

Adopting QoS Real-Time DCPS Models
and Other Reliability Measures
on High Performance and Grid Computing

BY

Raed Abdullah Al-Shaikh

A Dissertation Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

Computer Science and Engineering

February, 2012

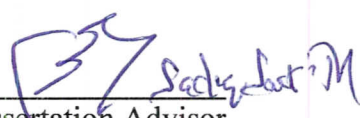
KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

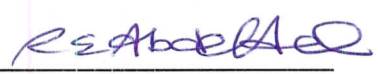
DHAHRAN, 31261, SAUDI ARABIA


DEANSHIP OF GRADUATE STUDIES

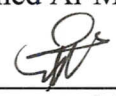
This dissertation, written by RAED A. AL-SHAIKH, under the direction of his dissertation advisor and approved by his dissertation committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING.


Dissertation Committee



Dissertation Advisor
Dr. Sadiq Sait

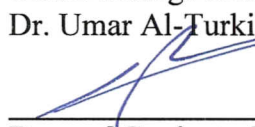

Co-Advisor / Member
Dr. Radwan Abdel Aal


Member
Dr. Mohammed Al-Mulhem


Member
Dr. Tarek Sheltami


Member
Dr. Aiman Al-Maleh


CCSE College Dean
Dr. Umar Al-Turki


Dean of Graduate Studies
Dr. Salam Zummo

20/2/12
Date

DEDICATION

This dissertation is dedicated to my parents.

ACKNOWLEDGMENTS

In the name of Allah, Most Gracious, Most Merciful

All praise and glory to Almighty Allah (SWT) who gave me courage and patience to carry out this work. Peace and blessing of Allah be upon last Prophet Muhammad (Peace Be upon Him).

Performing this research turned out to be a larger challenge than I ever imagined. I could not have completed it without the care and support of many wonderful people and organizations, and I'm delighted to be able to acknowledge them here.

My unrestrained appreciation goes to my advisor, Dr. Sadiq Sait. I couldn't have had a better mentor. He always made time to see me, no matter how busy his schedule was. He was a constant source of good ideas and a perfect detector of bad ones. More than anything, he has been a true friend.

I also wish to thank my thesis committee members, Dr. Mohammed Al-Mulhem, Dr. Tarek Sheltami, Dr. Aiman Al-Maleh and Dr. Radwan Abdel-Aal for their support and contribution. Thanks are also due to Dr. Basem Madani, the Computer Engineering Department Chairman, for his contribution and guidance. Also, my sincere appreciation goes to my friend Anas Al-Mousa for his assistance in scripting the initial DDS code.

I also thank Saudi Aramco for granting the permission to conduct this research on the High Performance Computing facilities available at the EXPEC Computer Center (ECC). Special thanks goes to RTI Co. for providing trial licenses to conduct this research.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGMENTS	iv
LIST OF TABLES	viii
LIST OF FIGURES	ix
DISSERTATION ABSTRACT.....	xi
CHAPTER1: INTRODUCTION.....	1
1.1 Motivation	4
1.2 Objectives.....	5
1.3 Contributions	6
CHAPTER 2: Recent QoS and Failure-Recovery Studies in Distributed Systems.....	8
2.1 QoSin Tightly-Coupled Distributed Systems	8
2.1.1 PromisQoS.....	8
2.1.2 QoS Aware MPI for Infiniband.....	12
2.1.3 Other Related Work on Fault Tolerant MPI.....	13
2.2 QoS in Loosely-Coupled Distributed Systems	18
2.2.1 Applying DDS for Large Scale Distributed Applications	18
2.2.2 Information Management for High Performance Autonomous Systems	21
2.2.3 Grid Technology and Information Management for Command and Control.....	24
2.2.4 Open Grid Service Architecture (OGSA).....	25
2.2.5 Other Related Work on QoS on Loosely-Coupled Distributed Systems.....	28
CHAPTER 3: HPC Background and Terminologies.....	30
3.1 The Communication Paradigms in Distributed Systems	30
3.2 Classification of Clusters.....	32

3.3	Resource Management System (RMS) and Scheduling	34
3.4	HPC Interconnects.....	35
3.4.1	Infiniband Architecture.....	36
3.4.2	Myricom Myrinet	38
CHAPTER 4: The HPC Systems Design and Performance Baseline		40
4.1	The System Architecture	41
4.2	High Performance LINPACK Benchmark.....	43
4.3	Intel IMBBenchmark.....	46
4.4	Performance Evaluation and Results.....	48
4.5	Conclusion.....	54
CHAPTER 5: Reducing Failure Rate Using Diskless HPC Clusters		55
5.1	Introduction	55
5.2	Related Work.....	57
5.3	The Diskless Cluster Design.....	60
5.4	HPL Experimental Results	61
5.5	BLAST Experimental Results	68
5.6	Conclusion and Future Work.....	72
CHAPTER 6: QoS and Performance Evaluation of the Infiniband Interconnect.....		74
6.1	InfiniBand and QoS.....	75
6.2	Infiniband Routing Algorithms	77
6.3	Performance Evaluation and Results.....	81
6.4	Conclusion.....	85
CHAPTER 7: Importing DDS-QoS into HPC and Grid Computing		87
7.1	The General Publish-Subscribe Framework in Data Distribution Services	88
7.1.1	QoS in DDS.....	90
7.2	The HPC-DDS Integration Model.....	92

7.2.1	Implemented Quality of Service Policies	97
7.3	Experimental Setup and Methodology	100
7.3.1	The Matrix Multiplication Application	101
7.3.2	The Primes Search Application	108
7.3.3	The Node-to-Node Streaming Application.....	113
7.4	Conclusion.....	116
CHAPTER 8: Conclusion and Future Work		117
8.1	Overview	117
8.2	Conclusion.....	118
8.3	Future Work.....	119
Appendix 1: The DDS QoS (as defined in www.omg.org).....		121
Appendix 2: The Matrix-Multiplication Pseudo-code Using DDS		127
Appendix 3: The Primes Search Pseudo-code Using DDS		128
Appendix 4: The Node-to-Node Pseudo-code Using DDS		129
REFERENCES		130
Vitae		136

LIST OF TABLES

Table 1: Performance numbers of different Infiniband technologies	37
Table 2: Myrinet performance of M3F-PCIXE-2 and M3F-PCIXF-2 HCAs.	39
Table 3: Test results for 32, 64 and 126 nodes LINPACK runs.....	46
Table 4: Temperature and power consumption for diskfull vs. diskless HPC	67
Table 5: Serial BLAST comparison using the two cluster configurations	70
Table 6: mpiBLAST performance benchmark using Drosoph database	70
Table 7: mpiBLAST performance benchmark using the Human genome database.....	71
Table 8: Infiniband routing Ping Pong Test (in MB/s).....	81
Table 9: Infiniband routing SendRecv Test (in MB/s).....	82
Table 10: Infiniband routing Exchange Test (in MB/s).....	82
Table 11: Communication overhead ratio in DDS while running on 32 nodes.....	107
Table 12: MPI vs. DDS Primes Search runtime while varying the number of nodes	111
Table 13: MPI vs. DDS Primes Search runtime while varying the inputsize on 32 nodes	112
Table 14: The delay when engaging a new node in DDS while running Primes Search	112

LIST OF FIGURES

Figure 1: PromisQoS Architecture	9
Figure 2: Distributed Broker Architecture for HPC, as proposed byN. Wang.....	24
Figure 3: The DPIM Architecture as proposed by Scott E. Spetka et al.	25
Figure 4: Interaction between Globus Toolkit components.	27
Figure 5: Globus Toolkit version 5 major components.	28
Figure 6: Generic HPC Cluster Architecture.....	33
Figure 7: The HPC scheduler workflow.....	35
Figure 8: The DDR Infiniband interconnect topology of a 126 nodes cluster.	42
Figure 9: The HPL file configuration for a 126 nodes cluster with N value set.....	45
Figure 10: IMB Ping Pong Test.....	49
Figure 11: IMBSendRecv Test.....	50
Figure 12: IMB Exchange Test	51
Figure 13: IMBAllReduce Test	51
Figure 14: IMB Reduce Test.	52
Figure 15: IMB Reduce Scatter Test.	53
Figure 16: IMB All Gather Test.	53
Figure 17: IMB Bcast Test.	54
Figure 18: HPL efficiency for diskless and diskfull HPC.	62
Figure 19: Execution speed in terms of GFLOPS for diskless and diskfull HPC.	63
Figure 20: HPL execution time for diskless and diskfull HPC.	63
Figure 21: Disk I/O measured at the disk node during the bootup of diskless compute nodes.	65
Figure 22: Network activities at the disk node during the bootup of diskless compute nodes.....	66
Figure 23: Infiniband Service Levels to Virtual Lanes mapping.....	76

Figure 24: Infiniband DAPL Architecture.....	77
Figure 25: Infiniband routing AllReduce Test.....	83
Figure 26: Infiniband routing Reduce Test.....	83
Figure 27: Infiniband routing Reduce Scatter Test.....	84
Figure 28: Infiniband routing All Gather Test.....	84
Figure 29: Infiniband routing Bcast Test.....	85
Figure 30: The general Publish-Subscribe modelwith persistence service.....	89
Figure 31: MPI vs. DDS layers	92
Figure 32: HPC-DDS integration model	94
Figure 33: The general HPC-DDS flowchart for implementing parallel programs.....	95
Figure 34: The HPC programming pseudo-code using DDS paradigm	96
Figure 35: The QoS policy file for our DDS-HPC design.....	99
Figure 36: MPI vs. DDS Mat. Mult.runtime when varying the number of nodes.....	104
Figure 37: MPI vs. DDS Mat. Mult.runtime when varying the matrices size on 32 nodes.....	105
Figure 38: The communication overhead for computing 1100x1100 matrices.....	106
Figure 39: Network delay when engaging a new node in DDSwhile running Mat. Mult.	107
Figure 40: The network utilization for running Matrix Multiplication on 32 nodes	108
Figure 41: Node-to-Node Throughput.....	115
Figure 42: Failing the receiver in DDS while running the Node-to-Node application.....	116

DISSERTATION ABSTRACT

Name: Raed A. Al-Shaikh

Title: Adopting QoS Real-Time DCPS Models and Other Reliability Measures on High Performance and Grid Computing

Major: Computer Science and Engineering

Date: February, 2012

In recent years, we have witnessed a growing interest in improving the reliability when running parallel batch jobs on the High Performance Computing (HPC) environments. However, existing distributed memory HPC systems do not provide proper quality of service (QoS) controls and reliability features because of two limitations. First, standard communication libraries such as Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) do not provide means for applications to specify service quality for computation and communication. Secondly, modern high-speed interconnects such as Infiniband, Myrinet and Quadrics are optimized for performance rather than fault-tolerance and QoS control. On the other hand, Data-Centric Publish-Subscribe (DCPS) model, which is the core of Data Distribution Service (DDS) systems, defines standards that enable applications running on heterogeneous platforms to control various QoS policies in a net-centric system. Notably, a number of DDS standards are comparable to those for High Performance Computing (HPC) systems. In this research, we present a comprehensive survey of the studies exploring the reliability factors of distributed computing in general and the Real Time Publish-Subscribe (RTPS) models for HPC and Grid computing in particular. We then investigate the QoS and reliability measures on the different HPC layers, such as the high speed interconnects and the diskless HPC. Finally, we present our model of incorporating DDS QoS and reliability controls into HPC. Our results show that DDS integration into HPC adds considerable overhead in terms of performance and network utilization when the application is mainly communication-bound, while the performance is comparable to those MPI-based applications when the program is computation-bound. In both cases, the solution is a viable option for those applications in which QoS is considered a priority, or for those HPC batch jobs that would run on commodity hardware, where the probability of failure is not negligible.

ملخص الرسالة

الاسم: رائد عبدالله الشيخ

عنوان الرسالة: تفعيل جودة الخدمة المطبقة بأنظمة النشر-والإشتراك في مجال الحاسبات فائقة السرعة والحوسبة الشبكية.

التخصص: علوم وهندسة الحاسب الالى.

