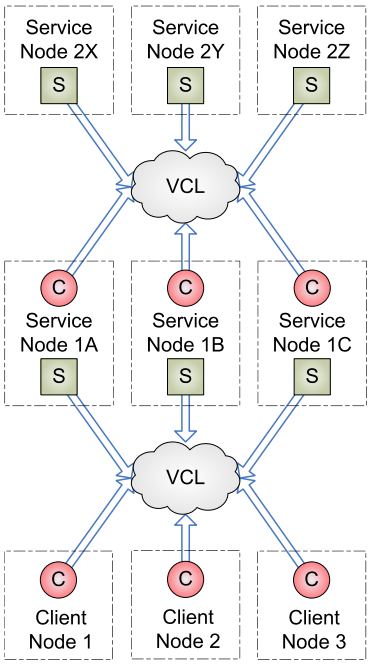
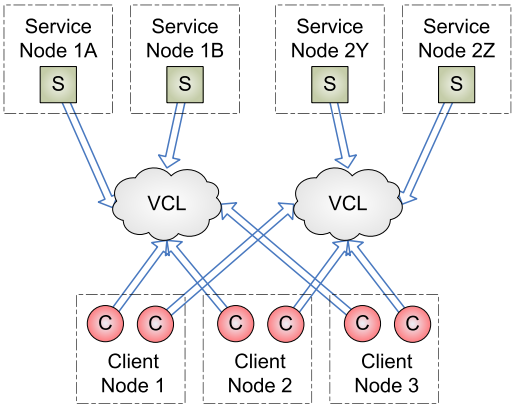
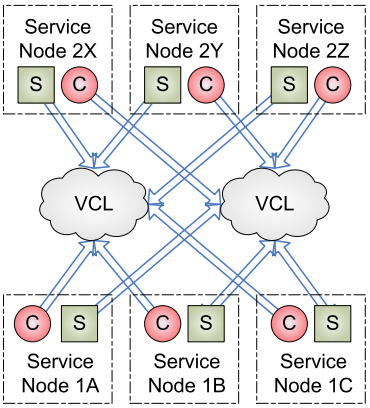


Figure 4.24: Transparent symmetric active/active replication framework architecture for client/client+service/service scenarios



(a) Client/2 services scenarios



(b) Service/service scenarios

Figure 4.25: Transparent symmetric active/active replication framework architecture for client/2 services and service/service scenarios

Going back to the Lustre cluster file system example, Figure 4.26 illustrates a highly available Lustre deployment using this high-level abstraction. This example uses 3 file system driver clients (client nodes 1-3), a replicated MDS service group, MDS nodes W-X, and one replicated OSS service group, OSS nodes Y-Z. This architecture is a combination of a group of clients communicating with two different replicated service groups (Figure 4.25a) and two interdependent replicated service groups (Figure 4.25b). Since Lustre supports many clients and several OSSs, the depicted example may be extended with respective components if needed.

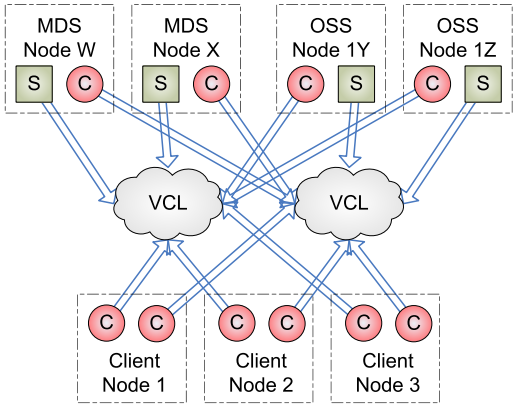


Figure 4.26: Example: Transparent symmetric active/active replication framework architecture for the Lustre cluster file system

4.5.4 Test Results

This enhancement of the previously developed preliminary proof-of-concept prototype of a symmetric active/active replication framework introduces two new basic replication configurations: (1) multiple serial VCLs, *i.e.*, nodes with a service and one or more clients (Figure 4.24), and (2) multiple parallel VCLs, *i.e.*, nodes with multiple clients (Figure 4.25a). Service-to-service scenarios (Figure 4.25b) are a combination of both.

In addition to the normal interference between multiple services communicating with their clients at the same time over the same network, multiple parallel VCLs may interfere with each other if the process group communication traffic is routed through the same network, *i.e.*, via the same network switch. Since this interference is highly application dependent, *e.g.*, bandwidth usage, collisions, and network quality of service, a generic performance evaluation does not make much sense. Furthermore, a separation of process group communication traffic for additional performance may be implemented by deploying a separate network between replicated service nodes.

Multiple serial VCLs interfere with each other by adding latency in a request/response scenario commonly used by HPC system services in the form of RPCs.

Reiterating the Lustre cluster file system example, the file system driver client on the compute nodes communicates with the MDS, which in turn communicates with OSSs. This scenario can be observed when deleting a file, where the MDS deletes the file record and notifies the OSSs to delete the file object before returning a response back to the file system driver client that initiated the file deletion request.

Since some HPC system services, such as the set of services deployed by Lustre, tend to be response-latency sensitive, tests were performed with a generic client/service/service architecture to measure the performance impact of using interceptor processes, *i.e.*, external replication, in a serial VCL configuration.

Two test series were performed using (1) a single service and (2) two serial services, each with and without interceptor processes, *i.e.*, using external and internal replication, between client and service 1, and service 1 and service 2 (Figure 4.24).

A test mockup of the preliminary proof-of-concept prototype implementation has been deployed on a dedicated cluster for performance testing. Each node contained dual Intel Pentium D 3GHz processors with 2GB of memory and 210GB of disk space. All nodes were connected via a single Gigabit Ethernet (1GBit/s full duplex) hub. Federa Core 5 64bit has been used as OS. A simple benchmark is used to perform and measure emulated RPC patterns by sending a payload to the service and waiting for its return. The RPCs are symmetric, *i.e.*, requests and responses have the same payload. Performance results are averages over 100 test runs.

Similar to the previous preliminary proof-of-concept prototype implementation (Section 4.4), the tests do not include any process group communication system as its performance impact is service dependent and has been studied before (Section 4.3).

The message ping-pong, *i.e.*, emulated RPC, latency performance tests (Figure 4.27 or Section A.1.4) of the transparent symmetric active/active replication framework proof-of-concept prototype in a serial VCL configuration clearly show the increasing performance impact of adding interceptor processes into the communication path of dependent services. The highest latency performance impact can be observed with small message payloads. The performance penalty for small payloads (0.1kB) for using client and service-side interceptors can be as high as 85% in comparison to an unmodified serial client/service system, in contrast to the penalty of 35% when using service-side interceptors only. However, the relative performance impact dramatically decreases when increasing the payload to 100KB, after which it increases again.

The message ping-pong bandwidth performance tests (Figure 4.28 or Section A.1.4) also clearly show the increasing impact of adding interceptor processes into the communication path. Similar to the latency impact, the highest bandwidth performance impact is with small messages. The performance for small payloads (0.1kB) for using client and service-

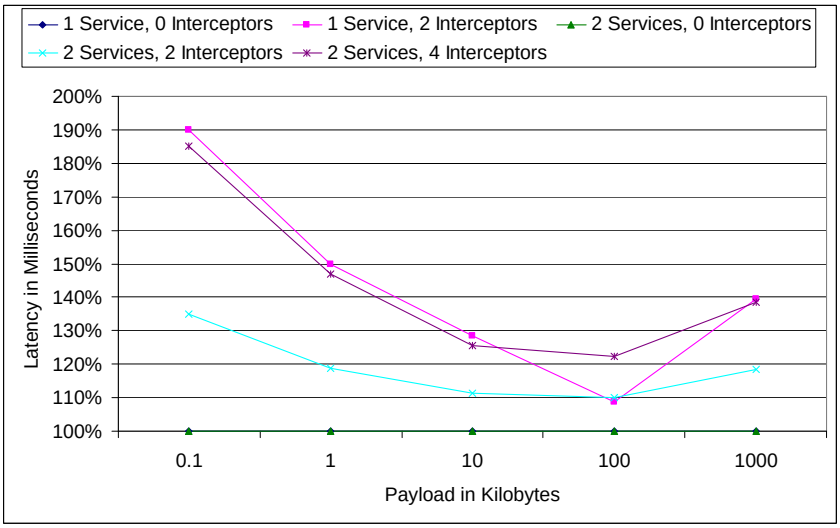


Figure 4.27: Normalised message ping-pong (emulated emulated remote procedure call) latency performance of the transparent symmetric active/active replication framework in a serial virtual communication layer configuration (averages over 100 tests)

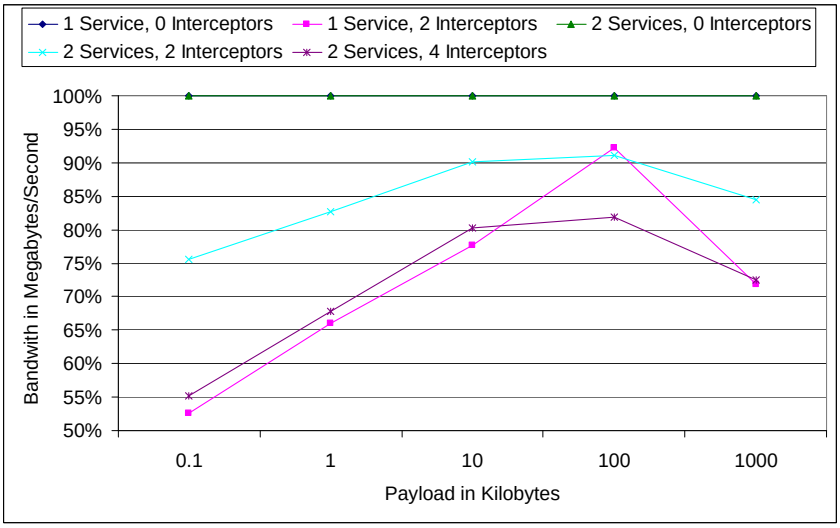


Figure 4.28: Normalised message ping-pong (emulated emulated remote procedure call) bandwidth performance of the transparent symmetric active/active replication framework in a serial virtual communication layer configuration (averages over 100 tests)

side interceptors can be as low as 55% of an unmodified serial client/service system, in contrast to the 76% when using service-side interceptors only. However, the relative performance impact also dramatically decreases when increasing the payload to 100KB, after which it also increases again.

There are two factors that influence both performance impacts: (1) latency added by the interceptor processes, and (2) local traffic, and respective congestion, added by the interceptor processes. Both are due to the fact that the interceptor processes cause network traffic to go twice through the OS, once to send/receive to/from the network and once to send/receive to/from its client or service.

Similar to the previous preliminary proof-of-concept prototype implementation (Section 4.4), latency and bandwidth performance are also highly dependent on the request/query processing latency of a service. The performed tests assume the worst-case scenario for the symmetric active/active replication infrastructure, a service request/query processing latency of 0μs.

4.5.5 Conclusions

With this preliminary proof-of-concept prototype, one important limitation of the previously developed preliminary proof-of-concept prototype of a transparent symmetric active/active replication framework has been addressed. Its deficiency, the inability to deal with dependent services, has been resolved by extending the replication framework using its already existing mechanisms and features to allow services to be clients of other services, and services to be clients of each other.

By using a high-level abstraction, dependencies between clients and services, and decompositions of service-to-service dependencies into respective orthogonal client-service dependencies can be mapped onto an infrastructure consisting of multiple symmetric active/active replication subsystems. Each subsystem utilises the VCL to hide the replication infrastructure for a specific service group as much as possible.

The enhanced preliminary proof-of-concept prototype is able to transparently provide high availability for dependent HPC system services by utilising existing mechanisms and features, thus avoiding an unnecessary increase in framework size and complexity.

The additional experience gained with this proof-of-concept prototype is important with respect to applying symmetric active/active replication to dependent HPC system services in practice. The performance results can be used as a guideline for choosing between the internal and external replication method depending on the performance requirements and architectural complexity of a specific service or service group.

4.6 Summary

This Chapter detailed objectives, technical approach, architecture, design, test results, and conclusions for each developed proof-of-concept prototype. First, a symmetric active/active high availability framework concept was outlined. This multi-layered framework concept coordinated individual solutions with regards to their respective field, and offered a modular approach that allows for adaptation to system properties and application needs. The four developed prototypes primarily focused on the virtual synchrony runtime environment of this framework concept as implementations of the other layers can be found in existing solutions. Future work may combine existing solutions with the virtual synchrony runtime environment presented in this Chapter.

The first prototype was the very first fully functional solution for providing symmetric active/active high availability for a HPC system service using the external replication approach that wraps an existing service into a virtually synchronous environment. It provided high availability for the TORQUE job and resource management service with an acceptable performance. A distributed mutual exclusion to unify replicated output was showcased. The developed proof-of-concept prototype provided an extraordinary service availability. With a node MTTF of 5,000 hours, service availability could be improved from 99.285% to 99.995% in a two-node system, an increase from a two-nines to a four-nines rating just by using a second node. Adding another node would increase service availability to 99.99996%, a six-nines rating. Experienced limitations included a certain latency performance overhead unacceptable for more latency-sensitive services, and complex interdependencies between individual system services on head, service and compute nodes incompatible with the client-service replication infrastructure.

The second prototype was a fully functional solution for providing symmetric active/active high availability for the MDS of PVFS using the internal replication approach, which modifies an existing service to inferface it with a process group communication system for virtual synchrony. The prototype offered a remarkable latency and throughput performance due to significant improvements in the process group communication total message order protocol. This improvement solved one of the experienced limitations of the first prototype. Similar to the previous prototype, this prototype provided an extraordinary service availability. While this particular MDS was an easy-to-modify solution due to the small size and complexity of its clients and service, other parallel file systems, such as Lustre, are rather complex.

The third prototype was a symmetric active/active replication software architecture that uses a VCL to accommodate both replication methods, internal and external, to allow for transparent client connection fail-over as well as for more reuse of code be-

tween individual service-level replication implementations. With the introduction of client-side interceptors/adaptors, clients and services were unaware of the symmetric active/active replication infrastructure provided by this preliminary proof-of-concept prototype as it supports all necessary mechanisms internally via the VCL. Adaptation of interceptors/adaptors to clients and services is only needed with regards to the used communication protocol and its semantics. This improvement solved the experienced limitation of the first two prototypes. The provided generic performance results can be used as a guideline for choosing between the internal and external replication method depending on the performance requirements or complexity of a specific service.

The fourth prototype removed another limitation, the inability to deal with dependent services. The third prototype was extended using its already existing mechanisms and features to allow services to be clients of other services, and services to be clients of each other. By using a high-level abstraction, this preliminary proof-of-concept prototype, in addition to normal dependencies between clients and services, maps decompositions of service-to-service dependencies into respective orthogonal dependencies between clients and services onto a replication infrastructure consisting of multiple symmetric active/active replication subsystems. Similar to the previous preliminary prototype, generic performance results are provided to be used as a guideline for choosing between the internal and external replication method depending on the performance requirements or complexity of a specific service or service group.

5 Summary, Conclusions, and Future Work

This Chapter concludes this thesis with an overall summary and evaluation of the presented research, and a discussion of future directions.

5.1 Summary

In the following, individual chapter summaries are reiterated to give an overall summary of the presented work.

Chapter 1 provided a more detailed description of the overall thesis research background, motivation, objectives, methodology, and contribution. The research background of scientific HPC and its significance to other science areas, such as climate dynamics, nuclear astrophysics, and fusion energy, has been explained. The trend toward larger-scale HPC systems beyond 100,000 computational, networking, and storage components and the resulting lower overall system MTTF has been detailed. The main objective of this thesis, efficient software state replication mechanisms for the redundancy of services running on HPC head and service nodes, has been stated and motivated by the fact that such service components are the "Achilles heel" of a HPC system.

The methodology and major research contributions of this thesis have been outlined with respect to theoretical ground work (see summary for Chapter 3) and proof-of-concept prototype development (see summary for Chapter 4).

Chapter 2 evaluated previous work within the research context of this thesis. Detailed past and ongoing research and development for HPC head and service node high availability included active/standby configurations using shared storage, active/standby configurations using software state replication, and high availability clustering. Certain pitfalls involving active/standby configurations using shared storage, such as backup corruption during failure, have been pointed out. There was no existing solution using symmetric active/active replication, *i.e.*, state-machine replication.

Described techniques for HPC compute node high availability included checkpoint/restart, message logging, algorithm-based fault tolerance, and proactive fault avoidance. The underlying software layer often relied on HPC head and service node high availability for coordination and reconfiguration after a failure.

Examined distributed systems efforts focused on state-machine replication, process group communication, virtual synchrony, distributed control, and practical Byzantine fault tolerance. Many of the distributed systems mechanisms were quite advanced in terms of communication protocol correctness and provided availability. There has been much less emphasis on high performance.

Detailed IT and telecommunication industry solutions covered a wide range of high availability configurations. The only two solutions using some variant of state-machine replication were T2CP-AR for TCP-based telecommunication services and Stratus Continuous Processing for DMR with hardware-supported instruction-level replication.

Chapter 3 provided the theoretical ground work of the research presented in this thesis. An extended generic taxonomy for service-level high availability has been presented that introduced new terms, such as asymmetric active/active and symmetric active/active, to resolve existing ambiguities of existing terms, such as active/active. The taxonomy also clearly defined the various configurations for achieving high availability of service and relevant metrics for measuring service availability. This extended taxonomy represented a major contribution to service availability research and development with respect to incorporating state-machine replication theory and resolving ambiguities of terms.

Current HPC system architectures were examined in detail and a more generalised architecture abstraction was introduced to allow the identification of availability deficiencies. HPC system services were categorised into critical and non-critical to describe their impact on overall system availability. HPC system nodes were categorised into single points of failure and single points of control to pinpoint their involvement in system failures, to describe their impact on overall system availability, and to identify their individual need for a high availability solution. This analysis of architectural availability deficiencies of HPC systems represented a major contribution to the understanding of high availability aspects in the context of HPC environments.

Using the taxonomy and a conceptual service model, various methods for providing service-level high availability were defined and their mechanisms and properties were described in detail. A theoretical comparison of these methods with regards to their performance overhead and provided availability was presented. This comparison represented a major contribution to service availability research and development with respect to incorporating state-machine replication theory. It clearly showed that symmetric active/active

replication provides the highest form of availability, while its performance impact highly depends on the employed process group communication protocol.

Chapter 4 detailed the developed proof-of-concept prototypes. First, a multi-layered symmetric active/active high availability framework concept was outlined that coordinated individual solutions with regards to their respective field and offered a modular approach for adaptation to system properties and application needs. The four developed proof-of-concept prototypes primarily focused on the virtual synchrony runtime environment as implementations of the other layers can be found in existing solutions.

The first proof-of-concept prototype was the very first fully functional solution for providing symmetric active/active high availability for a HPC system service using the external replication approach that wraps an existing service into a virtually synchronous environment. It provided extraordinary high availability for the TORQUE job and resource management service with an acceptable performance. With a node MTTF of 5,000 hours, service availability could be improved from 99.285% to 99.995% in a two-node system, an increase from a two-nines to a four-nines rating just by using a second node. A distributed mutual exclusion to unify replicated output was showcased. Experienced limitations included a certain performance overhead unacceptable for more latency-sensitive services, and complex interdependencies between individual system services on head, service and compute nodes incompatible with the client-service replication infrastructure.

The second proof-of-concept prototype was a fully functional solution for providing symmetric active/active high availability for the MDS of PVFS using the internal replication approach, which modifies an existing service to inferface it with a process group communication system for virtual synchrony. The proof-of-concept prototype offered a remarkable latency and throughput performance due to significant improvements in the process group communication protocol. This improvement solved the experienced limitation of the first proof-of-concept prototype, while it continued to provide an extraordinary service availability. While this particular MDS was an easy-to-modify solution due to the small size and complexity of its clients and service, other parallel file systems, such as Lustre, are rather complex.

The third proof-of-concept prototype was a symmetric active/active replication software architecture that uses a VCL to accommodate both replication methods, internal and external, to allow for transparent client connection fail-over as well as for more reuse of code. Clients and services were unaware of the symmetric active/active replication infrastructure provided by this preliminary proof-of-concept prototype as it supports all necessary mechanisms internally via the VCL. Adaptation of interceptors/adaptors to clients and services is only needed with regards to the used communication protocol. This

improvement solved an experienced limitation of the first two proof-of-concept prototypes. Provided generic performance results can be used as a guideline for choosing between the internal and external replication method depending on service performance requirements and architectural complexity.

The fourth proof-of-concept prototype removed another limitation, the inability to deal with dependent services. The third proof-of-concept prototype was extended using its already existing mechanisms and features to allow services to be clients of other services, and services to be clients of each other. By using a high-level abstraction, this preliminary proof-of-concept prototype, in addition to normal dependencies between clients and services, maps decompositions of service-to-service dependencies into respective orthogonal dependencies between clients and services onto a replication infrastructure consisting of multiple symmetric active/active replication subsystems. Generic performance results are provided to be used as a guideline for choosing between the internal and external replication method performance requirements and architectural complexity of a specific service or service group.

5.2 Conclusions

The theoretical ground work of this thesis research is an important contribution to a modern taxonomy for service-level high availability as well as a significant step in understanding high availability aspects in the context of HPC environments.

The introduction of the terms asymmetric and symmetric active/active replication resolved existing ambiguities, while it integrated state-machine replication theory with service-level high availability taxonomy. This provides the high availability research and development community with a common language across all high availability configurations and mechanisms.

The identification of critical and non-critical system services as well as of individual single points of failure and single points of control within a generalised HPC system architecture revealed several availability deficiencies and the need for high availability solutions for various service components, *i.e.*, for head and service nodes. This results in a greater awareness within the HPC community of such availability deficiencies and of needed high availability solutions.

The comparison of service-level high availability methods using a conceptual service model and a common taxonomy for employed algorithms, provided availability, and incurred performance overhead, showed that active/hot-standby and symmetric active/active replication provide the highest form of availability. In case of symmetric active/active replication, the performance impact highly depended on the employed process group com-

munication protocol. This comparison results in a greater awareness within the high availability research and development community about the various high availability methods and their properties.

The developed proof-of-concept prototypes documented in this thesis represent a giant leap forward in providing high availability for HPC system services as well as an important contribution in offering symmetric active/active high availability transparently to clients and services.

The symmetric active/active high availability proof-of-concept prototype for a HPC job and resource management service not only showed that high availability can be provided without modifying a service, but also demonstrated the giant advantage symmetric active/active replication offers with its extremely low failure recovery impact and extraordinary high availability. With a MTTF of 5,000 hours for a single head node, service availability was improved from 99.285% to 99.995% in a two-node system, and to 99.99996% with three nodes. This proof-of-concept prototype offers a convincing argument to the high availability community in general and to the HPC community in specific that symmetric active/active high availability is implementable and provides the highest degree of service availability.

The symmetric active/active high availability proof-of-concept prototype for a HPC parallel file system metadata service additionally showed that high availability can be provided in conjunction with high performance. While the original metadata service was modified for adaptation to a process group communication system, it was also enhanced with the capability to interleave non-concurrent state changes for better performance. The improvement of the process group communication protocol was instrumental to the displayed high performance. This proof-of-concept prototype offers a convincing argument to the high availability community in general and to the HPC community in specific that symmetric active/active high availability is simply better or at least equal to its active/standby and asymmetric active/active competitors.

The two preliminary proof-of-concept prototypes for a transparent symmetric active/active replication software framework for client-service and dependent service scenarios showed that transparent or semi-transparent deployment of the replication infrastructure completely or partially invisible to clients and services is possible. Based on the symmetric active/active high availability framework concept of this thesis, both preliminary proof-of-concept prototypes represent a path toward a production-type transparent symmetric active/active replication software infrastructure.

5.3 Future Work

Possible future work focuses on the following four research and development directions: (1) development of a production-type symmetric active/active replication software infrastructure, (2) development of production-type high availability support for HPC system services, (3) extending the transparent symmetric active/active replication software framework proof-of-concept prototype to support active/standby and asymmetric active/active configurations as well, and (4) extending the lessons learned and the prototypes developed to other service-oriented or service-dependent architectures.

The two preliminary proof-of-concept prototypes for a transparent symmetric active/active replication software framework for client-service and dependent service scenarios already show the path toward a production-type solution. Using the symmetric active/active high availability framework concept of this thesis, these preliminary proof-of-concept prototypes may be combined with other existing solutions to offer a production-type symmetric active/active replication software infrastructure. For example, the component-based framework of Open MPI (Section 2.2.3) that encapsulates communication drivers using interchangeable and interoperable components, the component-based framework for group communication mechanisms in Horus (Section 2.3.3), and the VCL of the symmetric active/active replication software framework proof-of-concept prototype for dependent service scenarios may be integrated to implement a production-type transparent symmetric active/active replication software framework.

Furthermore, the same two preliminary proof-of-concept prototypes for a transparent symmetric active/active replication software framework for client-service and dependent service scenarios may be used to develop production-type high availability support for HPC system services beyond the proof-of-concept prototype stage. The developed symmetric active/active high availability proof-of-concept prototype for the HPC job and resource management service and for the metadata service of the parallel file system were custom implementations. Production-type deployment and continued support within the code base of a service requires a transparent replication infrastructure with a standard API, such as the VCL of the preliminary proof-of-concept prototype for a transparent symmetric active/active replication software framework for dependent services.

The comparison of service-level high availability methods (Section 3.3) revealed that the interfaces and mechanisms of all methods are quite similar. A replication software framework that supports all these methods, such as active/standby, asymmetric active/active, and symmetric active/active, may be of great practical value. This would not only allow a practical comparison, but it would also yield a tunable replication infrastructure that allows adaptation to individual performance and availability needs.

Lastly, the work presented in this thesis, including theory and proof-of-concept prototypes, may be applied to other service-oriented or service-dependent architectures, such as to critical Web services (providers, brokers, ...), critical infrastructure services in peer-to-peer or collaborative environments (directory servers, share points, ...), and critical enterprise services (inventory and personnel databases, payroll services, ...).

6 References

[1] Hans Meuer, Erich Strohmaier, Jack J. Dongarra, and Horst Simon. Top 500 list of supercomputer sites, 2007. URL http://www.top500.org.

[2] Bianca Schroeder and Garth A. Gibson. Understanding failures in petascale computers. In *Journal of Physics: Proceedings of the Scientific Discovery through Advanced Computing Program (SciDAC) Conference 2007*, volume 78, pages 2022–2032, Boston, MA, USA, June 24-28, 2007. Institute of Physics Publishing, Bristol, UK. URL http://www.iop.org/EJ/abstract/1742-6596/78/1/012022.

[3] John T. Daly, Lori A. Pritchett-Sheats, and Sarah E. Michalak. Application MTTFE vs. platform MTTF: A fresh perspective on system reliability and application throughput for computations at scale. In *Proceedings of the* 8^{th} *IEEE International Symposium on Cluster Computing and the Grid (CCGrid) 2008: Workshop on Resiliency in High Performance Computing (Resilience) 2008*, Lyon, France, May 19-22, 2008. IEEE Computer Society. URL http://xcr.cenit.latech.edu/resilience2008/program/resilience08-10.pdf.

[4] Salim Hariri and Manish Parashar. *Tools and Environments for Parallel and Distributed Computing.* Wiley InterScience, John Wiley & Sons, Inc., Hoboken, NJ, USA, January 2004. ISBN 978-0-471-33288-6. URL http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471332887.html.

[5] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics,* 2^{nd} *Edition.* Wiley InterScience, John Wiley & Sons, Inc., Hoboken, NJ, USA, March 2004. ISBN 978-0-471-45324-6. URL http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471453242.html.