سنة التخرج: صفر، 1433هـ

زاد الإهتمام في الآونة الأخيرة بايجاد الطرق الفعالة لزيادة الإعتمادية في تشغيل البرامج المتوازية على الحاسبات فائقة السرعة. حيث أنه تقتصر تقنيات الحاسبات فائقة السرعة حاليا بالتركيز على سرعة الأداء - ومن غير الإهتمام الكامل بجودة الخدمة أو الإعتمادية - مثل اعادة تشغيل البرامج تلقائيا عند حدوث الأعطال أو اضافة موارد مساعدة عند الحاجة. ومن ناحية أخرى، تعتبر أنظمة "النشر-والإشتراك" من أفضل النظم المتوافقة والمطبقة لجودة الخدمة والإعتمادية في نقل البيانات لتطبيقات المعالجة الموزعة كبرامج النظير-للتظير (أو ما يعرف بالند - للند). والجدير بالذكر أن مصطلحات وقوانين أنظمة "النشر- والإشتراك" الرئيسية، وكذلك البنية التحتية لها، هي نفسها الموجودة في مجال الحاسبات فائقة السرعة، مع اختلاف المهام. في هذه الرسالة، نقدم بحثا شاملا لإدماج جودة الخدمة والإعتمادية في مجال الحاسبات الفائقة السرعة وبطبقاتها المختلفة. كما قمنا في هذا البحث بتقييم ومقارنة فعالية الجودة والإعتمادية المضافة ومدى تأثيرها على الأداء عند تطبيقها في مجال الحاسبات فائقة السرعة. تستنتج هذه الدراسة بأن تطبيق جودة الخدمة والإعتمادية في مجال الحاسبات الفائقة السرعة قد يضيف حملا اضافيا في بعض التطبيقات التي تعتمد على التواصل الشبكي بشكل كبير، ولكن قد توفر سرعة مضاهية للتطبيقات التي تعتمد بشكل رئيسي على المعالجة المركزية. وفي كلا الحالتين، فإن تطبيق جودة الخدمة يتيح لمستخدمي الحاسبات الفائقة السرعة الخيار باضافة الإعتمادية والتحكم بجودة الخدمة، وخصوصا عندما يكون وجودها مهماً كما في تنفيذ بعض التطبيقات التي تستغرق وقتا طويلا عند تشغيلها على الحاسبات الفائقة السرعة.

Chapter 1

INTRODUCTION

In recent years, there has been a growing interest in the field of distributed computing, where diverse machines and sub-systems are interconnected to provide computational capabilities and execute larger application tasks that have various requirements. These environments may be of different types, including parallel, distributed, clusters, and grids, and they can be found in industrial, laboratory, government, academic, and military settings, and may be used in production, computing center, embedded, or real-time environments [3].

The drive behind the interest in distributed computing in general, and high performance computing (HPC) in particular, is the fact that they offer several advantages over the conventional, tightly-coupled supercomputers and Symmetric Multiprocessing (SMP) machines. First, High Performance Clusters are intended to be a cheaper replacement for the more complex/expensive supercomputers to run common scientific applications such as simulations, biotechnology, financial market modeling, data mining and stream

processing [14]. Second, cluster computing can scale to very large systems; hundreds or even thousands of machines can be networked to suit the application needs. In fact, the entire Internet can be viewed as one gigantic cluster [15]. The third advantage is availability, in the sense that replacing a "faulty node" within a cluster is trivial compared to fixing a faulty SMP component, resulting in a lower mean-time-to-repair (MTTR) for carefully designed cluster configuration [71].

Despite the advantages of these distributed systems, they present new challenges not found in typical homogeneous environments. One key challenge is the ever-increasing number of hardware components in today's HPC systems. This increase in the hardware components is drastically affecting the probability of hardware failures in such systems - and thus the productivity of the end users - since every single failure on such HPC clusters would cause the whole running job to abort, resulting in tens of hours of computations to be wasted.

Although this challenge is known, existing distributed memory HPC clusters cannot provide extensive QoS-based communication and reliability controls because of two limitations: first, standard communication libraries such as Message Passing Interface (MPI) and Parallel Virtual Machine (PVM), do not provide alternatives for applications to control the quality of service for computation and communication. Second, modern high-speed interconnects such as Infiniband, Myrinet and Quadrics provide high-throughput and low-latency communication. However, low-level messaging interconnects are optimized for performance rather than fault-tolerance and QoS control.

One of the attempts to address the lack of extensive QoS-based and reliable communication in generic distributed systems is the foundation of the Data Distribution Service (DDS) [22], which is the first open international middleware standard directly

addressing heterogeneous communication for real-time systems, utilizing the publish/subscribe communication paradigm. While the publish/subscribe model can be the solution for the generic heterogeneous computing, one research interest is to support DDS specifications and its pre-defined QoS reliability controls in the more-specialized High Performance Computing environments (HPC) and incorporate its most important QoS polices, which is one of the main aims of this research work.

Other efforts to address the HPC reliability and availability at scale is done though investigating the hardware failure rate in HPC systems, and studying the reliability and QoS controls in the different layers of HPC architecture [88, 89], such as the high-speed interconnect [63], the operating system, and the HPC scheduler [76].

The rest of the dissertation chapters are organized as follows: in chapter 2, we present a comprehensive survey describing the recent research done for incorporating QoS controls in HPC and distributed systems. Chapter 3 describes the main HPC components and defines the fundamental terminologies used in such systems. In chapter 4, we conduct a system baseline to set the stage for our performance and reliability measurements done in the subsequent chapters. Chapters 5 investigates the performance and reliability measurements for diskless HPC clusters, as one of the techniques to enhance the HPC reliability in the hardware layer, while chapter 6 explores the limited QoS and reliability measures in the Infiniband interconnect. In chapter 7, we present our work of incorporating the DDS QoS into HPC, as an approach to increase the reliability and QoS control on HPC systems in the middleware layer, and report on its performance. We state our conclusion and future work in the last chapter.

1.1 Motivation

The trend of today's High Performance Clusters and other loosely-coupled distributed systems is that they are increasing in terms of nodes and hardware components. To illustrate, looking at the top500 worldwide supercomputers [66], we may see that what used to be the #1 HPC cluster (i.e. The RoadRunner) in November 2008 had 129,600 cores, while in June 2010, the #1 Jaguar cluster had 224,162 cores. Noticeably, this increase in the number of cores would drastically increase the probability of hardware failures in such systems.

Several studies were conducted to explore the correlation between scalability in HPCs and failure rates [87,88,89]. B. Schroeder and G. Gibson [87] advocated that the success of Petascale computing will depend on the ability to provide reliability and availability at scale. In their research, the authors collected and analyzed a number of large data sets of failures from real large-scale HPC systems for a period of ten years. Specifically, they collected data about: (a) complete node outages in HPC clusters, and (b) disk storage failures in HPC systems. In terms of complete node outages, the authors identified that hardware is the single largest component responsible for these outages, with more than 50% of failures assigned to this category, while software is the second largest category with 20%. The remaining percentage is related to human, environment and network outages. Further, the node failure rate for large-scale systems can be as high as 1100 failures per year. Given this extreme rate, an application running on such systems will be interrupted and forced into recovery more than two times per day [87].

In terms of storage and hard drive failures, the authors found out that the average annual failure and replacement rate (ARR) for hard drives in HPC systems is between 3% and

5%. This means that in a cluster of 512 nodes, the average failure rate for hard drives is around 1-2 drives every two weeks, which matches our findings in [74].

The authors concluded their research by stating that “the failure rate of a system grows proportional to the number of processor chips in the system”. Furthermore, as the number of sockets in future systems increases to achieve higher Petascale and even Exascale systems, it is expected that the system wide failure rate will only increase [88].

Conversely, nowadays there is little attention on QoS-based communication and reliability control on HPC. The reason is that users have witnessed a dramatic increase in performance over the last 10 years with regard to the HPC systems. What used to take one month of computation time in the 2000, it is taking only a few hours to run in the current systems. This advancement made it easier for users to resubmit their applications after offlining (i.e. fencing) the problematic node/core, rendering the wasted hours of the crashed job to be neglected. The goal of this study, therefore, is to focus on the HPC QoS and reliability controls - and in different HPC layers - to benefit those very long users’ jobs that require large-scale clusters.

1.2 Objectives

As demonstrated earlier, the need for extensive QoS controls and reliable HPC environments is unavoidable when HPC jobs would last for several days or even weeks to run. Therefore, the main objective of this research is to enhance the reliability of HPC parallel jobs by first investigating the limited reliability and QoS controls in the HPC hardware layer, particularly through examining the Infiniband interconnect and the diskless HPC clusters. Then, we present our model of incorporating DDS QoS controls

into HPC middleware by experimenting with three different parallel applications of different computation nature (compute-intensive, communication intensive, and hybrid parallel applications), where a number of benchmarks are done to examine the effect of this integration in terms of reliability, scalability, throughput and fault-tolerance. As demonstrated in the subsequent chapters, the integration of DDS into HPC provides the ability to control QoS properties on HPC and Grids that affect performance, reliability, and fault-tolerance. To the best of our knowledge, this is the first research focusing on the integration of DDS QoS policies into HPC computing.

1.3 Contributions

The contributions of this research work are:

- **Present a comprehensive survey on the attempts made to incorporate QoS in Distributed Systems**

A number of studies were made tackling particular areas for expanding the Real Time Publish Subscribe (RTPS) model and other QoS standards to the high performance (HPC) and Grid computing environments. As described in chapter 2, we classify these studies into two main groups: studies that are geared towards tightly-coupled systems [26, 32], and other studies that are concerned with loosely-coupled decoupled environments [1, 2, 3, 12, 29]. Examples of such efforts are: Applying DDS for Large-scale Distributed Applications [1], Information Management for High Performance Autonomous Intelligent Systems [2], Grid Technology and Information Management for Command and Control [3] and Sensor Event Processing on Grid Environments [4].

- **Investigate the limited QoS and other reliability and performance-related issues in the HPC environment.**

To some extent, HPC does offer limited QoS and reliability controls in its multi-layer architecture. In this research, we shall investigate the QoS and reliability controls in the HPC environment through examining the Diskless HPC clusters, and the Infiniband QoS and routing techniques as specified by the Infiniband Architecture Specifications (IBA), using a Westmere-based HPC cluster.

- **Implement DDS Reliability and other QoS policies on the HPC and Grid environment.**

Existing distributed memory HPC clusters cannot provide QoS based communication because of the already mentioned limitations. Thus, this research also focuses on adopting DDS QoS and reliability policies into HPC and Grid middleware. This integration will provide the ability to control QoS properties in HPC and Grids that affect performance, reliability and fault-tolerance, and align the resources to the most critical requirements.

- **Perform Performance Evaluation for QoS HPC.**

A number of evaluation experiments are done to examine the effect of adopting DDS QoS on HPC environment, in terms of reliability, scalability, throughput and fault-tolerance. It is also part of this work to study the performance when using state-of-art HPC technologies, such as the Quad-rate (4x) Infiniband interconnect and the latest Intel Westmere processor.

Chapter 2

Recent QoS and Failure-Recovery Studies in Distributed Systems

Studies in this area have resulted in a number of proposals discussing QoS and failure-recovery and control on distributed systems and high performance computing, and their implementations and designs in the middleware layer. In this chapter, we classify these studies into two main categories: studies that are geared towards tightly-coupled systems [26, 32, 47, 78], and other studies that are concerned with loosely-coupled decoupled environments [1, 2, 3, 12, 29].

2.1 QoS in Tightly-Coupled Distributed Systems

2.1.1 PromisQoS

The PromisQoS developers [26] conducted research work that attempts to provide a

tightly-coupled cluster platform that can guarantee access to computational and communication resources to distributed applications by the means of adopting QoS. The authors have developed PromisQoS as an architecture that is capable of executing hard real-time parallel applications on Linux clusters while providing high throughput and low-latency communication using Myrinet interconnect. PromisQoS attains the objective by the use of message-passing API (MPI/RT), an in-house implementation of a Linux based real-time scheduler (Turtle) [26] and a deterministic low-level messaging library (BDM-RT) running on the Myrinet network.