[6] David E. Culler, Jaswinder P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Burlington, MA, USA, August 1998. ISBN 978-1-558-60343-1. URL http://books.elsevier.com/us/mk/us/subindex.asp?isbn=9781558603431.

[7] Michael T. Heath. *Scientific Computing, 2^{nd} Edition*. McGraw-Hill, New York, NY, USA, August 2001. ISBN 978-0-072-39910-3. URL http://catalogs.mhhe.com/mhhe/viewProductDetails.do?isbn=0072399104.

[8] National Center for Atmospheric Research, Boulder, CO, USA. Community Climate System Model (CCSM) documentation, 2007. URL http://www.ccsm.ucar.edu.

[9] The Nobel Foundation, Stockholm, Sweden. The Nobel Peace Prize 2007, October 11, 2007. URL http://nobelprize.org/nobel_prizes/peace/laureates/2007.

[10] Intergovernmental Panel on Climate Change (IPCC). Press release: IPCC expresses surprise and gratitude at announcement of Nobel Peace Prize, October 12, 2007. URL http://www.ipcc.ch/press/prpnp12oct07.htm.

[11] National Center for Atmospheric Research, Boulder, CO, USA. Press release: NCAR scientists and technical staff share in Nobel Peace Prize with IPCC colleagues around the world, October 11, 2007. URL http://www.ncar.ucar.edu/news/press/press_release.php?id=2840.

[12] Oak Ridge National Laboratory, Oak Ridge, TN, USA. Terascale Supernova Initiative (TSI) documentation, 2007. URL http://www.phy.ornl.gov/tsi.

[13] National Center for Computational Sciences, Oak Ridge, TN, USA. Jaguar Cray XT system documentation, 2007. URL http://www.nccs.gov/computing-resources/jaguar.

[14] National Center for Computational Sciences, Oak Ridge, TN, USA. Leadership science, 2007. URL http://www.nccs.gov/leadership-science.

[15] Mark Seager. Operational machines: ASCI White. Talk at the 7^{th} Workshop on Distributed Supercomputing (SOS) 2003, March 4-6, 2003. URL http://www.cs.sandia.gov/SOS7/presentations/seager_white.ppt.

[16] Chung-Hsing Hsu and Wu-Chun Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing and Networking (SC) 2005*, Seattle, WA, USA, November 12-18, 2005. IEEE Computer Society. ISBN 1-59593-061-2. URL http://dx.doi.org/10.1109/SC.2005.3.

[17] National Energy Research Scientific Computing Center (NERSC), Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA, USA. Current and past HPC system availability statistics, 2007. URL http://www.nersc.gov/nusers/status/AvailStats.

[18] John Morrison. The ASCI Q system at Los Alamos. Talk at the 7^{th} Workshop on Distributed Supercomputing (SOS) 2003, March 4-6, 2003. URL http://www.cs.sandia.gov/SOS7/presentations/seager_white.ppt.

[19] Frederick H. Streitz. Simulating solidification in metals at high pressure – The drive to petascale computing. Keynote address at the 12^{th} Annual San Diego Supercomputer Center (SDSC) Summer Institute 2006, July 17-21, 2006. URL http://www.sdsc.edu/us/training/workshops/2006summerinstitute/docs/SI2006_Streitz_keynote.ppt.

[20] Ian R. Philp. Software failures and the road to a petaflop machine. In *Proceedings of the* 1^{st} *Workshop on High Performance Computing Reliability Issues (HPCRI) 2005, in conjunction with the* 11^{th} *International Symposium on High Performance Computer Architecture (HPCA) 2005*, San Francisco, CA, USA, February 12-16, 2005. IEEE Computer Society.

[21] Heather Quinn and Paul Graham. Terrestrial-based radiation upsets: A cautionary tale. In *Proceedings of the* 13^{th} *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) 2005*, pages 193–202, Napa, CA, USA, April 18-20, 2005. IEEE Computer Society. ISBN 0-7695-2445-1. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1508539.

[22] Cray Inc., Seattle, WA, USA. Cray XD1 computing platform documentation, 2007. URL http://www.cray.com/products/legacy.html.

[23] Hans Meuer, Erich Strohmaier, Jack J. Dongarra, and Horst Simon. Top 500 list of supercomputer sites: The advanced simulation and computing initiative (ASCI) Q system, 2007. URL http://www.top500.org/system/6359.

[24] Alan L. Robertson. The evolution of the Linux-HA project. In *Proceedings of the UKUUG LISA/Winter Conference – High-Availability and Reliability – 2004*, Bournemouth, UK, June 21, 2004. URL http://www.linux-ha.org/_cache/TechnicalPapers__HBevolution.pdf.

[25] Alan L. Robertson. Linux-HA heartbeat design. In *Proceedings of the* 4^{th} *Annual Linux Showcase and Conference 2000*, Atlanta, Georgia, October 10-14, 2000. URL http://www.linuxshowcase.org/2000/2000papers/papers/robertson/robertson.pdf.

[26] Linux-HA (High-Availability Linux) project. Heartbeat program documentation, 2007. URL http://www.linux-ha.org/HeartbeatProgram.

[27] Pedro Pla. Drbd in a heartbeat. *Linux Journal (LJ)*, September 2006. URL http://www.linuxjournal.com/article/9074.

[28] Philipp Reisner and Lars Ellenberg. Drbd v8 – Replicated storage with shared disk semantics. In *Proceedings of the* 12^{th} *International Linux System Technology Conference (Linux-Kongress) 2005*, Hamburg, Germany, October 11-14, 2005. URL http://www.drbd.org/fileadmin/drbd/publications/drbd8.pdf.

[29] Philipp Reisner and Lars Ellenberg. Distributed Replicated Block Device (DRDB) documentation, 2007. URL http://www.drbd.org.

[30] Susanne M. Balle and Dan Palermo. Enhancing an open source resource manager with multi-core/multi-threaded support. In *Lecture Notes in Computer Science: Proceedings of the* 13^{th} *Workshop on Job Scheduling Strategies for Parallel Processing (JSSP) 2007*, volume 4942, pages 37–50, Seattle, WA, USA, June 17, 2007. Springer Verlag, Berlin, Germany. ISBN 978-3-540-78698-6, ISSN 0302-9743. URL http://www.cs.huji.ac.il/~feit/parsched/jsspp07/p2-balle.pdf.

[31] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux utility for resource management. In *Lecture Notes in Computer Science: Proceedings of the* 9^{th} *International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, volume 2862, pages 44–60, Seattle, WA, USA, June 24, 2003. Springer Verlag, Berlin, Germany. ISBN 978-3-540-20405-3, ISSN 0302-9743. URL http://www.springerlink.com/content/c4pgx63utdajtuwn.

[32] Lawrence Livermore National Laboratory, Livermore, CA, USA. Simple Linux Utility for Resource Management (SLURM) documentation, 2007. URL http://www.llnl.gov/linux/slurm.

[33] José Moreira, Michael Brutman, nos José Casta Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a highly-scalable operating system: The Blue Gene/L story. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2006*, page 118, Tampa, FL, USA, November 11-17, 2006. ACM Press, New York, NY, USA. ISBN 0-7695-2700-0. URL http://doi.acm.org/10.1145/1188455.1188578.

[34] IBM Corporation, Armonk, NY, USA. IBM Blue Gene computing platform documentation, 2007. URL http://www-03.ibm.com/servers/deepcomputing/bluegene.html.

[35] Sun Microsystems, Inc, Santa Clara, CA, USA. Sun Grid Engine (SGE) documentation, 2007. URL http://www.sun.com/software/gridware.

[36] Sun Microsystems, Inc, Santa Clara, CA, USA. Open Source Grid Engine documentation, 2007. URL http://gridengine.sunsource.net.

[37] James Coomer. Introduction to the cluster grid – Part 1. *Sun Blueprints*, August 2002. Sun Microsystems, Inc., Palo Alto, CA, USA. URL http://www.sun.com/blueprints/0802/816-7444-10.pdf.

[38] Philip H. Carns. PVFS2 high-availability clustering using Heartbeat 2.0, 2007. URL http://www.pvfs.org/doc/pvfs2-ha-heartbeat-v2.pdf.

[39] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the* 4^{th} *Annual Linux Showcase and Conference 2000*, pages 317–327, Atlanta, Georgia, October 10-14, 2000. URL http://www.linuxshowcase.org/2000/2000papers/papers/robertson/robertson.pdf.

[40] PVFS Development Team. Parallel Virtual File System (PVFS) documentation, 2007. URL http://www.pvfs.org/.

[41] Weikuan Yu, Ranjit Noronha, Shuang Liang, and Dhabaleswar K. Panda. Benefits of high speed interconnects to cluster file systems: A case study with Lustre. In *Proceedings of the* 20^{st} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2006*, pages 8–15, Rhodes Island, Greece, April 25-29, 2006. IEEE Computer Society. ISBN 1-4244-0054-6. URL http://nowlab.cse.ohio-state.edu/publications/conf-papers/2006/yu-cac06.pdf.

[42] Sun Microsystems, Inc., Santa Clara, CA, USA. Lustre file system – High-performance storage architecture and scalable cluster file system, 2007. URL http://www.sun.com/software/products/lustre/docs/lustrefilesystem_wp.pdf. White paper.

[43] Cluster File Systems, Inc., Boulder, CO, USA. Lustre Cluster File System documentation, 2007. URL http://www.lustre.org.

[44] Ibrahim Haddad, Chokchai Leangsuksun, Stephen L. Scott, and Tong Liu. HA-OSCAR: Towards highly available linux clusters. *Linux World Magazine*, March 2004. URL http://linux.sys-con.com/read/43713.htm.

[45] Kshitij Limaye, Chokchai Leangsuksun, Zeno Greenwood, Stephen L. Scott, Christian Engelmann, Richard M. Libby, and Kasidit Chanchio. Job-site level fault tolerance for cluster and grid environments. In *Proceedings of the* 7^{th} *IEEE International Conference on Cluster Computing (Cluster) 2005*, pages 1–9, Boston, MA, USA, September 26-30, 2005. IEEE Computer Society. ISBN 0-7803-9486-0, ISSN 1552-5244. URL http://www.csm.ornl.gov/~engelman/publications/limaye05job-site.pdf.

[46] Louisiana Tech University, Ruston, LA, USA. High Availability Open Source Cluster Application Resources (HA-OSCAR) documentation, 2007. URL http://xcr.cenit.latech.edu/ha-oscar.

[47] Altair Engineering, Troy, MI, USA. OpenPBS documentation, 2007. URL http://www.openpbs.org.

[48] Altair Engineering, Troy, MI, USA. PBS Pro documentation, 2007. URL http://www.pbsgridworks.com.

[49] Altair Engineering, Troy, MI, USA. PBS Pro for Cray computing platforms, 2007. URL http://www.pbsgridworks.com/PBS/pdfs/PBSPro_Cray.pdf.

[50] Cluster Resources, Inc, Salt Lake City, UT, USA. Moab Workload Manager administrator‘s guide, 2007. URL http://www.clusterresources.com/products/mwm/docs.

[51] Cluster Resources, Inc, Salt Lake City, UT, USA. Moab Workload Manager documentation, 2007. URL http://www.clusterresources.com/products/mwm.

[52] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@home – Massively distributed computing for SETI. *Computing in Science & Engineering*, 3(1):78–83, 2001. IEEE Computer Society. ISSN 1521-9615. URL http://setiathome.berkeley.edu/sah_papers/CISE.pdf.

[53] David P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the* 5^{th} *International Workshop on Grid Computing (Grid) 2004*, pages 4–10, Pittsburgh, PA, USA, November 8, 2004. IEEE Computer Society. ISBN 0-7695-2256-4, ISSN 1550-5510. URL http://boinc.berkeley.edu/grid_paper_04.pdf.

[54] Space Sciences Laboratory, University of California, Berkeley, CA, USA. SETI@HOME documentation, 2007. URL http://setiathome.ssl.berkeley.edu.

[55] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and*

Experience, 17(2-4):323–356, 2005. Wiley InterScience, John Wiley & Sons, Inc., Hoboken, NJ, USA. ISSN 1532-0626. URL http://www.cs.wisc.edu/condor/doc/condor-practice.pdf.

[56] Jim Basney and Miron Livny. Deploying a high throughput computing cluster. In Rajkumar Buyya, editor, *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall PTR, Upper Saddle River, NJ, USA, May 1999. ISBN 978-0-130-13784-5. URL http://www.informit.com/store/product.aspx?isbn=0130137847.

[57] Computer Sciences Department, University of Wisconsin, Madison, WI, USA. Condor documentation, 2007. URL http://www.cs.wisc.edu/condor.

[58] Chokchai Leangsuksun, Venkata K. Munganuru, Tong Liu, Stephen L. Scott, and Christian Engelmann. Asymmetric active-active high availability for high-end computing. In *Proceedings of the* 2^{nd} *International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2) 2005, in conjunction with the* 19^{th} *ACM International Conference on Supercomputing (ICS) 2005*, Cambridge, MA, USA, June 19, 2005. URL http://www.csm.ornl.gov/~engelman/publications/leangsuksun05asymmetric.pdf.

[59] Linux-HA (High-Availability Linux) project. Node fencing explained, 2007. URL http://www.linux-ha.org/NodeFencing.

[60] Paul H. Hargrove and Jason C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters. In *Journal of Physics: Proceedings of the Scientific Discovery through Advanced Computing Program (SciDAC) Conference 2006*, volume 46, pages 494–499, Denver, CO, USA, June 25-29, 2006. Institute of Physics Publishing, Bristol, UK. URL http://www.iop.org/EJ/article/1742-6596/46/1/067/jpconf6_46_067.pdf.

[61] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings of the Los Alamos Computer Science Institute (LACSI) Symposium 2003*, Santa Fe, NM, USA, October 27-29, 2003. URL http://ftg.lbl.gov/CheckpointRestart/lacsi-2003.pdf.

[62] Lawrence Berkeley National Laboratory, Berkeley, CA, USA. Berkeley Lab Checkpoint/Restart (BLCR) documentation, 2007. URL http://ftg.lbl.gov/checkpoint.