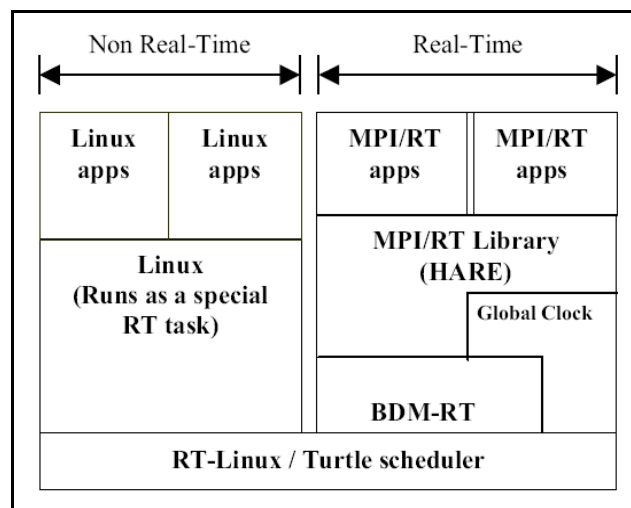


Figure 1: PromisQoS Architecture

PromisQoS uses the time-based channel of the Real-Time Message Passing Interface (MPI/RT), which was found to develop a standard that meets both requirements of high performance and QoS based communication without compromising portability. The design philosophy behind MPI/RT is that only MPI/RT implementers need to be experts

in platform-dependent communication features, while application developers provide only communication structure and QoS requirements of their applications. The MPI/RT transforms these QoS requirements to rules suited for the underlying platform. MPI/RT programs execute in two stages: non real-time and real-time. In the non real-time phase MPI/RT applications specify all resource requirements before execution, such as the number of communication channels and communication buffers, while QoS conditions can be specified as channel priority, upper bound on communication latency ...etc. Once all the requirements are specified, the application requests the MPI/RT library to perform an admission test by calling the MPIRT_Commit function [26]. The MPI/RT middleware then evaluates the request in terms of resource availability from the underlying system. If the requirements can be met, the resources are allocated and the application continues to run in the real-time phase. Although the applications need to specify their requirements up-front, i.e. before admission test, resources allocation is suspended until the admission test is complete. This two-stage arrangement assists the MPI/RT library to optimize resource allocation and make full use of the available system.

MPI/RT provides three types of QoS policies for channel message transfers [26]: time-based, event-based and priority-based QoS. In the time-based model, message transfers are scheduled at specific time intervals (specified by the application). The application typically specifies the time requirement in the form of a triplet – Period, Start-time and Deadline. In the event-based QoS, messages are scheduled based on their related events when they are triggered, and then messages are sent accordingly. In the priority-based schedule, messages are transferred based on the priority given to each node in the environment.

The second component of PromisQoS is “Turtle”, which is a modified version of RT-

Linux that adopts its scheduling policy and uses the “Critical deadline first” algorithm for its scheduling. In Turtle, as in the Earliest Deadline First algorithm (EDF), the periodic hard real-time tasks are represented by five parameters: the computation time for the task, the period of the task, the deadline of the task, the start time of the task, and the end time of the task. The Linux OS runs as a special real-time task that is guaranteed at least 1ms CPU-time every 10ms of real-time. Linux runs at all time periods that no other real-time task is ready to run on CPU. Also, Turtle provides mechanisms for early identification of possible deadline misses, thereby letting the application have an option of a graceful shutdown or recovery.

The last component of PromisQoS is BDM-RT, which is the real-time communication sub-system of PromisQoS over Myrinet [26]. BDM-RT consists of a message passing library that runs in kernel space, and a Myrinet Control Program (MCP) that has been designed to provide predictable message latency.

BDM-RT offers deterministic node-to-node communication latency at a minimal cost to performance by reducing contention of shared communication resources that include the PCI bus, LANai Myrinet processor time, and Myrinet switch ports. BDMRT supports techniques such as “blocking DMA”, a global clock implemented in software, and uses PCI DMA transfers for communication between host memory and on-board LANai memory. BDM-RT synchronizes PCI DMA transfers with the application's CPU schedules to control contention for the PCI bus between transmissions. For example, the BDMRT_Send call returns only when the message is completely transferred from host to LANai memory. BDM-RT holds the CPU for the duration of the host-to-LANai PCI transfer to ensure that other PCI transfer calls are not initiated by other real-time schedulers as well as by Linux processes. Similarly, at the recipient's side, sending

messages from the recipient's LANai memory to host memory is conducted only during the blocking call to BDMRT_Recv function. This guarantees that the PCI bus is always available for LANai-to-host transfer when the receiving task is active. Thus, by coordinating local resource accesses and network protocol calls with local CPU schedules, BDM-RT reduces contention for PCI bus bandwidth.

Although PromisQos is well suited to support the development of distributed hard real-time applications, Turtle does not support admission control. That is, it accepts all tasks and flags errors at run-time if it cannot satisfy the computational requirements of the admitted tasks. Furthermore, Turtle supports scheduling of tasks in the kernel space alone. Thus, all real-time applications execute in kernel mode.

2.1.2 QoS Aware MPI for Infiniband

H. Subramoni et al. [32] explore multiple options to effectively utilize the QoS concepts to enhance the tightly-coupled clusters performance. Their model is broadly classified into Inter-Job and Intra-Job schemes.

The rationale behind Inter-job scheme is that the jobs running in a large scale computing system can be in various types and have different QoS requirements, i.e., short-term or long-term jobs, jobs requiring high bandwidth and jobs sensitive to latency, ... etc. Treating all jobs equally will result in performance degradation for some, or all of the jobs. For example, the large application will create various congestion points in the network, resulting in bad performance for the delay sensitive application. To address this, the authors' solution is to provide job-level QoS. They define multiple "Service Levels" with varying performance metrics. Jobs can be mapped to different service levels. In particular, at the job launch time, the scheduler assigns a specific priority to a job. The

MPI library internally converts this priority to an InfiniBand service level (SL), which will, in turn, be used for all InfiniBand network communications. Based on this technique, the network elements will prioritize and classify the packets, ensuring the same QoS for the packet anywhere on the network.

As for the Intra-job scheme, the authors' modify the design of the MPI library to utilize different service levels for small and large messages. Their initial design is based on the Reliable Connected (RC) transport. According to the InfiniBand specification, the service level parameter can be changed only when initializing the Queue Pair (QP) (QP consists of a send queue and a receive queue. The send queue keeps instructions for sending data and the receive queue comprises of the instructions relating to the location of the receive buffer). Given this setup, the authors create two QPs for each process and associate one with the higher priority service level and another with the lower priority service level. The QP associated with the higher priority SL is used solely for small message transfers while the other SL is used for larger messages. As most of the control messages in MPI are small messages, granting higher priorities to small messages will not only enhance their performance, but also the performance of large messages whose progress relies on small control messages.

2.1.3 Other Related Work on Fault Tolerant MPI

Many fault tolerant and QoS-aware MPI implementations exist, such as LAM/MPI, Open MPI, WMPI (Windows implementation), and FT-MPI ...etc. The main difference between these implementations is the way they respond to process or nodes failures and their interaction with the hardware and communication layers. In particular, several implementations direct their fault tolerant techniques to the application level, while other

techniques target their implementation to the transport and data-link levels. It is important to mention that up to writing this research, all fault tolerant MPI implementations are still R&D and have not proven their practicality or to be generally available.

2.1.3.1 StarFish MPI

The initial implementation of StarFish runs on Linux and supports both Myrinet and Ethernet communication links [78]. Each node in a Starfish cluster runs a process, and all Starfish processes form a process group. Starfish processes maintain some data for each application running on the system, as well as some shared state that describes the existing cluster configuration and settings. These processes are responsible for interacting with clients, spawning the application processes, tracking and recovering from failures, and maintaining the system configuration [2]. Further, each application process is composed of 5 major components. These are: a group handler, which is responsible for communicating with the process in a node, an application part, which includes the MPI code to be run, a checkpoint/restart module, an MPI module, and a virtual network interface. These components communicate using an object bus based listener model [78]. To guarantee low latency and minimal impact on performance, the application part has a separate fast data path to and from the MPI module.

Starfish provides two forms of fault-tolerance for applications: The first fault-tolerant approach implemented in Starfish is checkpoint/restart. The checkpoint/restart module of Starfish is capable of performing both coordinated and uncoordinated checkpoints, which is either system driven or application driven [78]. Thus, when a node failure occurs, Starfish can automatically restart the application from the last checkpoint. The other fault-tolerance technique provided by Starfish is more application dependent, and it fits mostly

applications that can be easily parallelized by the system. For such applications, if a node that runs one of the application processes crashes, a notification signal is delivered to all surviving compute nodes. Once the surviving nodes receive the notification about the failing node, they redistribute the data sets on the living computes, and resume the application without any intervention from the user.

When an application is submitted to Starfish, the client determines the fault tolerant technique that should be used for this application, i.e., should automatic checkpoint/restart or nodes notifications be used, and some guidelines regarding how to choose the node on which a process will be started after a partial failure.

2.1.3.2 FT-MPI

FT-MPI is a fault tolerant MPI implementation that changes the semantics of the original MPI to allow the application to tolerate process failures [47]. In particular, FT-MPI can tolerate the failure of $n-1$ processes in an n -process job, provided that the application recovers the data structures and the data of the failed processes.

Managing failures in FT-MPI involves three phases: the first two phases are failure detection and notification. In these two phases, the run-time environment discovers failures and all remaining processes in the parallel job are notified about them. The third step is recovery, which consists of recovering the MPI library, the run-time environment, and the application.

There are two modes that can be specified when launching an FT-MPI application, these are [47]:

1) *The communicator mode*: this mode indicates the status of an MPI object after recovery. FT-MPI offers four different communicator modes that can be specified when starting the application:

- a) ABORT: this mode makes the application abort when an error occurs.
- b) BLANK: in this mode, failed processes are not replaced and all living processes will continue with the same rank as before the crash.
- c) SHRINK: in this mode, failed processes are not replaced. However, processes might be assigned new ranks after recovery.
- d) REBUILD: this is the default mode in FT-MPI. Failed processes are re-spawned, surviving processes have the same rank as before.

2) *The “communication mode”*: this mode indicates how to treat the messages that are on the way while an error takes place. FT-MPI provides two different communication modes for this situation:

- a) CONT/CONTINUE: in this mode, all operations which returned the error code MPI_SUCCESS will finish successfully, even if a process failure occurs during the operation.
- b) NOOP/RESET: in this mode, all ongoing messages are dropped and the application returns to its last consistent state. All currently ongoing messages are ignored.

2.1.3.3 Other Fault Tolerant Message Passing Implementations

As mentioned previously, MPI has a rich set of communication functions, which makes MPI favored over other implementations [72]. However, there are other popular parallel interfaces, such as PVM (Parallel Virtual Machine), and its various fault tolerant

implementations, such as DynamicPVM and MPVM that are able to provide the same MPI functionality. PVM is different than MPI in a way that it is built around the concept of a virtual machine, so it has the advantage when the application is going to run over a networked collection of hosts, especially if the hosts are heterogeneous. Moreover, PVM contains resource management and process control functions that are important for creating portable applications that run on clusters of workstations. L. Dikkenet et. al. [79] explore more on the differences between PVM and MPI.

For completeness, we view one of PVM implementations and study how it handles fault tolerant in parallel applications.

2.1.3.3.1 DynamicPVM

In general, PVM transmit communication messages using daemons, i.e. a message is first transferred to the sender's daemon, then forwarded to the daemon of the receiver and then delivered to the receiver. While standard PVM offers only a static process assignment to the application, DynamicPVM[79] provides dynamic process assignment and task scheduling, so that processes are migrated during runtime during failures. In particular, when a process failure is triggered in DynamicPVM, the local daemon on a new node prepares itself to receive the messages from the failing node, and sets its message buffer. The routing information of the local daemon on the old node gets updated so that messages which are still being sent to the old node are forwarded to the daemon on the new node. The sender daemon is informed about the new location of the process so that, in future, it sends directly to this process. One limitation in the current DynamicPVM implementation is that it is only possible to migrate one process at a time [79].

2.2 QoS in Loosely-Coupled Distributed Systems

In this section, we present the related work done for QoS in loosely-coupled distributed systems, as in the geographically-separated environments and large scale systems.

2.2.1 Applying DDS for Large Scale Distributed Applications

Most DDS implementations, such as OpenDDS [25], adopt multicast communication protocols to enable efficient and reliable data distribution to multiple recipients. However, setting up underlying multicast protocols means static pre-configuration of compute nodes, which defeats the purpose of data distribution, especially in *highly dynamic* WAN, HPC or computing Grid environments. One main concept in deploying dynamic DDS over a WAN-based net-centric environment is a “discovery framework” that supports the DDS infrastructure in automating the re-configuration of the underlying multicasting mechanisms to deliver the needed information to the right subscribers.

Nanbor Wang et al. [1] propose an adaptive discovery service for DDS-based applications in large-scale and highly dynamic Network-Centric systems. Their study is based on the need for the Department of Defense (DoD)’s to achieve the goal of information control, which resulted in adopting the standard of net-centric operations and warfare (NCOW). In their research, the authors identified the needs that earlier version of OMG DDS lacks to satisfy DoD’s NCOW requirements. These are:

- 1) **Increased dynamism** – which refers to the scalability of DDS; where not every node participating in an information exchange needs to know about every other entity in the exchange, or even the topology of the overall connection. In a net-centric environment, however, new and arbitrary nodes may need to join and participate in

information exchange in the domain. However, the nature of net-centric systems makes it difficult to pre-configure all systems as it will make the overall system vulnerable to minor changes, such as node joining and leaving multicast groups.

- 2) **Predictability and dependability** – Real-Time applications have rigid timing constraints. Environments with such constraints need to start reacting to incoming information as soon as they subscribe to the information. Similarly, information publishers anticipate the information they send to be handled immediately once published. Further, these highly-available applications must be able to tolerate faults, since nodes joining the information exchange may accidentally leave the network (e.g. hardware failure).
- 3) **More diversified environment** – most WAN-based net-centric systems are comprised of different interconnection technologies and hardware systems. Since net-centric middleware attempt to integrate the overall entities over the WAN, a pub/sub-based net-centric system should be compatible with all network types, including high-bandwidth, low-latency fiber optic connections; conventional LANs; and high-latency, narrow-bandwidth, and unreliable wireless links.

With these challenges identified, the authors addressed the essential requirements and needs for implementing an adaptive discovery service in pub/sub environment:

- 1) **Efficiently associating DDS entities with shared mutual interests.** There are many layers where DDS entities can participate in sharing data. Different DDS implementations require different techniques to define shared interests, and thus different programming techniques exist (such as domain participants in a Real-Time Publish-Subscribe (RTPS)). Regardless of each implementation, the components accountable for establishing the communication paths within the same domain share

the same interest based on the data available in this domain. It is important to note that identifying and matching these shared interests, however, not only involves a common topic, but also various QoS properties associated with the topic.

- 2) **Supporting robust communication hints.** WAN-DDS detection service should provide an automated way to reach the entities participating in the domain. For example, a combination of Internet address, port number describe the most basic information for reaching an entity using low level transports such as IP Multicast. More advanced multicast mechanisms used by a DDS implementation may provide better QoS support. For example, a domain may have a pool of prioritized entities that can be reached easily by providing communication hints and/or other automated mechanisms.
- 3) **Seamless integration with standards.** To enable information exchange between senders and receivers that use different implementations of DDS, a separate OMG DDS Interoperability Wire Protocol Specification defines the network protocol based on the RTPS wire protocol specification [5], which is a networking mechanism for industrial control and measurement based on LAN. All DDS implementations must be compatible with this specification when they communicate with other implementations.

Taking into account other characteristics of discovery services, the authors have identified the following key features needed for a WAN-DDS discovery service once information about other aspects of systems is available:

- 1) The discovery service should permit the use of other detection techniques to adjust to the already defined heterogeneous environment. For example, a hierarchical discovery service using a fixed set of servers may scale well for DDS applications running over

an enterprise network. On the other hand, a discovery service built on peer-to-peer protocols, may better serve DDS environments running over mobile ad-hoc network.

- 2) The discovery service should be incorporated with the underlying DDS configuration system transparently to allow dynamic reconfiguration of multicasting network. This incorporation would fulfill DDS applications from the need to use multicasting network that incorporates all possible participants by establishing smaller segregated multicasting networks.
- 3) An adaptive discovery service framework needs to tackle the scalability issue in RTPS protocol by decreasing the amount of messages sent by new participants to join a new DDS environment.

2.2.2 Information Management for High Performance Autonomous Systems

The publisher/subscriber architecture permits autonomous systems to interact without the need for a centralized brokering system. However, when each system is responsible for part of the whole brokering function, each system imposes cost for its local system resources and may reduce the functionality of each node of that system. This raises the concern of whether there should be defined rules where each autonomous system can utilize, in order to circumvent over-committing resources for brokering, such that the local nodes of that system are not affected.

Scott Spetka et al. [2] address the mentioned issues which affect autonomy in a publisher/subscriber system, where it can operate across HPC systems to allow load balancing and support processing for jobs that require more processors than may be

available on any other HPC systems. Computations can also be distributed across hybrid HPC platforms (i.e. forming a Grid) when part of the computation may be performed more efficiently on particular architectures. For example, some parts of HPC codes perform better on shared memory systems, like the IBM P6, while other parts of the computation can take advantage of message passing on Linux clusters.

However, system resources to support distributed brokering activities on behalf of remote dedicated systems have the highest impact on autonomy [2]. Specifically, system performance will be degraded due to the support for other communicating systems that share the common publisher/subscriber infrastructure. Further, brokering systems call for increased distributed state information, which results in increased bandwidth, storage and processing for each system participating in the domain. Therefore, the proposed HPC brokering system, implemented on an HPC cluster, provides a capability for offloading processing, thereby enhancing autonomy for brokers and improving QoS processing.

Autonomy in HPC Broker Implementation

In a brokering system, cooperating brokers can offload tasks to other brokers, which result in an improving global system performance. However, forcing specific operations on a broker may impact its ability to meet the pre-defined QoS controls.

In the authors' proposal, processing nodes may be committed to either do the brokering job or process other HPC programs within each HPC environment. When supplementary brokering nodes are required, for example due to increasing demands, they can be added to the HPC where the additional load will be supported. The decision to allocate the load to a particular HPC, and whether the allocated processing load should be accepted, does impact the autonomy of the system.

There are two scenarios for adding brokering resources: If HPCs make local decisions to voluntarily add supplementary resources to the pool, other HPCs would work in smaller number of brokering nodes, resulting in an unfair distribution of brokers. On the other hand, adding advance automation to increase the number of broker nodes causes additional overhead and can lead to group decisions to allocate additional brokers at a given HPC, resulting in breaking the autonomy for the HPC which must supply resources. In the authors' HPC publisher/subscriber implementation, increased communication requirements are supported by gradually reducing available processing resources to maintain an appropriate level of communications support for applications where processing is distributed across HPC systems. Figure 2 shows four HPC systems sharing resources to provide an execution environment for three parallel applications. One of the applications is performing digital signal processing, another is performing cryptanalysis and another is running the atmospheric analysis program. These applications rely on the publish/subscribe architecture, which is crossing all four HPCs, for their communications needs. Each HPC center is providing a subset of the compute nodes for use by the publisher/subscriber system in supporting the entire environment.

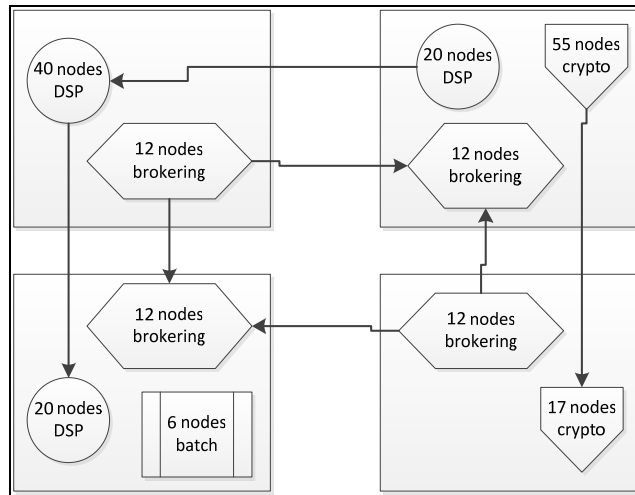


Figure 2: Distributed Broker Architecture for HPC, as proposed by N. Wang

2.2.3 Grid Technology and Information Management for Command and Control

Two paired systems, a Distributed Processing (DP) system and an Information Management (IM) system, are created to process data flows immediately and satisfy the need for data manipulation in the Grid environments [3]. Under DPIM, required compute power can be seamlessly added to the HPC infrastructure, in order to satisfy the increased processing requirements. Scott E. Spetka et al. [3] propose a combined system (DPIM) that addresses the performance and the scalability required for future Command and Control (C2) systems.

Figure 3 illustrates the author's DPIM Architecture. This architecture can be implemented on different HPC and distributed systems, where clients are allowed to communicate directly with grid services to call for various QoS controls and other system functions. For instance, a client can call for a QoS change in the publication rate for status information

or can send requests to the HPC job scheduler directly to check for compute availability and usability.

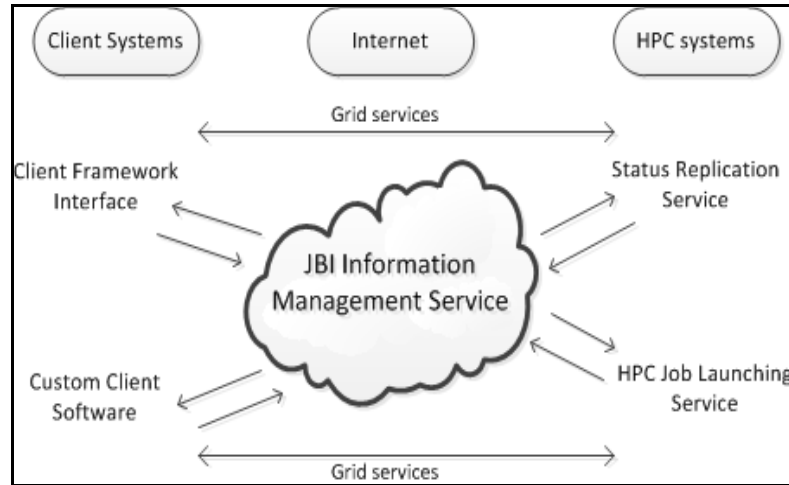


Figure 3: The DPIM Architecture as proposed by Scott E. Spetka et al.

Both the client and HPC infrastructure use the Java Business Integration (JBI) information management services [11] for publication and subscription. Clients send requests for service and subscribe to results. Requests and results can be considered information objects that are forwarded through the JBI, allowing the system to adapt to the changing infrastructure. Further, the HPC job scheduler receives requests, and then forwards them for execution and publishes the results. When an HPC sub-system encounters a partial failure, processing requests can be forwarded to other HPC systems that are defined in the domain.

2.2.4 Open Grid Service Architecture (OGSA)

The Open Grid Service Architecture (OGSA) is a distributed computing architecture that is built around different sub-systems. The main purpose of OGSA is to ensure

interoperability on distributed systems, and hide complexity of the different architectures. Alternatively, OGSA has been described as a refinement of the emerging Web Services architecture, specifically designed to support Grid requirement [34]. Recently, OGSA standards have been adopted as grid architecture by a number of grid projects such as the Globus Alliance [12].

The latest OGSA version 1.5 defines multiple of requirements, such as: how jobs should be executed within the grid, how data should be accessed, the minimum security requirements that should be included in the grid environment, and so on. One of these OGSA specifications is QoS assurance that focuses on availability, security, and performance. In general, they can be classified as:

- Service level agreement: QoS are defined through agreements between clients and servers prior to service execution. Standard protocols should be provided to form and manage such agreements.
- Service level attainment: mechanisms for monitoring service quality, assessing resource utilization, and planning for and controlling resource usage are needed.
- Migration: it should be possible to migrate executing services or applications to adjust workloads for performance or availability.