[63] Jeffrey M. Squyres and Andrew Lumsdaine. A component architecture for LAM/MPI. In *Lecture Notes in Computer Science: Proceedings of the* 10^{th} *European PVM/MPI Users' Group Meeting (EuroPVM/MPI) 2003*, volume 2840, pages 379–387, Venice, Italy, September 29 - October 2, 2003. Springer Verlag, Berlin, Germany. ISBN 978-3-540-20149-6, ISSN 0302-9743. URL http://www.lam-mpi.org/papers/euro-pvmmpi2003/euro-pvmmpi-2003.pdf.

[64] Indiana University, Bloomington, IN, USA. Local Area Multicomputer Message Passing Interface (LAM-MPI) documentation, 2007. URL http://www.lam-mpi.org.

[65] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack J. Dongarra. *MPI: The Complete Reference (Vol. 1),* 2^{nd} *Edition*. MIT Press, Cambridge, MA, USA, September 1998. ISBN 978-0-262-69215-1. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=3898.

[66] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing and Networking (SC) 2005*, page 9, Seattle, WA, USA, November 12-18, 2005. IEEE Computer Society. ISBN 1-59593-061-2. URL http://hpc.pnl.gov/people/fabrizio/papers/sc05.pdf.

[67] Pacific Northwest National Laboratory, Richland, WA, USA. TICK: Transparent Incremental Checkpointing at Kernel-level documentation, 2007. URL http://hpc.pnl.gov/sft/tick.html.

[68] Joseph F. Ruscio, Michael A. Heffner, and Srinidhi Varadarajan. DejaVu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *Proceedings of the* 21^{st} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2007*, Long Beach, CA, USA, March 26-30, 2007. ACM Press, New York, NY, USA. ISBN 978-1-59593-768-1. URL http://doi.ieeecomputersociety.org/10.1109/IPDPS.2007.370309.

[69] Srinidhi Varadarajan. An evaluation of the EverGrid Deja Vu checkpoint/restart software. Technical Report LBNL/PUB-960, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, Baltimore, MD, USA, 2006. URL http://www.lbl.gov/cs/html/reports.html.

[70] Evergrid, Fremont, CA, USA. Evergrid Availability Services documentation, 2007. URL http://www.evergrid.com/products/availability-services.html.

[71] Youngbae Kim, James S. Plank, and Jack J. Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing (JPDC)*, 43(2):125–138, 1997. Elsevier, Amsterdam, The Netherlands. ISSN 0743-7315. URL http://dx.doi.org/10.1006/jpdc.1997.1336.

[72] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 9(10):972–986, 1998. IEEE Computer Society. ISSN 1045-9219. URL http://doi.ieeecomputersociety.org/10.1109/71.730527.

[73] Luis M. Silva and Joao Gabriel Silva. An experimental study about diskless checkpointing. In *Proceedings of the* 24^{th} *Euromicro Conference on Engineering Systems and Software for the Next Decade 1998*, page 10395, Vesteras, Sweden, August 25-27, 1998. IEEE Computer Society. ISBN 8186-8646-4-1, ISSN 1089-6503. URL http://doi.ieeecomputersociety.org/10.1109/EURMIC.1998.711832.

[74] Christian Engelmann and George A. Geist. A diskless checkpointing algorithm for super-scale architectures applied to the fast fourier transform. In *Proceedings of the Challenges of Large Applications in Distributed Environments Workshop (CLADE) 2003, in conjunction with the* 12^{th} *IEEE International Symposium on High Performance Distributed Computing (HPDC) 2003*, page 47, Seattle, WA, USA, June 21, 2003. IEEE Computer Society. ISBN 0-7695-1984-9. URL http://www.csm.ornl.gov/~engelman/publications/engelmann03diskless.pdf.

[75] Christian Engelmann and George A. Geist. Super-scalable algorithms for computing on 100,000 processors. In *Lecture Notes in Computer Science: Proceedings of the* 5^{th} *International Conference on Computational Science (ICCS) 2005, Part I*, volume 3514, pages 313–320, Atlanta, GA, USA, May 22-25, 2005. Springer Verlag, Berlin, Germany. ISBN 978-3-540-26032-5, ISSN 0302-9743. URL http://www.csm.ornl.gov/~engelman/publications/engelmann05superscalable.pdf.

[76] Aurélien Bouteiller, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Franck Cappello. MPICH-V: A multiprotocol fault tolerant MPI. *International Journal of High Performance Computing and Applications (IJHPCA)*, 20(3):319–333, 2006. SAGE Publications, Thousand Oaks, CA, USA. ISSN 1094-3420. URL http://mpich-v.lri.fr/papers/ijhpca_mpichv.pdf.

[77] Darius Buntinas, Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *Proceedings of*

the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2006, page 18, Tampa, FL, USA, November 11-17, 2006. ACM Press, New York, NY, USA. ISBN 0-7695-2700-0. URL http://mpich-v.lri.fr/papers/mpichv_sc2006.pdf.

[78] University of Paris-South, France. MPICH-V message logging layer documentation, 2007. URL http://mpich-v.lri.fr.

[79] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985. ACM Press, New York, NY, USA. ISSN 0734-2071. URL http://doi.acm.org/10.1145/214451.214456.

[80] Graham E. Fagg, Antonin Bukovsky, and Jack J. Dongarra. Fault-tolerant MPI for the Harness metacomputing system. In *Lecture Notes in Computer Science: Proceedings of the* 1^{st} *International Conference on Computational Science (ICCS) 2002, Part I*, volume 2073, pages 355–366, San Francisco, CA, USA, May 28-30, 2001. Springer Verlag, Berlin, Germany. URL http://www.netlib.org/utk/people/JackDongarra/PAPERS/ft-harness-iccs2001.ps.

[81] Graham E. Fagg, Antonin Bukovsky, and Jack J. Dongarra. Harness and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1495, 2001. Elsevier, Amsterdam, The Netherlands. ISSN 0167-8191. URL http://dx.doi.org/10.1016/S0167-8191(01)00100-4.

[82] Innovative Computing Laboratory (ICL), Computer Science Department, University of Tennessee, Knoxville, TN, USA. FT-MPI documentation, 2007. URL http://icl.cs.utk.edu/ftmpi.

[83] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Lecture Notes in Computer Science: Proceedings of the* 11^{th} *European PVM/MPI Users' Group Meeting (EuroPVM/MPI) 2004*, volume 3241, pages 97–104, Budapest, Hungary, September 19-22, 2004. Springer Verlag, Berlin, Germany. ISBN 3-540-23163-3. URL http://www.open-mpi.org/papers/euro-pvmmpi-2004-overview/euro-pvmmpi-2004-overview.pdf.

[84] Open MPI Team. Open MPI documentation, 2007. URL http://www.open-mpi.org.

[85] Zizhong Chen and Jack J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Proceedings of the* 20^{st} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2006*, page 10, Rhodes Island, Greece, April 25-29, 2006. IEEE Computer Society. ISBN 1-4244-0054-6. URL http://icl.cs.utk.edu/news_pub/submissions/checkpoint_free.pdf.

[86] Arun B. Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of the* 21^{st} *ACM International Conference on Supercomputing (ICS) 2007*, pages 23–32, Seattle, WA, USA, June 16-20, 2007. ACM Press, New York, NY, USA. ISBN 978-1-59593-768-1. URL http://www.csm.ornl.gov/~engelman/publications/nagarajan07proactive.pdf.

[87] Anand Tikotekar, Geoffroy Vallée, Thomas Naughton, Stephen L. Scott, and Chokchai Leangsuksun. Evaluation of fault-tolerant policies using simulation. In *Proceedings of the* 9^{th} *IEEE International Conference on Cluster Computing (Cluster) 2007*, Austin, TX, USA, September 17-20, 2007. IEEE Computer Society. URL http://www.csm.ornl.gov/~engelman/publications/tikotekar07evaluation.pdf.

[88] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984. ACM Press, New York, NY, USA. ISSN 0164-0925. URL http://doi.acm.org/10.1145/2993.2994.

[89] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990. ACM Press, New York, NY, USA. ISSN 0360-0300. URL http://doi.acm.org/10.1145/98163.98167.

[90] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998. ACM Press, New York, NY, USA. ISSN 0734-2071. URL http://doi.acm.org/10.1145/279227.279229.

[91] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer Verlag, Berlin, Germany, July 2005. ISBN 978-0-387-21509-9. URL http://www.springerlink.com/content/xk081t/?p=8cc942450c1c45dfa261abdf5295c9ec&pi=0.

[92] Gregory V. Chockler, Idid Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):

427–469, 2001. ACM Press, New York, NY, USA. ISSN 0163-5980. URL http://doi.acm.org/10.1145/503112.503113.

[93] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36 (4):372–421, 2004. ACM Press, New York, NY, USA. ISSN 0360-0300. URL http://doi.acm.org/10.1145/1041680.1041682.

[94] Roberto Baldoni, Stefano Cimmino, and Carlo Marchetti. Total order communications: A practical analysis. In *Lecture Notes in Computer Science: Proceedings of the* 5^{th} *European Dependable Computing Conference (EDCC) 2005*, volume 3463, pages 38–54, Budapest, Hungary, April 20-22, 2005. Springer Verlag, Berlin, Germany. ISBN 978-3-540-25723-3, ISSN 0302-9743. URL http://www.springerlink.com/content/rlcqfrdjuj33vnlq.

[95] M. Frans Kaashoek and Andrew S. Tanenbaum. An evaluation of the Amoeba group communication system. In *Proceedings of the* 16^{th} *IEEE International Conference on Distributed Computing Systems (ICDCS) 1996*, pages 436–447, Hong Kong, June 27-30, 1996. IEEE Computer Society. ISBN 0-8186-7398-2. URL http://doi.ieeecomputersociety.org/10.1109/ICDCS.1996.507992.

[96] M. Frans Kaashoek, Andrew S. Tanenbaum, and Kees Verstoep. Group communication in Amoeba and its applications. *Distributed Systems Engineering*, 1(1): 48–58, 1993. Institute of Physics Publishing, Bristol, UK. ISSN 0967-1846. URL http://www.iop.org/EJ/article/0967-1846/1/1/006/ds930106.pdf.

[97] Andrew S. Tanenbaum, M. Frans Kaashoek, Robbert van Renesse, and Henri E. Bal. The Amoeba distributed operating system – A status report. *Computer Communications*, 14(6):324–335, 1991. Butterworth-Heinemann, Newton, MA, USA. ISSN 0140-3664. URL http://www.bsslab.de/download/documents/amoeba_docs/comcom91.pdf.

[98] Department of Computer Science, VU University, Amsterdam, The Netherlands. Amoeba distributed operating system documentation, 2007. URL http://www.cs.vu.nl/pub/amoeba.

[99] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. ACM Press, New York, NY, USA. ISSN 0001-0782. URL http://doi.acm.org/10.1145/359545.359563.

[100] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425, Department of Computer Science, Cornell University, Ithaca, NY, USA, 1994. URL http://cs-tr.cs.cornell.edu/TR/CORNELLCS:TR94-1425.

[101] Ziv Bar-Joseph, Idit Keidar, and Nancy Lynch. Early-delivery dynamic atomic broadcast. In *Lecture Notes in Computer Science: Proceedings of the* 16^{th} *International Conference on Distributed Computing (DISC) 2002*, volume 2508, pages 1–16, Toulouse, France, October 28-30, 2002. Springer Verlag, Berlin, Germany. URL http://www.springerlink.com/content/pxby6vht7669wpxv.

[102] Danny Dolev, Shlomo Kramer, and Dalia Malki. Early delivery totally ordered multicast in asynchronous environments. In *Proceedings of the* 23^{rd} *Annual International Symposium on Fault-Tolerant Computing (FTCS) 1993*, pages 544–553, Toulouse, France, June 22-24 1993. IEEE Computer Society. ISBN 0-8186-3680-7. URL http://ieeexplore.ieee.org/iel3/4964/13650/00627357.pdf?arnumber=627357.

[103] Bettina Kemme, Fernando Pedone, Gustavo Alonso, Schiper Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(4): 1018–1032, 2003. IEEE Computer Society. ISSN 1041-4347. URL http://doi.ieeecomputersociety.org/10.1109/TKDE.2003.1209016.

[104] Pedro Vicente and Luis Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of the* 21^{th} *IEEE Symposium on Reliable Distributed Systems (SRDS) 2002*, pages 92–101, Suita, Japan, October 13-16, 2002. IEEE Computer Society. ISBN 0-7695-1659-9, ISSN 1060-9857. URL http://doi.ieeecomputersociety.org/10.1109/RELDIS.2002.1180177.

[105] Kenneth P. Birman and Robbert van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society, April 1994. ISBN 978-0-8186-5342-1. URL http://www.wiley.com/WileyCDA/WileyTitle/productCd-0818653426.html.

[106] Kenneth P. Birman and Timothy Clark. Performance of the ISIS distributed computing toolkit. Technical Report 94-1432, Department of Computer Science, Cornell University, Ithaca, NY, USA, 1994. URL http://cs-tr.cs.cornell.edu/TR/CORNELLCS:TR94-1432.

[107] Cornell University, Ithaca, NY, USA. Isis documentation, 2007. URL http://www.cs.cornell.edu/Info/Projects/ISIS.

[108] Louise E. Moser, Yair Amir, Peter M. Melliar-Smith, and Deborah A. Agarwal. Extended virtual synchrony. In *Proceedings of the* 14^{th} *IEEE International Conference on Distributed Computing Systems (ICDCS) 1994*, pages 56–65, Poznan, Poland, June 21-24, 1994. IEEE Computer Society. URL http://ieeexplore.ieee.org/iel2/980/7460/00302392.pdf?arnumber=302392.

[109] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996. ACM Press, New York, NY, USA. ISSN 0001-0782. URL http://doi.acm.org/10.1145/227210.227227.

[110] Hebrew University of Jerusalem, Israel. Transis documentation, 2007. URL http://www.cs.huji.ac.il/labs/transis.