2.2.4.1 Globus Toolkit

Globus toolkitv.4.0 (GT4) [12], developed by Globus Alliance, is an implementation of the OGSA standard. The toolkit is basically a set of libraries and programs that address common problems that occur when building distributed system services and applications. Based on OGSA specifications, it addresses the following grid requirements by the following software:

1. Resource management: by using Grid Resource Allocation & Management Protocol (GRAM)
2. Information Services: by using Monitoring and Discovery Service (MDS)
3. Security Services: by using Grid Security Infrastructure (GSI)
4. Data Movement and Management: by using Global Access to Secondary Storage (GASS) and GridFTP.

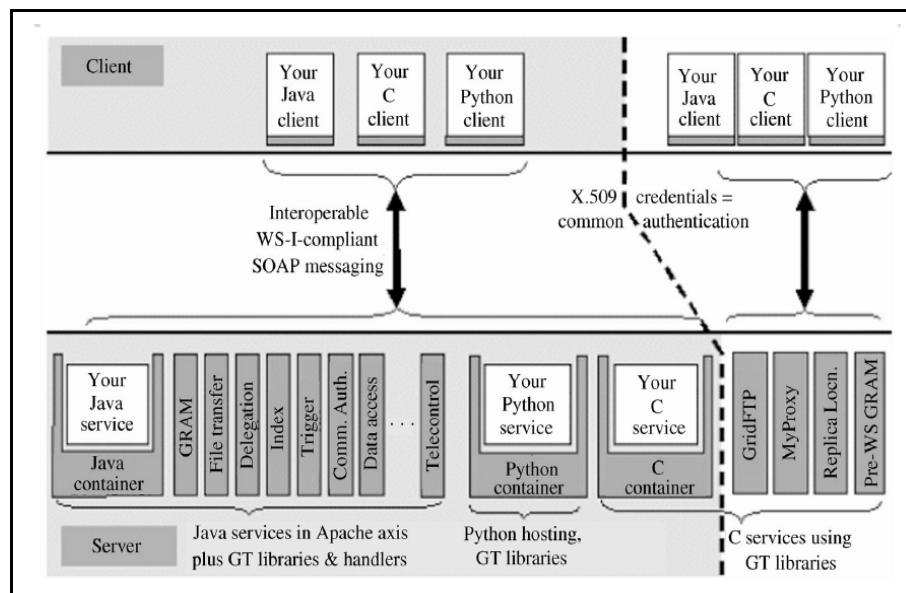


Figure 4: Interaction between Globus Toolkit components.

Figure 4 illustrates various components of GT5 architecture. Based on their functions, these blocks can be grouped into the following three sets of components:

- A set of *infrastructure services* to address infrastructure tasks, such as execution management (GRAM), data access and movement (GridFTP, RFT, OGSA-DAI[28]), replica management (RLS, DRS), monitoring and discovery (Index, Trigger, WebMDS),

credential management (MyProxy, Delegation, SimpleCA), and instrument management (GTCP) [28].

- Three *containers* that can be utilized to serve the user-defined applications written in Java, Python, and C.
- A set of *client libraries* to permit user programs to invoke operations on both GT4 and user-developed services.

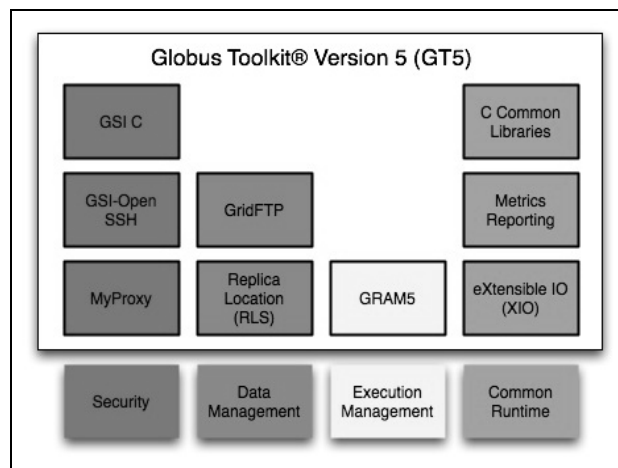


Figure 5: Globus Toolkit version 5 major components.

2.2.5 Other Related Work on QoS on Loosely-Coupled Distributed Systems

Other studies were done to apply QoS on loosely-coupled distributed systems, such as in multimedia applications and resource management. In this section, we present some of the work done to create middleware-based solutions that can provide QoS to specific environments.

Hafid et al. [27] designed a QoS manager that supports running multimedia applications on distributed systems. Based on the user's request, the QoS manager looks for potential system configurations, and selects the best option to run the applications, for example the

compute resources, and the network bandwidth. This search is also supported during the run of the multimedia. If a different system configuration is selected and the needed resources are reserved, the QoS manager then seamlessly adapts to the new system setup. Chu et al. [28] designed another Soft Real-time (SRT) system for multimedia applications. SRT supports multiple CPU service classes for real-time processes based on the usage pattern of these processes. They use the concept of ‘contracts’ to specify the CPU service level together with a flag used to reserve the required CPU time. If for example the number of frame changes, then the required CPU time would automatically adjust for some processes, and hence the contract parameters suit these changes. One obvious advantage of this adaptation is the capability to use just enough CPU time to execute the required processes, even in the case of partial failures in the environment.

In the context of resource management adaptation, Cardei et al. [30] presented a Real-Time Adaptive Resource Manager (RTARM), developed at the Honeywell Technology Center. RTARM middleware architecture focuses on real-time mission critical distributed applications, to better control the integrated services in such environments. RTARM focuses three scenarios where QoS for an application may change: (i) QoS decline when a new application begins, (ii) QoS improvement when an application terminates and releases resources, and (iii) feedback adaptation. Cases (i) and (ii) result in contract alteration because of such adaptation. Feedback adaptation, on the other hand, monitors the environment for the offered QoS and the actual usage of these services, and adapt accordingly. Similar to the other systems in described earlier, the main purpose of this architecture is to utilize ‘just enough’ resources, while ensuring adequate QoS and reliability controls on such distributed real-time systems.

Chapter 3

HPC Background and Terminologies

In this chapter, we shed some light on the common communication paradigms and technologies used in the HPC and distributed environments, with a further focus on the components having the most of the QoS and reliability controls defined. These are: the HPC job scheduler and the interconnect technologies. MPI will be discussed in more detail in chapter 7 when exploring the QoS controls in the middleware layer.

3.1 The Communication Paradigms in Distributed Systems

Three major middleware communication paradigms [25] have emerged for distributed computing, these are: client-server, message passing, and publish-subscribe communication model.

Client-server is fundamentally a many-to-one design that works well for systems with centralized information, such as databases, transaction processing systems, and central file servers. However, if multiple nodes generate and share information, client-server

architectures require that all the information be sent to the server for later redistribution to the clients, resulting in inefficient client-to-client communication. The central server is a potential bottleneck and single-point of failure. It also adds an unknown delay (and therefore indeterminism) to the system, because the receiving client does not know when it has a message waiting.

Message-passing architectures work by implementing queues of messages. Processes can create queues, send messages, and service messages that arrive. This extends the many-to-one client-server design to a more distributed topology. Message passing allows direct peer-to-peer connection; it is much easier to exchange information between many nodes in the system with a simple messaging design. However, the message-passing architecture does not support a data-centric model. Applications have to find data indirectly by targeting specific sources (e.g., by process ID or "channel" or queue name) on specific nodes. So, this architecture doesn't address how applications know where a process/channel is, what happens if that process/channel doesn't exist ...etc. The application must determine where to get data, where to send it, and when to perform the transaction. In the message-passing architecture, there is a model of the means to transfer data but no real model of the data itself.

Publish-subscribe scheme adds a different data model to messaging in the heterogeneous environments. Publish-subscribe systems basically "publish" information they have and "subscribe" to information they need. Messages logically pass directly between the communicating nodes. The fundamental communications model involves both discovery (i.e. what information should be sent) and delivery (i.e. when and where to send the information). This design adopts information delivery systems in everyday life (e.g. publications, broadcasts, magazines) [90]. Publish-subscribe systems are adequate for

distributing large amount of information quickly, even in case of unreliable communication means. Publish-subscribe can be efficient in some cases because the data flows directly from source to destination without the need for file servers or hubs. Multiple sources and destinations are easily defined within the domain, making redundancy and fault tolerance fundamental elements in its design.

In summary, client-server systems can be the best fit for centralized data environments and for systems that are service-oriented by design, such as file servers and web services. Client-server architecture, however, is not suitable for systems that involve many, often poorly-defined data paths.

Since message passing uses dedicated data-paths and prior information to know where the data resides, it suits systems with clear and simple dataflow needs. Therefore, message passing architecture is better than client-server middleware at free-form data sharing.

Publish-subscribe architecture, on the other hand, can be classified as a data-centric information distribution system. It suits well data-sharing environments by providing both discovery and messaging, and implements nodes communication simply by sending the data the publishers have to specific participants asking for this data.

3.2 Classification of Clusters

In this section, we classify the cluster systems according to their purpose: High Available Clusters, Load-balancing, HPC clusters and Grid Clusters:

High-availability (HA) clusters: High-availability clusters are designed primarily to increase the uptime of services that are running on the platform. They usually have hot-spare nodes, which are then utilized to provide service when the primary platform fails.

Load-balancing clusters: Load-balancing clusters attempt to distribute the load on many available nodes or sub-clusters. Although they are designed mainly for best performance, they usually include high-availability features as well.

High-performance (HPC) clusters: High-performance clusters are designed mainly to sustain the highest performance possible by segmenting a parallel program across many different nodes in the cluster, and are most commonly used in scientific applications. One of the more known HPC implementations is a cluster with commodity hardware nodes Linux as the OS and open source codes to implement the parallelism. This configuration is often called a Beowulf cluster, which is illustrated in figure 6.

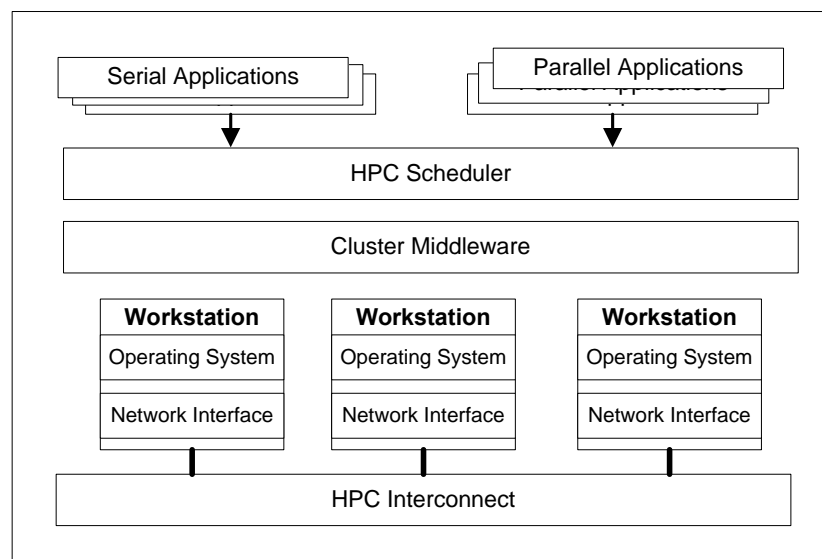


Figure 6: Generic HPC Cluster Architecture

Grid Clusters: In short, a grid is a collection of other nodes or sub-clusters. The main differences between grids and HPC clusters are that grids interconnect collections of computers or sub-clusters that may not process the same data or application, whereas HPC clusters are tightly coupled and function as a single image. In other words, Grids are

designed to manage jobs distribution to computers which in turn will perform the work independently of the rest of the grid cluster. In addition, resources such as storage pools may be shared by all the nodes, but staging results of one application do not affect other running applications on other nodes of the grid.

3.3 Resource Management System (RMS) and Scheduling

As loosely-coupled computing clusters grow in size and speed, providing proper controls running jobs becomes critical. This requirement is more important in larger clusters where chances for congestion are higher.

In an HPC cluster, a resource management system (or job scheduler) manages the processing load by preventing jobs from competing with each other for limited compute resources and enables effective and efficient utilization of resources available. This software consists of Resource Manager and a job scheduler. The scheduler communicates with the resource manager to get information about queues, loads on compute nodes, and resource availability to make scheduling decisions.

Resource Manager: Resource Managers perform basic node state monitoring, receive job submission requests and execute the requests on the compute node. Some resource managers have basic scheduling or policy controls. On complex cluster environments, a resource manager can increase the utilization of a system from 40% up to 70% by effectively managing the resources on the cluster.

Job Scheduler: Job scheduler guides the resource manager on the actions need to be taken, and when the schedule the subsequent jobs, and on which nodes. It imposes reservations and set priorities on running jobs, controls resources, and imposes policies

and events in line with the pre-defined HPC objectives. Thus, the scheduler permits users to submit jobs on the needed resources at the right time, and helps to speed up the overall scheduling of the workload. Figure 7 illustrates the interaction between the resource manager and the job scheduler in an HPC environment.

The RMS architecture is based on the client-server communication paradigm. To run a batch job, a user provides the job information to the system via the RMS client. This information consists of the physical job location, the location of the input data, the desired location for the results, the required resources such as CPU time and memory size, and so on. Once the job has been submitted to the RMS environment, it uses job details to place, scheduler and run the job in the appropriate way.

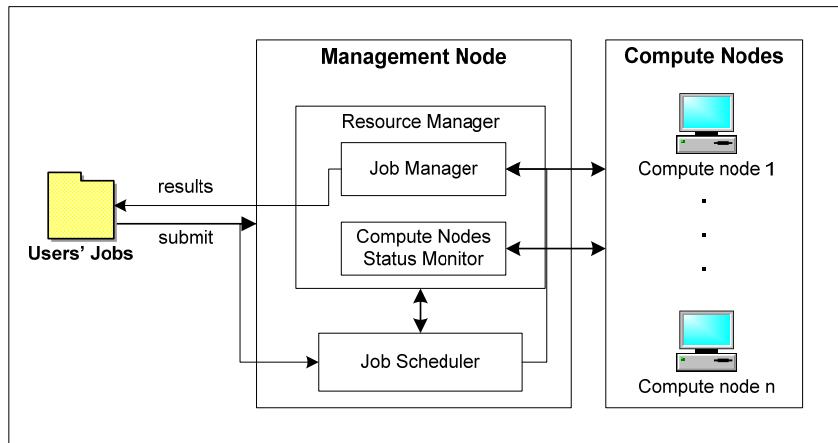


Figure 7: The HPC scheduler workflow.

3.4 HPC Interconnects

There are several network interconnects that provide ultra-low latency (less than 1 microsecond) and high bandwidth (several gigabytes per second). Some of these

interconnects provide flexibility by allowing user-level access to the network interface cards to perform communication, and also supporting access to remote processes' memory address spaces [35]. Examples of these interconnects are Myrinet from Myricom Inc., Quadrics and Infiniband [35]. All experiments in this chapter are done on the Infiniband architecture, which is one of the latest industry standards, offering low latency and high bandwidth as well as many advanced features such as Remote Direct Memory Access (RDMA), atomic operations, multicast and QoS [2]. Currently, available Infiniband products can achieve latency of 200 nanoseconds as hardware overhead for small messages, and a bandwidth of up to 4GB/s [35]. As a result, it is becoming increasingly popular as a high-speed interconnect technology option for building high performance clusters.

3.4.1 Infiniband Architecture

Infiniband is a technology that provides a high bandwidth I/O communication over a high speed serial data bus [33]. InfiniBand uses a switched fabric topology, rather than a hierarchical switched network like Ethernet. It is designed to directly route data from one point to another point through a switch, where all transmissions begin or end at a channel adapter (CA). The Infiniband serial connection signaling rate is 2.5 Gbit/s in single data rate (SDR) technology, 5.0 Gbit/s in double data rate (DDR) technology or 10 Gbit/s in quad data rate (QDR), in each direction per connection. Moreover, the links can be aggregated in units of 4 or 12, designated as 4X and 12X. However, Infiniband uses 8B/10B encoding (14B/16B for the recent FDR and EDR technologies), which implies four fifths of the traffic is useful, therefore DDR 4X link carries 20 Gbit/s raw, or

16Gbit/s of useful (users) data [35]. Table 1 summarizes the different Infiniband technologies with their associated theoretical performance numbers.

IB Technology	SDR InfiniBand Data rate	DDR InfiniBand Data rate	QDR InfiniBand Data rate	FDR InfiniBand Data rate	EDR InfiniBand Data rate
1x	2 Gbps	4 Gbps	8 Gbps	14 Gbps	25Gbps
4x	8 Gbps	16 Gbps	32 Gbps	56Gbps	100Gbps
12x	24 Gbps	48 Gbps	96 Gbps	168Gbps	300 Gbps

Table 1: Performance numbers of different Infiniband technologies

Infiniband uses a hardware-offload protocol stack. Extra memory copies that are sent from the application to an adapter can be avoided by the zero copy mechanism that optimizes the message transfer time. Moreover, Infiniband allows moving data from local memory to remote memory using RDMA (Remote Direct Memory Access), which allows the zero copy mechanism without involving the receiver host processor [35].

The number of user-kernel context switching and memory copies can be reduced by the direct access to the Infiniband HCA through the RDMA (Remote Direct Memory Access). Obviously, enabling communication between devices and hosts, without the traditional system resource overhead associated with network protocols, off-loads data movement from the server CPUs to the InfiniBand HCA. Through virtual lanes (VLs), InfiniBand offers traffic management, creating multiple virtual links within a single physical link that allows a pair of linked devices to isolate communication interference from other connected devices.

3.4.2 Myricom Myrinet

Myrinet was developed by Myricom to provide high speed network technology that is used to interconnect system to form a cluster of computing machines [62]. It has much less protocol overhead than Ethernet, which therefore, provides better performance using the host CPU. Due to the flexible size of the Myrinet packet, it can encapsulate other types of packets, each with an identifier, without an adaptation layer. The two low-level message passing provided by Myricom are GM and GX. They both support other message passing interfaces such as MPI, DAPL, and PVM, as well as emulated Ethernet [62]. GM and GX systems provide protected user-level access to the Myrinet reliable ordered delivery of messages, network mapping and route computation, in order to ensure robust and error-free communication.

Myrinet is available in two series: Myrinet-2000 and Myri-10G, as an alternative to Gigabit Ethernet and 10 Gigabit Ethernet (10GigE). Both Myrinet series employ the same network architecture and protocols. One good advantage of Myri-10G is that it can run on the same physical layer of 10GigE and its NIC can work as 10GigE as well. With MX unidirectional data rate can reach 495 Mbytes/s for Myrinet-2000 and 1.2 Gbyte/s for Myri-10G. [62]. Table 2 shows the performance matrix for the different Myrinet technologies.

Performance Matrices	M3F-PCIxE-2	M3F-PCIXF-2
Sustained one-way data rate for large messages	495 MByte/s	248 MByte/s
Sustained two-way data rate for large messages	912 MByte/s	495 MByte/s
Latency for short messages	2.6 μ s	2.5 μ s

Table 2: Myrinet performance of M3F-PCIxE-2 and M3F-PCIXF-2 HCAs.

Chapter 4

The HPC Systems Design and Performance

Baseline

As stated earlier, this research focuses on exploring the limited QoS and reliability capabilities that are available in native HPC systems, and also on adopting DDS QoS policies into HPC and Grid environments. Before exploring these objectives in detail, we first conduct a performance baseline for our HPC system to set the stage for our performance and reliability measurements done in the subsequent chapters.

Tuning HPC clusters is considered one of the most critical tasks in any HPC evaluation process, due to the fact that these HPC clusters consist of several tunable layers, such as the interconnect, the operating system, the system nodes, and the MPI stack. Any misconfiguration to any of these layers could result in wrong readings to the evaluation

process. To draw the baseline for our HPC cluster, we used two sets of benchmark tools, these are High Performance LINPACK (HPL) [49] and Intel IMB benchmark [64]. The first benchmarking tool is used to make sure that the actual TFlops (trillion Floating Point Operations Per Second) is very close to the theoretical figure with high efficiency percentage, which means that the cluster components (interconnect, OS, firmware configuration ...etc) are tuned to perform at their best configuration. The latter benchmarking tool is used to make sure that the MPI layer is tuned to deliver the best performance possible and is not affected by increasing the number of nodes participating in the communication. For HPL testing, it is sufficient to test only one cluster to conclude that our tune up is correct, since all the other three clusters would have the same. Intel IMB benchmark, however, will be carried on all the clusters since the MPI layer is tuned differently in every MPI implementation.

4.1 The System Architecture

To perform our experimental tests, a cluster of DELL PowerEdge M610 Blade Servers was used. The cluster consisted of 126 nodes with dual sockets and Intel Hexa-Core (Westmere) 2.93GHz processors. The operating system running on the nodes was RedHat Enterprise Linux Server 5.3 with the 2.6.18-128.el5 kernel. Each node was equipped with an Infiniband Host Channel Adapter (HCA) supporting 4x Dual Data Rate (DDR) connections with the speed of 16Gbps, and 1Gbps Ethernet connection. The Infiniband connection was used for the actual inter-process communication while the Ethernet connection was mainly used for the OS image boot-up and remote access. Each node also

had 12 GB (6 x 2GB) DDR3 1333MHz of memory, therefore, the total amount of memory the system had was around 1.5TB.

The physical layout of our cluster consists of six racks, each rack contains two chassis, and each chassis can host up to 12 blade nodes. That is, each rack supports 24 nodes. From each node we had a 4x-DDR Infiniband connection going to a central 144-port Qlogic Infiniband switch. Figure 8 shows the Infiniband interconnection design as described. It is important to mention that this design is considered non-blocking as each node guarantees to have the full 4x DDR 16Gbps interconnect speed.

Our Infiniband interconnect topology uses three types of switches. A top-level switch that connects two leaf switches, and the chassis switches. Each leaf switch can support up to 72 nodes, as it connects 3 racks with each rack supporting 24 compute nodes. Under this configuration, IPC communication among nodes of 32 and 64 clusters is localized to one leaf switch, but for the cluster of 126 nodes, the top-level switch is involved to support more nodes.

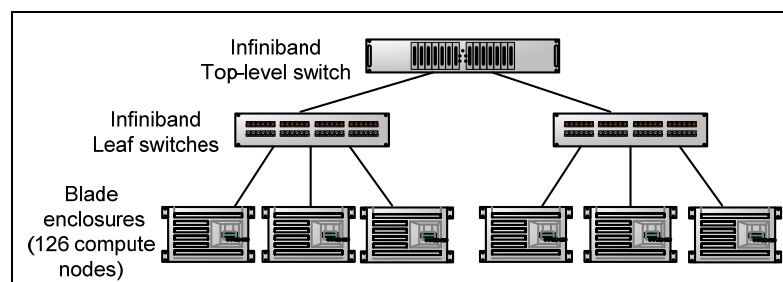


Figure 8: The DDR Infiniband interconnect topology of a 126 nodes cluster.

4.2 High Performance LINPACK Benchmark

To compare performance and build a baseline for our machines, we use the High performance LINPACK (HPC). HPL is one of the standard benchmarking tools for HPC. It is a collection of subroutines that analyze and solve linear equations and linear least-squares problems. The package solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square. In addition, the package computes the QR and singular value decompositions of rectangular matrices and applies them to least-squares problems. HPL uses column-oriented algorithms to increase efficiency by preserving locality of reference [49].

The HPL benchmark uses Basic Linear Algebra Subprograms (BLAS), which is a collection of routines to perform basic vector and matrix operations. Therefore, the HPL benchmark performance heavily depends on the implementation of the BLAS package being used. In our evaluation, we used the version provided by Intel's Math Kernel Libraries (MKL) since it is the one recommended to be used with Intel's Nehalem processor in order to make the most use of the processor's enhanced features.

Tuning the input file parameters for HPL can be a challenging task. For each cluster size that we were evaluating, a different tuned HPL input file had to be generated. We describe next HPL's main input parameters that were of interest to us to tune. We also discuss our methods and criteria in choosing these input parameters. The four parameters of interest were P , Q , N , and NB .

The $(P \times Q)$ value represents the size of the computational grid that HPL resolves, which is equal to the number of processors the system has. We have noticed that the best performances were achieved when we chose the value of $(P \times Q)$ to be as "square" as

possible as a grid shape, so we chose them to be approximately equal keeping in mind that Q needs to be slightly larger than P .