[111] Kenneth P. Birman, Bob Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohul Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX) 2000*, volume 1, pages 149–161, Hilton Head, SC, USA, January 25-27 2000. IEEE Computer Society. ISBN 0-7695-0490-6. URL http://www.cs.cornell.edu/projects/quicksilver/public_pdfs/Horus%20and%20Ensemble.pdf.

[112] Robbert van Renesse, Takako M. Hickey, and Kenneth P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report 94-1442, Department of Computer Science, Cornell University, Ithaca, NY, USA, 1994. URL http://cs-tr.cs.cornell.edu/TR/CORNELLCS:TR94-1442.

[113] Cornell University, Ithaca, NY, USA. Horus documentation, 2007. URL http://www.cs.cornell.edu/Info/Projects/Horus.

[114] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth P. Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the* 17^{th} *ACM SIGOPS Symposium on Operating System Principles (SOSP) 1999*, pages 80–92, Kiawah Island Resort, SC, USA, December 12-15, 1999. ACM Press, New York, NY, USA. ISBN 1-5811-3140-2. URL http://www.cs.cornell.edu/Info/Projects/Spinglass/public_pdfs/Building%20Reliable%20High.pdf.

[115] Ohad Rodeh, Kenneth P. Birman, and Danny Dolev. The architecture and performance of security protocols in the Ensemble group communication system: Using

diamonds to guard the castle. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):289–319, 2001. ACM Press, New York, NY, USA. ISSN 1094-9224. URL http://doi.acm.org/10.1145/501978.501982.

[116] Cornell University, Ithaca, NY, USA. Ensemble documentation, 2007. URL http://dsl.cs.technion.ac.il/projects/Ensemble.

[117] Danny Dolev and Dalia Malki. The design of the Transis system. In *Lecture Notes in Computer Science: Proceedings of the Dagstuhl Workshop on Unifying Theory and Practice in Distributed Computing 1994*, volume 938, pages 83–98, Dagstuhl Castle, Germany, September 5-9, 1994. Springer Verlag, Berlin, Germany. ISBN 978-3-540-60042-8, ISSN 0302-9743. URL http://www.springerlink.com/content/f4864h647516803m/.

[118] Deborah A. Agarwal. *Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks*. PhD thesis, University of California, Santa Barbara, CA, USA, August 1994. URL http://www.csm.ornl.gov/~engelman/publications/engelmann01distributed.pdf.

[119] Yair Amir, Louise E. Moser, Peter M. Melliar-Smith, Deborah A. Agarwal, and Paul W. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems (TOCS)*, 13(4):311–342, 1995. ACM Press, New York, NY, USA. ISSN 0734-2071. URL http://doi.acm.org/10.1145/210223.210224.

[120] Deborah A. Agarwal, Louise E. Moser, Peter M. Melliar-Smith, and Ravi K. Budhia. A reliable ordered delivery protocol for interconnected local area networks. In *Proceedings of the* 3^{rd} *International Conference on Network Protocols (ICNP) 1995*, Tokyo, Japan, November 7-10, 1995. IEEE Computer Society. ISBN 0-8186-7216-1. URL http://doi.ieeecomputersociety.org/10.1109/ICNP.1995.524853.

[121] Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Jonathan Stanton. The Spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Johns Hopkins University, Center for Networking and Distributed Systems, Baltimore, MD, USA, 2004. URL http://www.cnds.jhu.edu/pub/papers/cnds-2004-1.pdf.

[122] Spread Concepts LLC, Savage, MD, USA. Spread documentation, 2007. URL http://www.spread.org.

[123] Silvano Maffeis. The object group design pattern. In *Proceedings of the* 2^{nd} *USENIX Conference on Object-Oriented Technologies (COOTS) 1996*, page 12, Toronto, ON,

Canada, June 17-21, 1996. USENIX Association, Berkeley, CA, USA. URL http://www.usenix.org/publications/library/proceedings/coots96/maffeis.html.

[124] Sean Landis and Silvano Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997. Wiley InterScience, John Wiley & Sons, Inc., Hoboken, NJ, USA. URL http://dx.doi.org/10.1002/(SICI)1096-9942(1997)3:1⟨31::AID-TAPO4⟩3.0.CO;2-A.

[125] George A. Geist, James A. Kohl, Stephen L. Scott, and Philip M. Papadopoulos. HARNESS: Adaptable virtual machine environment for heterogeneous clusters. *Parallel Processing Letters (PPL)*, 9(2):253–273, 1999. World Scientific Publishing Company, Singapore. URL http://dx.doi.org/10.1142/S0129626499000244.

[126] Micah Beck, Jack J. Dongarra, Graham E. Fagg, G. Al Geist, Paul Gray, James Kohl, Mauro Migliardi, Keith Moore, Terry Moore, Philip Papadopoulous, Stephen L. Scott, and Vaidy Sunderam. Harness: A next generation distributed virtual machine. *Future Generation Computing Systems (FGCS)*, 15(5-6):571–582, 1999. Elsevier, Amsterdam, The Netherlands. ISSN 0167-739X. URL http://www.csm.ornl.gov/harness/publications/beck98harness.pdf.

[127] Oak Ridge National Laboratory, Oak Ridge, TN, USA. Harness project documentation, 2007. URL http://www.csm.ornl.gov/harness.

[128] Christian Engelmann. Distributed peer-to-peer control for Harness. Master's thesis, Department of Computer Science, University of Reading, UK, July 7, 2001. URL http://www.csm.ornl.gov/~engelman/publications/engelmann01distributed2.pdf. Double diploma in conjunction with the Department of Engineering I, Technical College for Engineering and Economics (FHTW) Berlin, Berlin, Germany. Advisors: Prof. V. N. Alexandrov (University of Reading, Reading, UK); George A. Geist (Oak Ridge National Laboratory, Oak Ridge, TN, USA).

[129] Christian Engelmann, Stephen L. Scott, and George A. Geist. Distributed peer-to-peer control in Harness. In *Lecture Notes in Computer Science: Proceedings of the 2nd International Conference on Computational Science (ICCS) 2002, Part II: Workshop on Global and Collaborative Computing*, volume 2330, pages 720–727, Amsterdam, The Netherlands, April 21-24, 2002. Springer Verlag, Berlin, Germany. ISBN 3-540-43593-X, ISSN 0302-9743. URL http://www.csm.ornl.gov/~engelman/publications/engelmann02distributed.pdf.

[130] Christian Engelmann, Stephen L. Scott, and George A. Geist. High availability through distributed control. In *Proceedings of the High Availability and Performance*

Workshop (HAPCW) 2004, in conjunction with the Los Alamos Computer Science Institute (LACSI) Symposium 2004, Santa Fe, NM, USA, October 12, 2004. URL http://www.csm.ornl.gov/~engelman/publications/engelmann04high.pdf.

[131] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. ACM Press, New York, NY, USA. ISSN 0164-0925. URL http://doi.acm.org/10.1145/357172.357176.

[132] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM SIGOPS Operating Systems Review (OSR)*, 35 (5):15–28, 2001. ACM Press, New York, NY, USA. ISSN 0163-5980. URL http://doi.acm.org/10.1145/502059.502037.

[133] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002. ACM Press, New York, NY, USA. ISSN 0734-2071. URL http://doi.acm.org/10.1145/571637.571640.

[134] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of the* 21^{st} *ACM SIGOPS Symposium on Operating System Principles (SOSP) 2007*, pages 59–72, Stevenson, WA, USA, October 14-17, 2007. ACM Press, New York, NY, USA. ISBN 978-1-59593-591-5. URL http://doi.acm.org/10.1145/1294261.1294268.

[135] Michael G. Merideth, Arun Iyengar, Thomas Mikalsen, Stefan Tai, Isabelle Rouvellou, and Priya Narasimhan. Thema: Byzantine-fault-tolerant middleware for Web-service applications. In *Proceedings of the* 24^{th} *IEEE Symposium on Reliable Distributed Systems (SRDS) 2005*, pages 131–142, Orlando, FL, USA, October 26-28, 2005. IEEE Computer Society. ISBN 0-7695-2463-X. URL http://dx.doi.org/10.1109/RELDIS.2005.28.

[136] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the* 21^{st} *ACM SIGOPS Symposium on Operating System Principles (SOSP) 2007*, pages 45–58, Stevenson, WA, USA, October 14-17, 2007. ACM Press, New York, NY, USA. ISBN 978-1-59593-591-5. URL http://doi.acm.org/10.1145/1294261.1294267.

[137] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proceedings of the* 21^{st} *ACM SIGOPS Symposium on*

Operating System Principles (SOSP) 2007, pages 73–86, Stevenson, WA, USA, October 14-17, 2007. ACM Press, New York, NY, USA. ISBN 978-1-59593-591-5. URL http://doi.acm.org/10.1145/1294261.1294269.

[138] Hewlett-Packard Development Company, L.P., Palo Alto, CA, USA. Managing Serviceguard – Fifteenth edition, 2007. URL http://docs.hp.com/en/B3936-90122/B3936-90122.pdf.

[139] Hewlett-Packard Development Company, L.P., Palo Alto, CA, USA. HP Serviceguard quorum server version – A.02.00 release notes, fifth edition, 2007. URL http://docs.hp.com/en/B8467-90036/B8467-90036.pdf.

[140] Hewlett-Packard Development Company, L.P., Palo Alto, CA, USA. HP Integrity NonStop Computing, 2007. URL http://h20223.www2.hp.com/nonstopcomputing/cache/76385-0-0-0-121.aspx.

[141] High-Availability.com, Knutsford, Cheshire, UK. RSF-1 documentation, 2007. URL http://www.high-availability.com/links/2-8-rsf1.php.

[142] High-Availability.com, Knutsford, Cheshire, UK. RSF-1 success story: University of Salford, 2007. URL http://www.high-availability.com/downloads/WestLB.pdf.

[143] High-Availability.com, Knutsford, Cheshire, UK. RSF-1 success story: WestLB Bank, 2007. URL http://www.high-availability.com/downloads/WestLB.pdf.

[144] High-Availability.com, Knutsford, Cheshire, UK. RSF-1 success story: China Post, 2007. URL http://www.high-availability.com/downloads/ChinaPost.pdf.

[145] IBM Corporation, Armonk, NY, USA. Reliable business continuity with HACMP, 2007. URL http://www-03.ibm.com/systems/p/advantages/ha.

[146] IBM Corporation, Armonk, NY, USA. High availability cluster multi-processing for AIX – Concepts and facilities guide, 2007. URL http://publib.boulder.ibm.com/epubs/pdf/c2348649.pdf.

[147] Symantec Corporation, Cupertino, CA, USA. Veritas Cluster Server documentation, 2007. URL http://www.symantec.com/business/products/overview.jsp?pcid=2258&pvid=20_1.

[148] Symantec Corporation, Cupertino, CA, USA. Veritas Cluster Server users guide for Solaris 5.0, 2006. URL http://ftp.support.veritas.com/pub/support/products/ClusterServer_UNIX/283869.pdf.

[149] Sun Microsystems, Inc., Santa Clara, CA, USA. Solaris Cluster documentation, 2007. URL http://www.sun.com/cluster.

[150] Sun Microsystems, Inc., Santa Clara, CA, USA. Solaris Operating System: Two-node cluster how-to guide, 2007. URL http://www.sun.com/software/solaris/howtoguides/twonodecluster.jsp.

[151] Sun Microsystems, Inc., Santa Clara, CA, USA. Sun Java availability suite: Supported cluster configurations for disaster recovery, 2007. URL http://www.sun.com/software/javaenterprisesystem/suites/avail_suite_config_ds.pdf.

[152] Sun Microsystems, Inc, Santa Clara, CA, USA. Open High Availability Cluster (OHAC) documentation, 2007. URL http://opensolaris.org/os/community/ha-clusters/ohac.

[153] Microsoft Corporation, Redmond, WA, USA. Technical overview of Windows Server 2003 clustering services, 2007. URL http://www.microsoft.com/windowsserver2003/techinfo/overview/clustering.mspx.

[154] Microsoft Corporation, Redmond, WA, USA. Microsoft Cluster Server administrator's guide, 2007. URL http://www.microsoft.com/technet/archive/winntas/proddocs/mscsadm0.mspx.

[155] Red Hat, Inc., Raleigh, NC, USA. Red Hat Cluster Suite documentation, 2007. URL http://www.redhat.com/cluster_suite.

[156] Matthew O'Keefe and John Ha. Open source high-availability clustering. *Red Hat Magazine*, July 9, 2005. Red Hat, Inc., Raleigh, NC, USA. URL https://www.redhat.com/magazine/009jul05/features/cluster.

[157] Red Hat, Inc., Raleigh, NC, USA. Red Hat Enterprise Linux documentation, 2007. URL http://www.redhat.com/rhel.

[158] SteelEye Technology, Inc., Palo Alto, CA, USA. Lifekeeper documentation, 2007. URL http://www.steeleye.com/products.

[159] Silicon Graphics, Inc., Sunnyvale, CA, USA. Linux FailSafe documentation, 2007. URL http://oss.sgi.com/projects/failsafe.

[160] Linuxha.net. High availability application clustering for Linux, 2007. URL http://www.linuxha.net.

[161] Mission Critical Linux. Kimberlite Cluster documentation, 2007. URL http://www.missioncriticallinux.com/projects/kimberlite.

[162] Karla M. Sorenson. Installation and administration – Kimberlite Cluster version 1.1.0, December 2000. URL http://www.missioncriticallinux.com/projects/kimberlite/kimberlite.pdf.

[163] Narjess Ayari, Denis Barbaron, Laurent Lefèvre, and Pascale Vicat-Blanc Primet. T2CP-AR: A system for transparent tcp active replication. In *Proceedings of the* 21^{st} *IEEE International Conference on Advanced Information Networking and Applications (AINA) 2007*, pages 648–655, Niagara Falls, ON, Canada, May 21-23, 2007. IEEE Computer Society. ISBN 0-7695-2846-5, ISSN 1550-445X. URL http://doi.ieeecomputersociety.org/10.1109/AINA.2007.134.