The next important parameter for HPL's input is " N ", which is the size of the problem. Our goal was to find the largest problem size N that would fit in our system's memory and would give us the tuned performance. We have chosen " N " to be close to our system's total memory size (in double precision 8 bytes), but keeping in mind not to make it equal to 100% of the memory size since some of the memory needs to be used by the system. It is important to note that when we choose a small value for " N ", this will result in not enough work performed on each CPU and will give us low performance results and low efficiency. While if we choose a value of " N " exceeding our memory's size, swapping will take place and the performance will go down. From our experiment with various values of N , the best performance was achieved when N was equal to 92% of the size of the system's total memory.

The last parameter that we discuss is " NB ", which is the block size in our grid. Usually block sizes giving good results are within the (96, 104, 112, 120, 128, ..., 256) range, and from our experimental runs, the value of 224 for NB has shown to give the best results compared to the various test runs we did with other values of NB .

Figure 9 shows an example of the HPL input file that was used for our 126-node benchmark runs with the tuned input parameters. Two other HPL input files were generated using the same techniques above for choosing the values of P , Q and N , one to be run on a system with 32 nodes and the other to be run a system with 64 nodes.


```

HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out output file name (if any)
file  device out (6=stdout,7=stderr,file)
1     # of problems sizes (N)
414400 Ns
1     # of NBs
224 NBs
0 PMAP process mapping (0=Row-,1=Column-major)
1     # of process grids (P x Q)
16 Ps
63 Qs
16.0  threshold
1     # of panel fact
0     PFACTs (0=left, 1=Crout, 2=Right)
1     # of recursive stopping criterium
4     NBMINs (>= 1)
1     # of panels in recursion
2     NDIVs
1     # of recursive panel fact.
0     RFACTs (0=left, 1=Crout, 2=Right)
1     # of broadcast
0     BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1     # of lookahead depth
0     DEPTHs (>=0)
2     SWAP (0=bin-exch,1=long,2=mix)
128   swapping threshold
0     L1 in (0=transposed,1=no-transposed) form
0     U  in (0=transposed,1=no-transposed) form
1     Equilibration (0=no,1=yes)
8     memory alignment in double (> 0)

```

Figure 9: The HPL file configuration for a 126 nodes cluster with N value set to 92% of available memory.

To evaluate the performance of our system, the theoretical peak execution speed of the system had to be calculated in GFLOPS to know the maximum theoretical speed which cannot be exceeded. In the Top500 supercomputers terminology [48], the maximum theoretical system speed is referred to as “Rpeak”, while “Rmax” is the actual speed obtained when HPL is run. The “Efficiency” of the system is the ratio of Rmax to Rpeak (Rmax/Rpeak). The efficiency can be affected by the underlying interconnect technology used for IPC communication among compute nodes, the amount of RAM available for

individual compute nodes, as well as the MPI implementation used for communication among cluster nodes of the system [57].

For example, to calculate the theoretical R_{peak} value for a system that consists of 126 nodes each with 8 Nehalem cores capable of 4 operations per cycle with a speed of 2.93GHz per core, the following formula is used:

$$\begin{aligned}
 R_{peak} &= CPU\ Speed(GHz) \times Total\ Cores \times Ops / Cycles \\
 &= 2.93 \times (8 \times 126) \times 4 \\
 &= 11813\ GFLOPS
 \end{aligned}$$

Once we calculated the theoretical performance (R_{peak}) for each of our various size systems (32, 64, and 126 node systems), we proceeded with running the HPL benchmark, using the corresponding HPL input file for each system size, to get the actual performance (R_{max}) of each system. Table 3 shows the results of our HPL tests and confirms the consistency of the system.

Number of nodes	HPL results	Theoretical results
32	2,598 Gflops	2,953 Gflops
64	5,256 Gflops	5,906.5 Gflops
126	10,395 Gflops	11,813 Gflops

Table 3: Test results for 32, 64 and 126 nodes LINPACK runs.

4.3 Intel IMB Benchmark

IMB 3.2 was used during this evaluation and it is the successor of PMB 2.2 from Pallas GmbH, Intel MPI Benchmarks 2.3, 3.0, and 3.1 [56]. This is a popular set of benchmarks which provides an efficient way to measure the performance of some of the important

MPI functions. It consists of three parts: IMB-MPI1, IMB-MPI2 and IMB-IO. We will focus on IMB-MPI1 which is used in our evaluation and it mainly replaces the formerly known Pallas benchmarks. The IMB-MPI1 benchmarks are classified into 3 groups, single transfer benchmarks, parallel transfer benchmarks, and collective benchmarks.

Single transfer benchmarks focus on measuring startup and throughput of a single message sent between two processes. For our evaluation we used the two benchmarks in this category, the ping-pong and ping-ping benchmarks. In ping-pong a process sends a single message to another process then the second process sends it back to the first process. For ping-ping benchmark, both processes send a message at the same time.

Parallel Transfer benchmarks focus on calculating the throughput of concurrence messages sent or received by a particular process in a periodic chain. For our evaluation we used two benchmarks in this category, the SendRecv and Exchange benchmarks. The SendRecv is based on the `mpi_sendrecv` function where each process in the communication chain sends to the process on its right and receives from the process on its left. In the exchange benchmark, each process exchanges data with both right and left process in the communication chain.

Collective benchmarks measure the time needed to communicate between a group of processes in different behaviors. There are several benchmarks of this category and the following is description of the collective benchmarks that was used in our evaluation:

- *Reduce*: each process sends a number to the root then the total number is calculated by the root.
- *Allreduce*: same as reduce but the final result is sent to all processes.

- *Scatter*: the root of the process sends a message to all processes. The size of the message equals to the chosen size * number of processes.
- *Reduce_scatter*: same as reduce but followed by scatter.
- *Gather*: all processes send the same message to the root.
- *Alltoall*: all processes send a message of a size equal to the chosen size * number of processes to all processes.
- *Bcast*: the root process broadcasts data to all processes.

4.4 Performance Evaluation and Results

In this section, we discuss our measurement criteria and interpret the obtained IMB benchmark results. In order to evaluate the performance of the two implementations of the MPI, the benchmarks were run on the cluster starting with 8 and up to 1,512 processes of the entire 126 nodes (remember that each node has 6x2 cores) [75].

We started with the IMB Ping Pong test, which is the classical pattern for measuring startup and throughput of a single message sent between two processes. In this test, we compared the latency for the two different types of MPI; they are about the same (~200 ns). As the message size gets bigger (>128k), Intel MPI starts to pick up and match the performance of MVAPICH. As shown in figure 10, both implementations are capable of delivering up to around 3039MB/s with a message size of 8M. We notice a dip in performance when using Intel MPI at a message size of 64k due to caching effect. This caching effect is common to some MPI implementations [86] whose large-message communication schemes suffer from high CPU utilization and cache pollution because of the use of a double-buffering strategy. This method results in two copies, one from the

source buffer (i.e. source node) into the copy buffer and another out of the copy buffer into the destination buffer (i.e. destination node). While this technique is useful for small message sizes (< 32 K bytes), it slows down the overall program for large message sizes, since it requires both nodes to actively take part in the transfer, which prevents them from performing useful application computation [86].

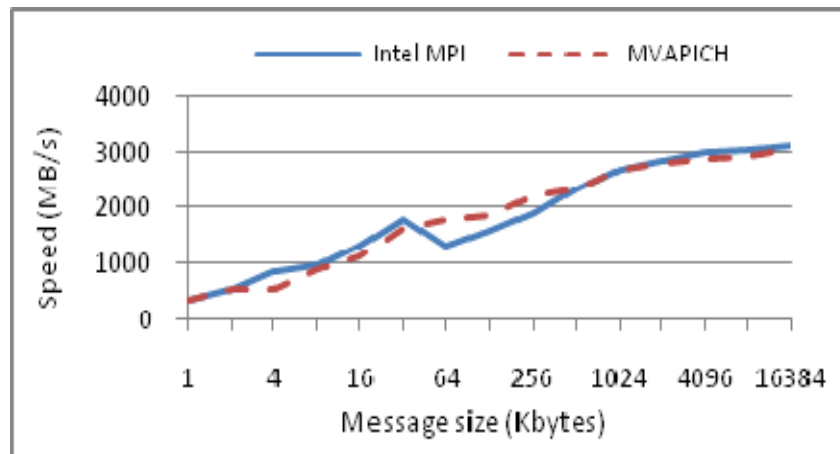


Figure 10: IMB Ping Pong Test

In the Pallas SendRecv test, each process sends to the right and receives from the left neighbor in the chain. The turnover count is two messages sample (1 in, 1 out) for each process. It is observed that MVAPICH gets faster between message sizes 4 Kbytes to 256 Kbytes, as shown in figure 11. The reason for this behavior is that while Intel MPI takes advantage of the double buffers for smaller message transfers, it requires to synchronize its source node double buffers with its adjacent nodes sequentially, starting with the left node and then synchronize again with the right node. Since the communication overhead takes precedence in the small message transfer phases, this effect is clearly shown in

messages smaller than 256 K bytes. For larger message sizes, both MPI versions turn to be equal at around 1600MB/s.

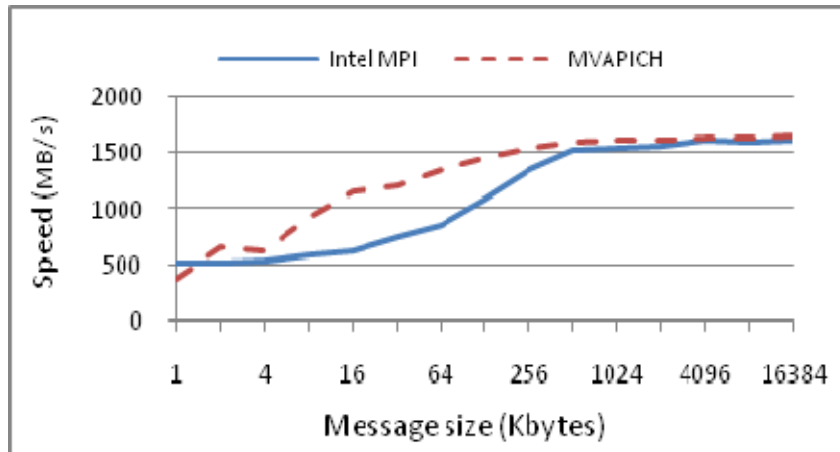


Figure 11: IMB SendRecv Test

Pallas Exchange test is a communications pattern that often occurs in grid splitting algorithms. The group of processes is seen as a periodic chain, and each process exchanges data with both left and right neighbor in the chain. Figure 12 shows the Pallas Exchange test and it is observed that for large size message $> 16\text{MB}$, MVAPICH performance starts to decrease, matching the performance of Intel MPI. Throughout the runs beyond 16 MB messages, the MVAPICH implementation started to swap to disk, causing the decrease in performance. We also notice a dip in performance at message size of 16k. This is due to caching effect in exchange since each processor will have essentially 32k.

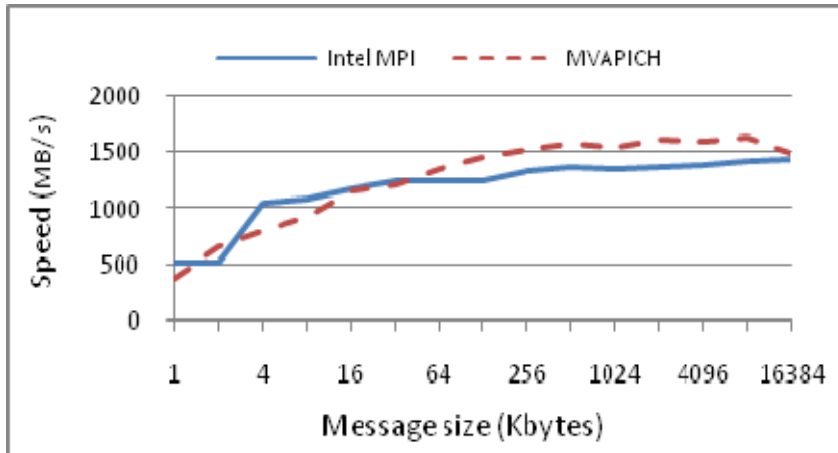


Figure 12: IMB Exchange Test

The following set of tests measures the time needed to communicate between a group of processes in different behaviors. Figure 13 shows IMB Allreduce test. Allreduce reduces vectors of length L float items from every process to a single vector and distributes it to all processes. As shown in the figure, the time increases as we increase the message size for all type of interconnects. Intel MPI performs slightly better when the message size exceeds 256KB.