[164] Narjess Ayari, Pablo Neira Ayuso, Laurent Lefèvre, and Denis Barbaron. Towards a dependable architecture for highly available internet services. In *Proceedings of the* 3^{rd} *International Conference on Availability, Reliability and Security (ARES) 2008*, pages 422–427, Barcelona, Spain, March 4-7, 2008. IEEE Computer Society. ISBN 978-0-7695-3102-1. URL http://doi.ieeecomputersociety.org/10.1109/ARES.2008.166.

[165] Stratus Technologies, Maynard, MA, USA. Benefit from Stratus continuous processing technology: Automatic 99.999% uptime for Microsoft Windows Server environments, 2007. URL http://www.stratus.com/download/?file=/pdf/whitepapers/cp.pdf.

[166] Stratus Technologies, Maynard, MA, USA. Enterprise servers for Windows, Linux, and VMware: Stratus ftServer family, 2007. URL http://www.stratus.com/products/ftserver/index.htm.

[167] The Linux Foundation, San Francisco, CA, USA. Open Application Interface Specification (OpenAIS) standards based cluster framework documentation, 2007. URL http://www.openais.org.

[168] Service Availability Forum, Portland, OR, USA. Application Interface Specification (AIS) documentation, 2007. URL http://www.saforum.org/specification/AIS_Information.

[169] Robert P. Goldberg. Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, Cambridge, MA, USA, March 26-27, 1973. ACM Press, New York, NY, USA.

[170] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the* 19^{th} *ACM Symposium on Operating Systems Principles (SOSP) 2003*, pages 164–177, Bolton Landing, NY, USA, October 19-22, 2003. ACM Press, New York, NY, USA. ISBN 1-58113-757-5. URL http://doi.acm.org/10.1145/945445.945462.

[171] XenSource, Inc., Palo Alto, CA, USA. Open Source Xen Hypervisor Technology, 2007. URL http://www.xensource.com.

[172] VMware, Inc., Palo Alto, CA, USA. VMware virtualization products, 2007. URL http://www.vmware.com.

[173] Erin M. Farr, Richard E. Harper, Lisa F. Spainhower, and Jimi Xenidis. A case for high availability in a virtualized environment (haven). In *Proceedings of the* 3^{rd} *International Conference on Availability, Reliability and Security (ARES) 2008*, pages 675–682, Barcelona, Spain, March 4-7, 2008. IEEE Computer Society. ISBN 978-0-7695-3102-1. URL http://doi.ieeecomputersociety.org/10.1109/ARES.2008.166.

[174] Ron I. Resnick. A modern taxonomy of high availability, 1996. URL http://www.verber.com/mark/cs/systems/A%20Modern%20Taxonomy%20of%20High%20Availability.htm.

[175] Enrique Vargas. High availability fundamentals. *Sun Blueprints*, November 2000. Sun Microsystems, Inc., Palo Alto, CA, USA. URL http://www.sun.com/blueprints/1100/HAFund.pdf.

[176] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, Burlington, MA, USA, July 2007. ISBN 978-0-12-088525-1. URL http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems.

[177] Thomas L. Sterling. *Beowulf cluster computing with Linux*. MIT Press, Cambridge, MA, USA, October 2001. ISBN 978-0-262-69274-8. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=8681.

[178] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, Cambridge, MA, USA, May 1999. ISBN 978-0-262-69218-2. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=3898.

[179] Robert G. Brown. *Engineering a Beowulf-Style Compute Cluster*. May 2004. URL http://www.phy.duke.edu/~rgb/Beowulf/beowulf_book/beowulf_book.

[180] Scyld Software, Penguin Computing Inc., San Francisco, CA, USA. Beowulf.org: The Beowulf cluster site, 2007. URL http://www.beowulf.org.

[181] Cray Inc., Seattle, WA, USA. Cray XT4 computing platform documentation, 2007. URL http://www.cray.com/products/xt4.

[182] Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the* 47^{th} *Cray User Group (CUG) Conference 2005*, Albuquerque, NM, USA, May 16-19, 2005. URL http://www.cs.sandia.gov/~smkelly/SAND2005-2780C-CUG2005-CatamountArchitecture.pdf.

[183] Ron Brightwell, Suzanne M. Kelly, and John van Dyke. Catamount software architecture with dual core extensions. In *Proceedings of the* 48^{th} *Cray User Group (CUG) Conference 2006*, Lugano, Ticino, Switzerland, May 8-11, 2006. URL http://www.sandia.gov/~rbbrigh/papers/catamount-cug06.pdf.

[184] John van Dyke, Courtenay Vaughan, and Suzanne M. Kelly. Extending Catamount for multi-core processors. In *Proceedings of the* 49^{th} *Cray User Group (CUG) Conference 2007*, Seattle, WA, USA, May 7-10, 2007. URL http://www.cs.sandia.gov/~smkelly/SAND2007-2744C-CUG2007-VanDyke.pdf.

[185] Novell Inc. SUSE Linux Enterprise Distribution documentation, 2007. URL http://www.novell.com/linux.

[186] IEEE POSIX Certification Authority. Portable Operating System Interface (POSIX) documentation, 2007. URL http://standards.ieee.org/regauth/posix.

[187] Terry Jones, Andrew Tauferner, and Todd Inglett. HPC system call usage trends. In *Proceedings of the* 8^{th} *LCI International Conference on High Performance Computing (LCI) 2007*, South Lake Tahoe, CA, USA, May 14-17, 2007. URL http://www.linuxclustersinstitute.org/conferences/archive/2007/PDF/jones_21421.pdf. Linux Clusters Institute.

[188] IBM Corporation, Armonk, NY, USA. MareNostrum eServer computing platform documentation, 2007. URL http://www.ibm.com/servers/eserver/linux/power/marenostrum.

[189] Cray Inc., Seattle, WA, USA. Cray X1 computing platform documentation, 2007. URL http://www.cray.com/products/x1.

[190] SGI, Mountain View, CA, USA. Altix computing platform documentation, 2007. URL http://www.sgi.com/products/servers/altix.

[191] Advanced Supercomputing Division, National Aeronautics and Space Administration (NASA), Ames, CA, USA. Columbia SGI Altix Supercluster computing platform documentation, 2007. URL http://www.nas.nasa.gov/About/Projects/Columbia/columbia.html.

[192] Christian Engelmann, Stephen L. Scott, Chokchai Leangsuksun, and Xubin He. Symmetric active/active high availability for high-performance computing system services. *Journal of Computers (JCP)*, 1(8):43–54, 2006. Academy Publisher, Oulu, Finland. ISSN 1796-203X. URL http://www.csm.ornl.gov/~engelman/publications/engelmann06symmetric.pdf.

[193] Kai Uhlemann, Christian Engelmann, and Stephen L. Scott. JOSHUA: Symmetric active/active replication for highly available HPC job and resource management. In *Proceedings of the* 8^{th} *IEEE International Conference on Cluster Computing (Cluster) 2006*, Barcelona, Spain, September 25-28, 2006. IEEE Computer Society. ISBN 1-4244-0328-6. URL http://www.csm.ornl.gov/~engelman/publications/uhlemann06joshua.pdf.

[194] Kai Uhlemann. High availability for high-end scientific computing. Master's thesis, Department of Computer Science, University of Reading, UK, March 6, 2006. URL http://www.csm.ornl.gov/~engelman/students/uhlemann06high.pdf. Double diploma in conjunction with the Department of Engineering I, Technical College for Engineering and Economics (FHTW) Berlin, Berlin, Germany. Advisors: Prof. Vassil N. Alexandrov (University of Reading, Reading, UK); George A. Geist and Christian Engelmann (Oak Ridge National Laboratory, Oak Ridge, TN, USA).

[195] Cluster Resources, Inc, Salt Lake City, UT, USA. TORQUE Resource Manager documentation, 2007. URL http://www.clusterresources.com/torque.

[196] Cluster Resources, Inc, Salt Lake City, UT, USA. TORQUE v2.0 administrator manual, 2007. URL http://www.clusterresources.com/wiki/doku.php?id=torque:torque_wiki.

[197] Christian Engelmann, Stephen L. Scott, David E. Bernholdt, Narasimha R. Gottumukkala, Chokchai Leangsuksun, Jyothish Varma, Chao Wang, Frank Mueller, Aniruddha G. Shet, and Ponnuswamy Sadayappan. MOLAR: Adaptive runtime support for high-end computing operating and runtime systems. *ACM SIGOPS*

Operating Systems Review (OSR), 40(2):63–72, 2006. ACM Press, New York, NY, USA. ISSN 0163-5980. URL http://www.csm.ornl.gov/~engelman/publications/engelmann06molar.pdf.

[198] Christian Engelmann, Stephen L. Scott, Chokchai Leangsuksun, and Xubin He. Active/active replication for highly available HPC system services. In *Proceedings of the* 1^{st} *International Conference on Availability, Reliability and Security (ARES) 2006:* 1^{st} *International Workshop on Frontiers in Availability, Reliability and Security (FARES) 2006*, pages 639–645, Vienna, Austria, April 20-22, 2006. IEEE Computer Society. ISBN 0-7695-2567-9. URL http://www.csm.ornl.gov/~engelman/publications/engelmann06active.pdf.

[199] Christian Engelmann and Stephen L. Scott. Concepts for high availability in scientific high-end computing. In *Proceedings of the High Availability and Performance Workshop (HAPCW) 2005, in conjunction with the Los Alamos Computer Science Institute (LACSI) Symposium 2005*, Santa Fe, NM, USA, October 11, 2005. URL http://www.csm.ornl.gov/~engelman/publications/engelmann05concepts.pdf.

[200] Cluster Resources, Inc, Salt Lake City, UT, USA. Maui Cluster Scheduler documentation, 2007. URL http://www.clusterresources.com/products/maui.

[201] Li Ou, Christian Engelmann, Xubin He, Xin Chen, and Stephen L. Scott. Symmetric active/active metadata service for highly available cluster storage systems. In *Proceedings of the* 19^{th} *IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS) 2007*, Cambridge, MA, USA, November 19-21, 2007. ACTA Press, Calgary, AB, Canada. ISBN 978-0-88986-703-1. URL http://www.csm.ornl.gov/~engelman/publications/ou07symmetric.pdf.

[202] Xubin He, Li Ou, Christian Engelmann, Xin Chen, and Stephen L. Scott. Symmetric active/active metadata service for high availability parallel file systems. *Journal of Parallel and Distributed Computing (JPDC)*, 2008. Elsevier, Amsterdam, The Netherlands. ISSN 0743-7315. Submitted, under review.

[203] Li Ou, Xubin He, Christian Engelmann, and Stephen L. Scott. A fast delivery protocol for total order broadcasting. In *Proceedings of the* 16^{th} *IEEE International Conference on Computer Communications and Networks (ICCCN) 2007*, Honolulu, HI, USA, August 13-16, 2007. IEEE Computer Society. ISBN 978-1-4244-1251-8, ISSN 1095-2055. URL http://www.csm.ornl.gov/~engelman/publications/ou07fast.pdf.

[204] Li Ou. *Design of a High-Performance and High-Availability Distributed Storage System*. PhD thesis, Department of Electrical and Computer Engineering, Tennessee Technological University, Cookeville, TN, USA, December 2006. URL http://www.csm.ornl.gov/~engelman/students/ou06design.pdf. Graduate advisory committee: Prof. Xubin He, Periasamy K. Rajan, Prof. Roger L. Haggard, Prof. Martha J. Kosa, Prof. Kwun Lon Ting (Tennessee Technological University, Cookeville, TN, USA); Prof. Jeffrey Norden (State University of New York, Binghamton, NY, USA), and Stephen L. Scott (Oak Ridge National Laboratory, Oak Ridge, TN, USA). Internship supervisors: Christian Engelmann and Stephen L. Scott (Oak Ridge National Laboratory, Oak Ridge, TN, USA).

[205] Manoj Pillai and Mario Lauria. CSAR: Cluster storage with adaptive redundancy. In *Proceedings of the IEEE International Conference on Parallel Processing (ICPP) 2003*, pages 223–230, Kaohsiung, Taiwan, October 6-9, 2003. IEEE Computer Society. ISBN 0-7695-2017-0, ISSN 0190-3918. URL http://ieeexplore.ieee.org/iel5/8782/27813/01240584.pdf.

[206] Sheng-Kai Hung and Yarsun Hsu. Modularized redundant parallel virtual file system. In *Lecture Notes in Computer Science: Proceedings of the* 10^{th} *Asia-Pacific Conference, (ACSAC) 2005*, volume 3740, pages 186–199, Singapore, October 24-26, 2005. Springer Verlag, Berlin, Germany. ISBN 978-3-540-29643-0, ISSN 0302-9743. URL http://www.springerlink.com/content/b285223lv1177404.

[207] Sheng-Kai Hung and Yarsun Hsu. DPCT: Distributed parity cache table for redundant parallel file system. In *Lecture Notes in Computer Science: Proceedings of the* 7^{th} *International Conference on High Performance Computing and Communications (HPCC) 2006*, volume 4208, pages 320–329, Munich, Germany, September 13-15, 2006. Springer Verlag, Berlin, Germany. ISBN 978-3-540-39368-9, ISSN 0302-9743. URL http://www.springerlink.com/content/j30g04715784303x/.

[208] Christian Engelmann, Stephen L. Scott, Chokchai Leangsuksun, and Xubin He. Transparent symmetric active/active replication for service-level high availability. In *Proceedings of the* 7^{th} *IEEE International Symposium on Cluster Computing and the Grid (CCGrid) 2007:* 7^{th} *International Workshop on Global and Peer-to-Peer Computing (GP2PC) 2007*, pages 755–760, Rio de Janeiro, Brazil, May 14-17, 2007. IEEE Computer Society. ISBN 0-7695-2833-3. URL http://www.csm.ornl.gov/~engelman/publications/engelmann07transparent.pdf.

[209] Christian Engelmann, Stephen L. Scott, Chokchai Leangsuksun, and Xubin He. Symmetric active/active replication for dependent services. In *Proceedings of*

the 3rd International Conference on Availability, Reliability and Security (ARES) 2008, pages 260–267, Barcelona, Spain, March 4-7, 2008. IEEE Computer Society. ISBN 978-0-7695-3102-1. URL http://www.csm.ornl.gov/~engelman/publications/engelmann08symmetric.pdf.

[210] Matthias Weber. High availability for the Lustre file system. Master's thesis, Department of Computer Science, University of Reading, UK, March 14, 2007. URL http://www.csm.ornl.gov/~engelman/students/weber07high.pdf. Double diploma in conjunction with the Department of Engineering I, Technical College for Engineering and Economics (FHTW) Berlin, Berlin, Germany. Advisors: Prof. Vassil N. Alexandrov (University of Reading, Reading, UK); Christian Engelmann (Oak Ridge National Laboratory, Oak Ridge, TN, USA).

A Appendix

A.1 Detailed Prototype Test Results

A.1.1 External Symmetric Active/Active Replication for the HPC Job and Resource Management Service

Software Configuration	Active Head Nodes	Latency
TORQUE	1	98ms (100%)
TORQUE+JOSHUA	1	134ms (137%)
TORQUE+JOSHUA	2	265ms (270%)
TORQUE+JOSHUA	3	304ms (310%)
TORQUE+JOSHUA	4	349ms (356%)

Table A.1: Job submission latency performance of the symmetric active/active HPC job and resource management service prototype (averages over 100 tests)

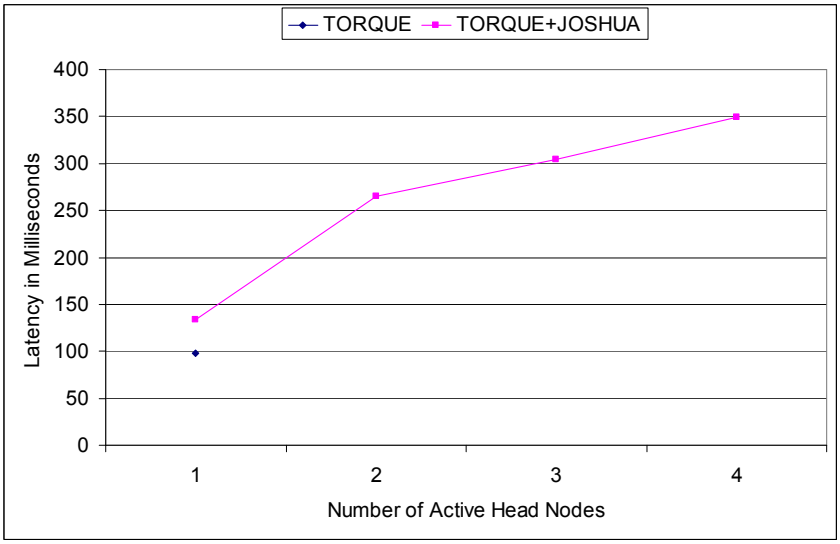


Figure A.1: Job submission latency performance of the symmetric active/active HPC job and resource management service prototype (averages over 100 tests)

Software Configuration	Active Head Nodes	10 Jobs	50 Jobs	100 Jobs
TORQUE	1	0.93s (100%)	4.95s (100%)	10.18s (100%)
TORQUE+JOSHUA	1	1.32s (70%)	6.48s (76%)	14.08s (72%)
TORQUE+JOSHUA	2	2.68s (35%)	13.09s (38%)	26.37s (39%)
TORQUE+JOSHUA	3	2.93s (32%)	15.91s (31%)	30.03s (34%)
TORQUE+JOSHUA	4	3.62s (26%)	17.65s (28%)	33.32s (31%)

Table A.2: Job submission throughput performance of the symmetric active/active HPC job and resource management service prototype (averages over 100 tests)

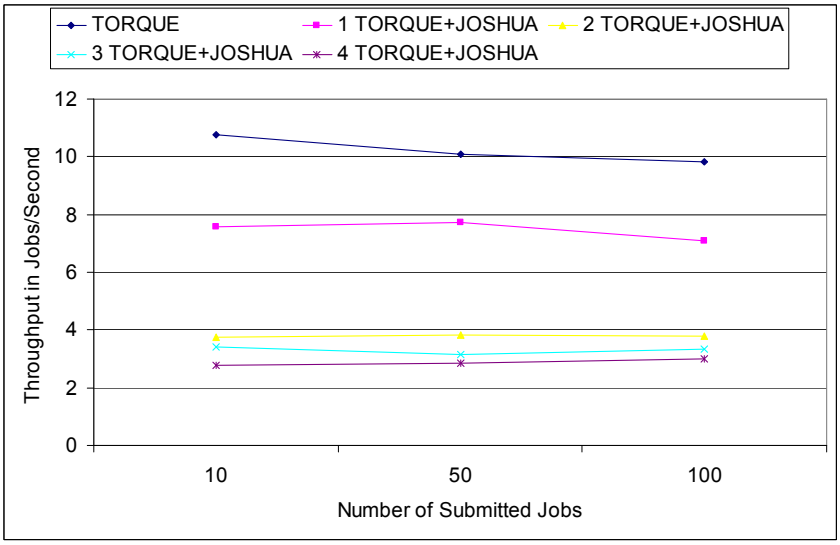


Figure A.2: Job submission throughput performance of the symmetric active/active HPC job and resource management service prototype (averages over 100 tests)

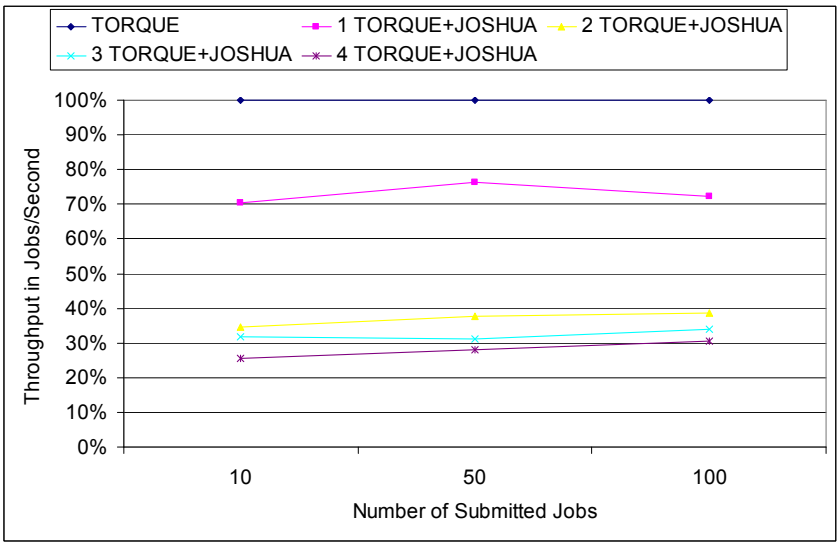


Figure A.3: Normalized job submission throughput performance of the symmetric active/active HPC job and resource management service prototype (averages over 100 tests)

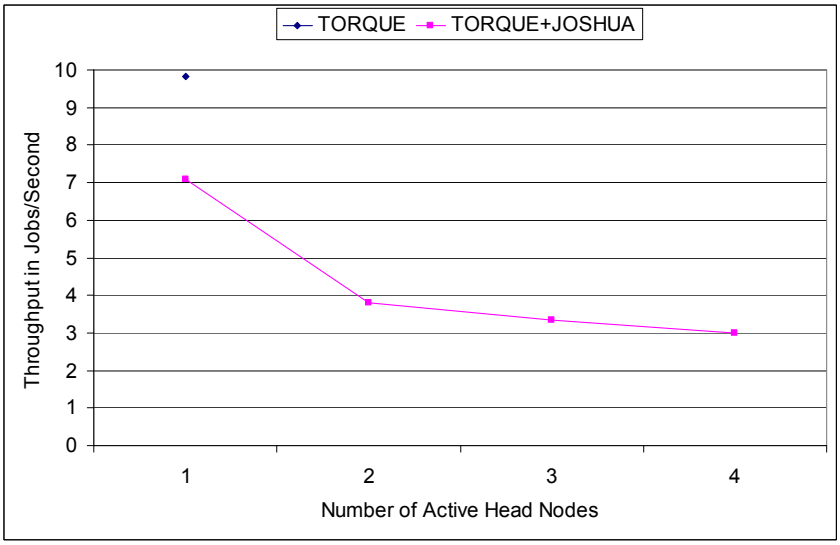


Figure A.4: Job submission throughput performance of the symmetric active/active HPC job and resource management service prototype (100 submissions, averages over 100 tests)

Active Head Nodes	MTTF 500 hours	MTTF 1,000 hours	MTTF 5,000 hours
1	93.284%	96.5251%	99.28514%
2	99.549%	99.8792%	99.99488%
3	99.970%	99.9958%	99.99996%
3	99.998%	99.9998%	99.99999%
Active Head Nodes	MTTF 10,000 hours	MTTF 50,000 hours	
1	99.641290%	99.928052%	
2	99.998711%	99.999948%	
3	99.999991%	99.999999%	
4	99.999994%	99.999999%	

Table A.3: Availability of the symmetric active/active HPC job and resource management service prototype

A.1.2 Internal Symmetric Active/Active Replication for the HPC Parallel File System Metadata Service

Processes (P)	Transis 1 Sender (Minimum)	Improved Transis 1 Sender	Transis P Senders	Improved Transis P Senders
1	230μs	235μs	227μs	
2	940μs	952μs	966μs	971μs
3	1,020μs	1,031μs	1,458μs	1,461μs
4	1,070μs	1,089μs	1,842μs	1,845μs
5	1,150μs	1,158μs	2,231μs	2,236μs
6	1,200μs	1,213μs	2,639μs	2,646μs
7	1,280μs	1,290μs	3,068μs	3,073μs
8	1,340μs	1,348μs	3,509μs	3,514μs

Maximum of Transis 1 sender = heartbeat interval, *e.g.*, ≈500,000 μs)

Table A.4: Latency performance of the fast-delivery protocol (averages over 100 tests)

PVFS MDS Clients	1	2	4
1 PVFS MDS	11ms (100%)	23ms (100%)	52ms (100%)
1 Symmetric Active/Active PVFS MDS	13ms (118%)	27ms (117%)	54ms (104%)
2 Symmetric Active/Active PVFS MDS	14ms (127%)	29ms (126%)	56ms (108%)
4 Symmetric Active/Active PVFS MDS	17ms (155%)	33ms (143%)	67ms (129%)
PVFS MDS Clients	8	16	32
1 PVFS MDS	105ms (100%)	229ms (100%)	470ms (100%)
1 Symmetric Active/Active PVFS MDS	109ms (104%)	234ms (102%)	475ms (101%)
2 Symmetric Active/Active PVFS MDS	110ms (105%)	237ms (103%)	480ms (102%)
4 Symmetric Active/Active PVFS MDS	131ms (125%)	256ms (112%)	490ms (104%)

Table A.5: Request latency performance of the symmetric active/active HPC parallel file system metadata service (averages over 100 tests)

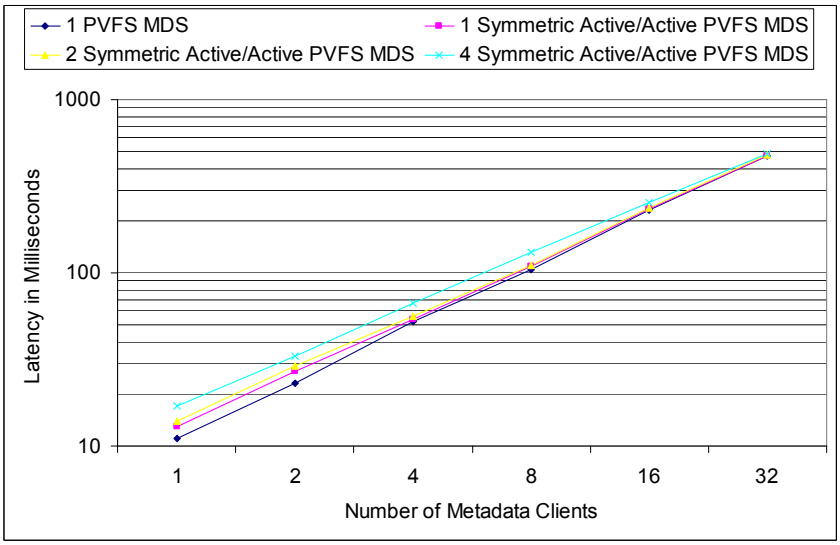


Figure A.5: Request latency performance of the symmetric active/active HPC parallel file system metadata service (averages over 100 tests)

PVFS MDS Clients	1	2	4
1 PVFS MDS	122/s (100%)	115/s (100%)	111/s (100%)
1 Symmetric Active/Active PVFS MDS	122/s (100%)	115/s (100%)	111/s (100%)
2 Symmetric Active/Active PVFS MDS	206/s (169%)	215/s (187%)	206/s (186%)
4 Symmetric Active/Active PVFS MDS	366/s (300%)	357/s (310%)	351/s (316%)
PVFS MDS Clients	8	16	32
1 PVFS MDS	107/s (100%)	103/s (100%)	88/s (100%)
1 Symmetric Active/Active PVFS MDS	107/s (100%)	103/s (100%)	88/s (100%)
2 Symmetric Active/Active PVFS MDS	205/s (192%)	194/s (188%)	155/s (176%)
4 Symmetric Active/Active PVFS MDS	347/s (324%)	340/s (330%)	334/s (380%)

Table A.6: Query throughput performance of the symmetric active/active HPC parallel file system metadata service (averages over 100 tests)

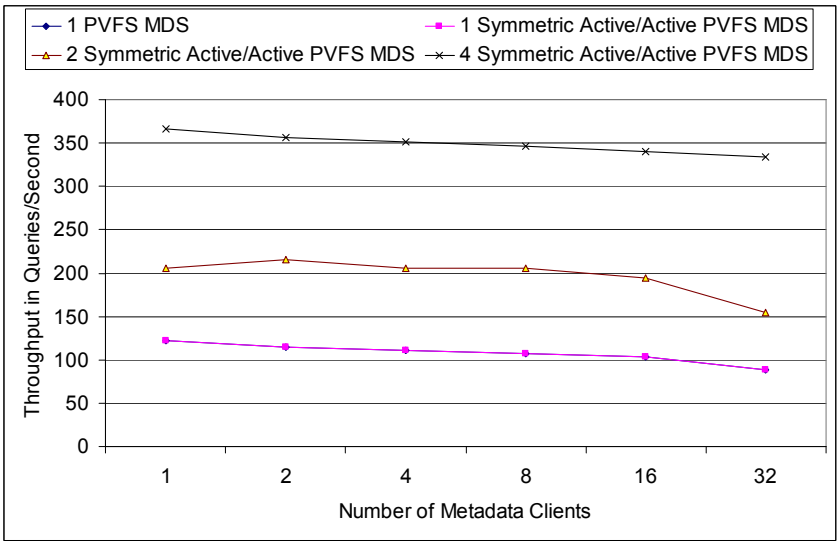


Figure A.6: Query throughput performance of the symmetric active/active HPC parallel file system metadata service (averages over 100 tests)

PVFS MDS Clients	1	2	4
1 PVFS MDS	98/s (100%)	90/s (100%)	84/s (100%)
1 Symmetric Active/Active PVFS MDS	94/s (96%)	86/s (96%)	80/s (95%)
2 Symmetric Active/Active PVFS MDS	79/s (81%)	71/s (79%)	71/s (85%)
4 Symmetric Active/Active PVFS MDS	72/s (73%)	67/s (74%)	66/s (79%)
PVFS MDS Clients	8	16	32
1 PVFS MDS	81/s (100%)	78/s (100%)	69/s (100%)
1 Symmetric Active/Active PVFS MDS	77/s (95%)	76/s (97%)	69/s (100%)
2 Symmetric Active/Active PVFS MDS	67/s (83%)	66/s (85%)	66/s (96%)
4 Symmetric Active/Active PVFS MDS	65/s (80%)	63/s (81%)	62/s (90%)

Table A.7: Request throughput performance of the symmetric active/active HPC parallel file system metadata service (averages over 100 tests)

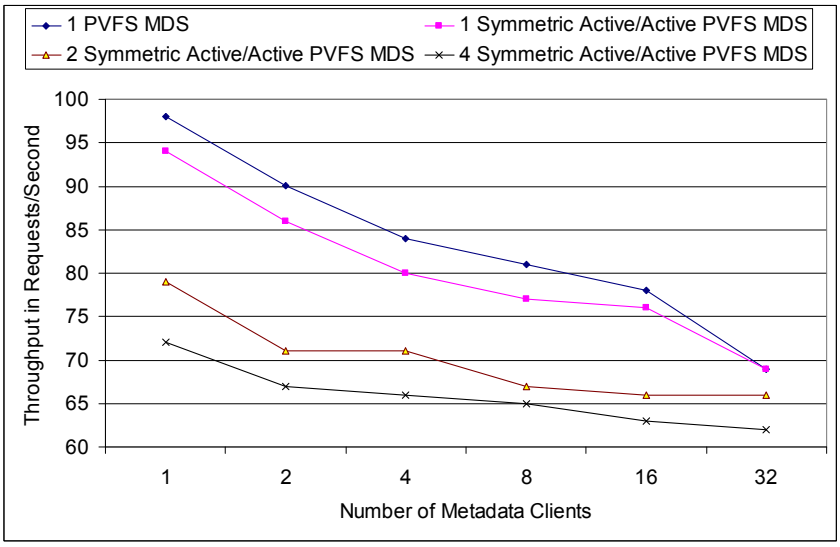


Figure A.7: Request throughput performance of the symmetric active/active HPC parallel file system metadata service (averages over 100 tests)

A.1.3 Transparent Symmetric Active/Active Replication Framework for Services

Payload	Without Interceptors	With Service Interceptor	With Both Interceptors
0.1kB	150μs (100%)	151μs (101%)	178μs (119%)
1.0kB	284μs (100%)	315μs (111%)	347μs (122%)
10.0kB	1,900μs (100%)	1,900μs (100%)	2,000μs (105%)
100.0kB	22,300μs (100%)	22,500μs (101%)	22,700μs (102%)

Table A.8: Message ping-pong (emulated emulated remote procedure call) latency performance of the transparent symmetric active/active replication framework using external replication (averages over 100 tests)

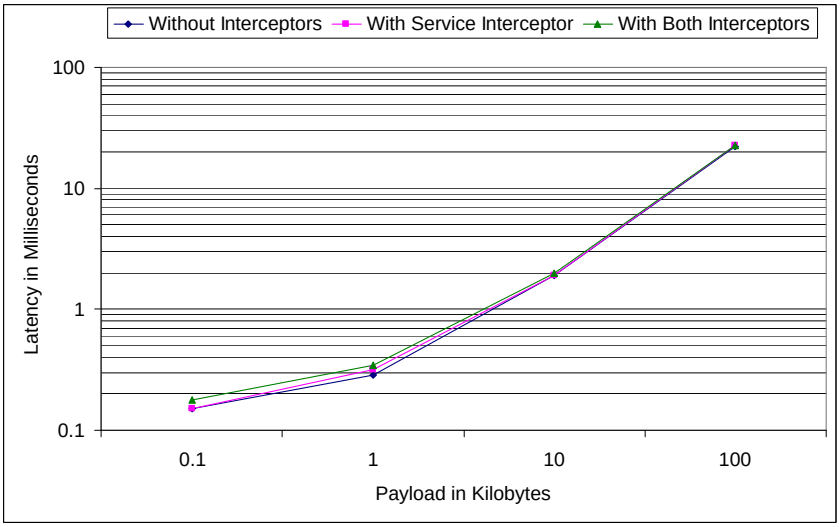


Figure A.8: Message ping-pong (emulated emulated remote procedure call) latency performance of the transparent symmetric active/active replication framework using external replication (averages over 100 tests)

Payload	Without Interceptors	With Service Interceptor	With Both Interceptors
0.1kB	0.7MBps (100%)	0.7MBps (100%)	0.6MBps (86%)
1.0kB	3.5MBps (100%)	3.2MBps (91%)	2.9MBps (83%)
10.0kB	5.3MBps (100%)	5.2MBps (98%)	5.0MBps (94%)
100.0kB	4.5MBps (100%)	4.4MBps (98%)	4.4MBps (98%)

Table A.9: Message ping-pong (emulated emulated remote procedure call) bandwidth performance of the transparent symmetric active/active replication framework using external replication (averages over 100 tests)

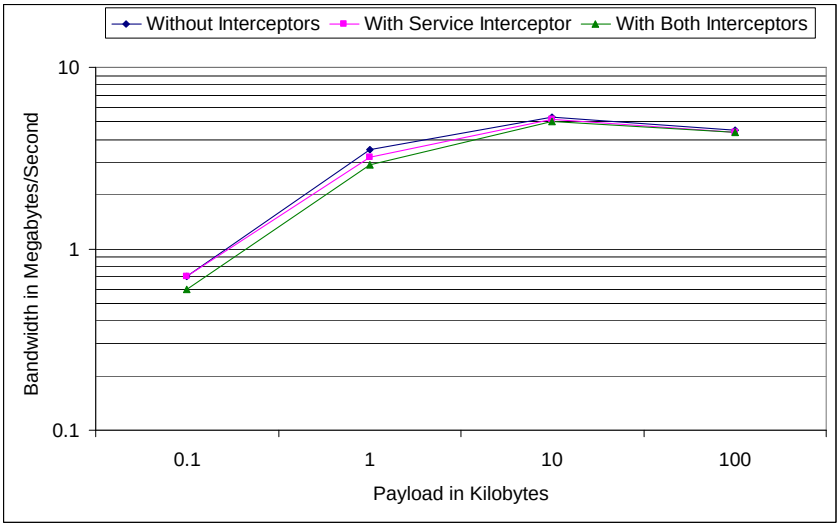


Figure A.9: Message ping-pong (emulated emulated remote procedure call) bandwidth performance of the transparent symmetric active/active replication framework using external replication (averages over 100 tests)

A.1.4 Transparent Symmetric Active/Active Replication Framework for Dependent Services

Payload	1 Service 0 Interceptors	1 Service/2 Interceptors	
0.1kB	0.10ms (100%)	0.19ms (190%)	
1.0kB	0.16ms (100%)	0.24ms (150%)	
10.0kB	0.35ms (100%)	0.45ms (129%)	
100.0kB	2.21ms (100%)	2.40ms (109%)	
1000.0kB	17.20ms (100%)	24.00ms (140%)	
Payload	2 Services/0 Interceptors	2 Services/2 Interceptors	2 Services/4 Interceptors
0.1kB	0.20ms (100%)	0.27ms (135%)	0.37ms (185%)
1.0kB	0.32ms (100%)	0.38ms (119%)	0.47ms (147%)
10.0kB	0.70ms (100%)	0.78ms (111%)	0.88ms (126%)
100.0kB	3.86ms (100%)	4.25ms (110%)	4.72ms (122%)
1000.0kB	34.40ms (100%)	40.80ms (119%)	47.70ms (139%)

Table A.10: Message ping-pong (emulated emulated remote procedure call) latency performance of the transparent symmetric active/active replication framework in a serial virtual communication layer configuration (averages over 100 tests)

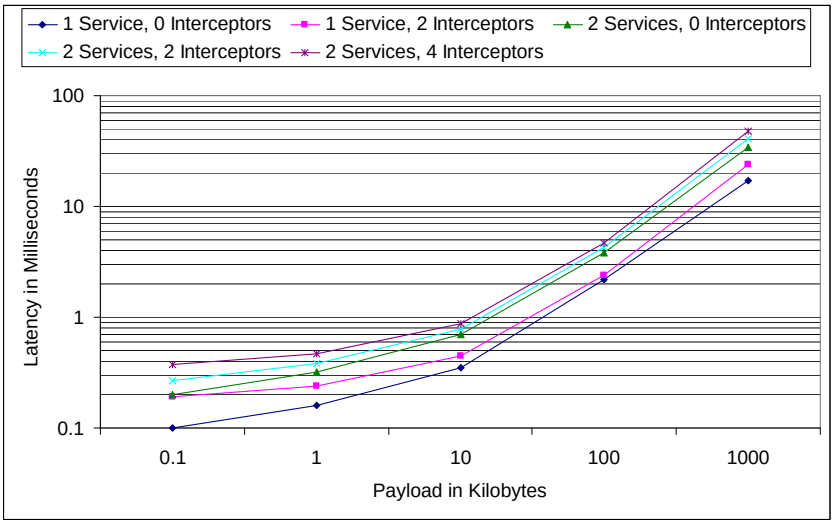


Figure A.10: Message ping-pong (emulated emulated remote procedure call) latency performance of the transparent symmetric active/active replication framework in a serial virtual communication layer configuration (averages over 100 tests)

Payload	1 Service 0 Interceptors	1 Service/2 Interceptors	
0.1kB	0.99MB/s (100%)	0.52MB/s (53%)	
1.0kB	6.28MB/s (100%)	4.15MB/s (66%)	
10.0kB	28.30MB/s (100%)	22.00MB/s (78%)	
100.0kB	45.20MB/s (100%)	41.70MB/s (92%)	
1000.0kB	58.00MB/s (100%)	41.70MB/s (72%)	
Payload	2 Services/0 Interceptors	2 Services/2 Interceptors	2 Services/4 Interceptors
0.1kB	0.49MB/s (100%)	0.37MB/s (76%)	0.27MB/s (55%)
1.0kB	3.17MB/s (100%)	2.62MB/s (83%)	2.15MB/s (68%)
10.0kB	14.20MB/s (100%)	12.80MB/s (90%)	11.40MB/s (80%)
100.0kB	25.90MB/s (100%)	23.60MB/s (91%)	21.20MB/s (82%)
1000.0kB	29.00MB/s (100%)	24.50MB/s (84%)	21.00MB/s (72%)

Table A.11: Message ping-pong (emulated remote procedure call) bandwidth performance of the transparent symmetric active/active replication framework in a serial virtual communication layer configuration (averages over 100 tests)

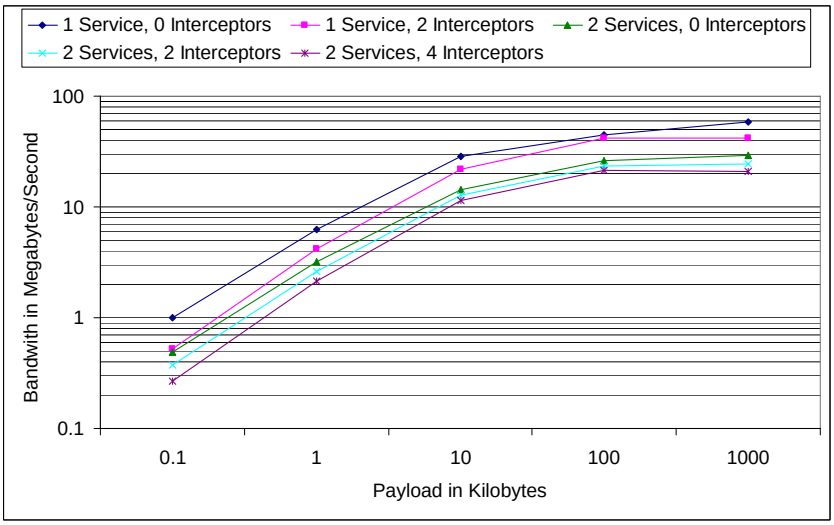


Figure A.11: Message ping-pong (emulated emulated remote procedure call) bandwidth performance of the transparent symmetric active/active replication framework in a serial virtual communication layer configuration (averages over 100 tests)