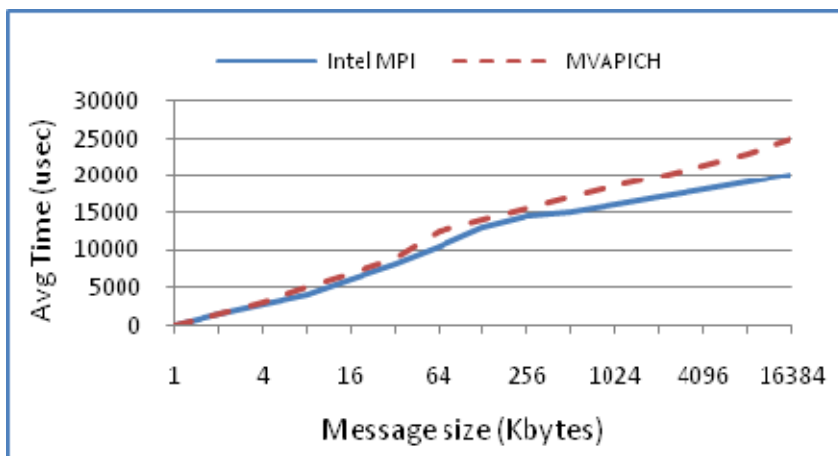


Figure 13: IMB AllReduce Test

Reduce test, which also reduce vectors of length L float items from every process to a single vector but in the root process. The root of the operation is changed cyclically. Clearly, Intel MPI performs better when the message size exceeds 2 MB, as shown in figure 14, as MVAPICH implementations started to swap to disk.

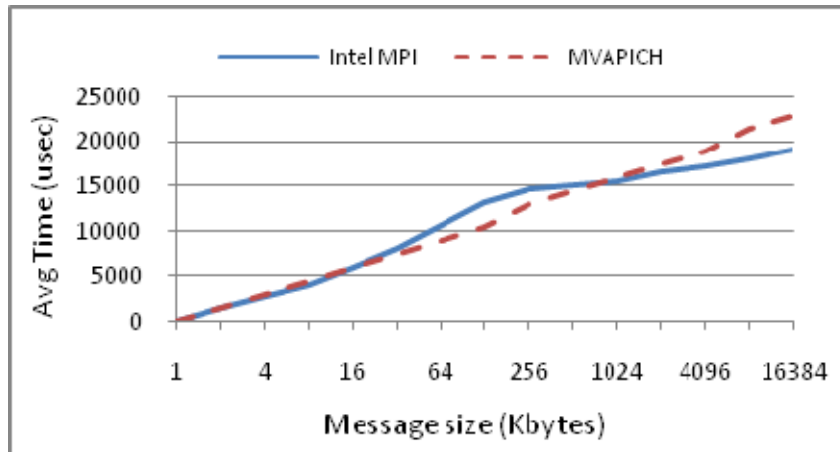


Figure 14: IMB Reduce Test.

Figure 15 shows the same case for Reduce Scatter test, which as well reduces vectors (of length float items) from every process to a single vector but, this time, the L items are split as evenly as possible between all processes. Again, Intel MPI suffers from the caching effect when the message size is about 512 K bytes, as a result of the use of the double-buffering strategy explained earlier.

On All Gather test, as in figure 16, every process sends X bytes and receives the gathered $X * (\text{\#processes})$ bytes from the receivers. As in Reduce test, MVAPICH implementation swapped to disk beyond 2 MB message sizes.

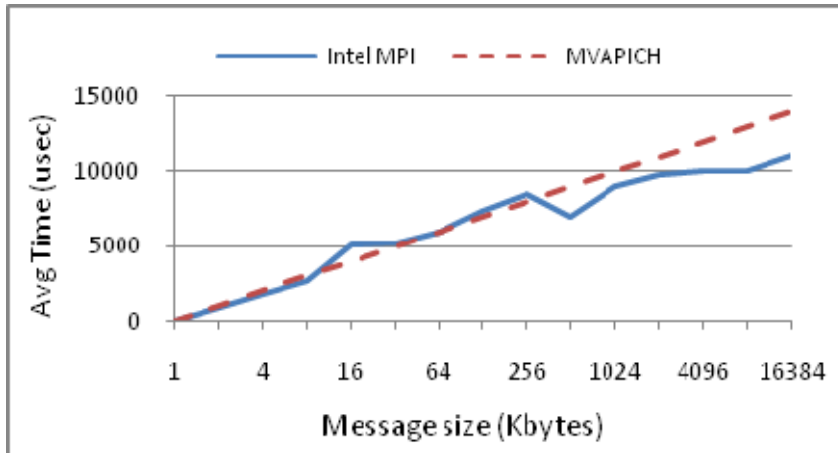


Figure 15: IMB Reduce Scatter Test.

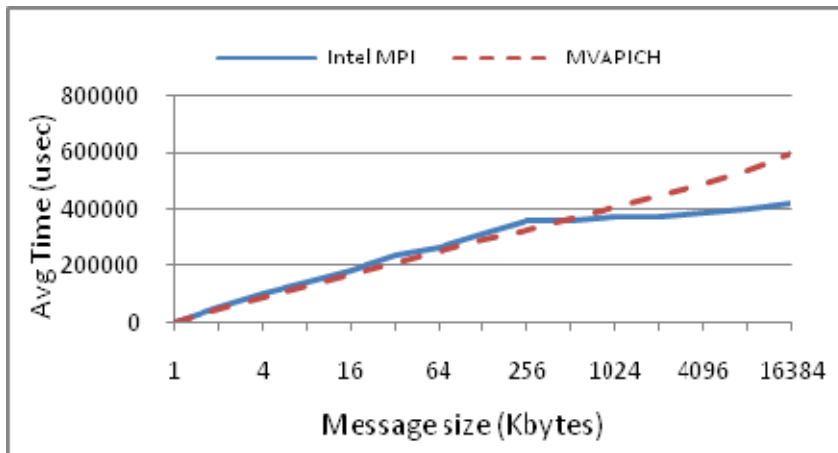


Figure 16: IMB All Gather Test.

In Pallas Bcast test, the root process broadcasts X bytes to all other processes, Intel MPI is still faster than MVAPICH, see figure 17. In particular, as we increase the message size, the number of performance difference increases. For large message size (16MB) we see the time to do bcast using Intel MPI is about 15000 usec where the time using MVAPICH is about 18000 usec.

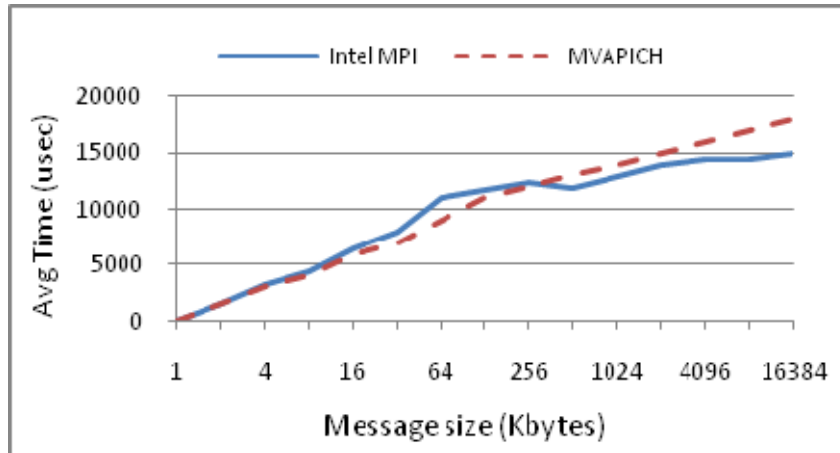


Figure 17: IMB Bcast Test.

4.5 Conclusion

In this chapter, we evaluated a large-scale Infiniband cluster, equipped with Intel’s latest Westmere processor using two MPI implementations. These experiments help us to draw the baseline for our HPC performance, in order to assure that it is properly tuned for our next experiments. The chapter presents the cluster configuration and evaluates its performance using High Performance LINPACK (HPL) and Intel MPI (IMB) benchmarks. Our results show that system scalability can still be achieved with up to 87% efficiency when considering the right combination of MPI, interconnect and CPU technologies. Further, our tests showed that in such a cluster, MVAPICH implementation excels in single-transfer communication where a single message is sent between two processes, while Intel MPI performs better in collective communication between groups of processes.

Chapter 5

Reducing Failure Rate Using Diskless HPC Clusters

5.1 Introduction

As illustrated in chapter 1, component failure in large-scale HPC installations is becoming an ever larger problem as the number of hardware components in a single cluster approaches a million. Additionally, power and cooling have become a major issue in designing High Performance Computing (HPC) solutions. Green Top500 [36] was established primarily to address this concern. For all those reasons, many HPC providers and HPC centers are striving to attain all these goals (increasing the systems reliability while decreasing the power and cooling requirements) with the least amount of side-effects possible. One of these attempts is researching the diskless HPC systems.

Diskless HPC clusters consist of compute nodes with no local disks. Instead, the compute nodes get their OS image during boot-up by using a centrally located device (or disk node) over a local LAN. In some designs, an internal network (e.g. 1 Gbps Ethernet) is used to provide not only inter-processor communications (IPC) among compute nodes but

also a medium for booting and file transmission. In other advanced designs, as exhibited in Section 6.3, the IPC communication is carried out on a separate extremely high-speed interconnect technology such as Infiniband or Myrinet. Each diskless compute node boots through the NIC's boot ROM with a small bootstrap, and then use either protocols such as BOOTP, DHCP, or NIC's Preboot Execution Environment (PXE) to get the OS image from a remote machine (in our case the disk node). Typically, a broadcast BOOTP request is first sent to a DHCP server to obtain an IP address. Then, the compute node sends a request to the TFTP server to get the boot image, point to the OS image, and start the booting process. During booting, all the necessary system files get transmitted through the network. The compute node completes the bootup when the remote file system is mounted as root file system (NFS_ROOT).

There are a number of obvious advantages to diskless clusters. First, the cost per cluster node becomes lower. Nowadays, the average cost of a server-level disk is about \$200 [74]. This translates to \$102,400 for a 512 nodes cluster. Second, diskless clusters have smaller footprints, i.e., lower power and cooling requirements. Third, cluster configuration and setup are consistent. In a diskfull cluster, system administrators spend considerable amount of time in developing and running script to ensure identical installations of OS images and files for all individual cluster nodes. In diskless cluster, since all nodes bootup over a network from a centralized disk server, identical OS images and installation files are ensured, thereby achieving system and file consistency across all compute nodes.

The real advantage to diskless clusters, however, is the reduced maintenance, or downtimes. With diskless systems, all mechanical parts – apart from the internal fans – are eliminated. For example, the mean time between failures (MTBF) of an internal disk

is reported to be 300,000 hours, or 34 years of continuous operation [74]. Thus, if there is a cluster of 100 nodes, 3 to 4 disks will be replaced every year. If there is a cluster with 12,000 nodes, then on average, a disk fails every 25 hours, or around every day.

On the other hand, there are clearly obvious drawbacks associated with diskless HPC. The most obvious drawback is the added network traffic. Since the compute nodes load their OS image by using a centrally located device over a local LAN, a diskless HPC cluster configuration generates more network traffic than a diskfull HPC cluster by reading the image over LAN. Moreover, if the network connection or the centralized OS image is not available, none of the compute nodes will be accessible. Solutions exist for these drawbacks [3], such as creating a RAM disk on each compute node by allocating part of the compute node's main memory as a partition for the file system. The RAM disk will be used for storing the most frequently accessed files. Therefore, the compute node can access some files from local memory instead of through the network.

Probably the most undesired situation that occurs during diskless HPC computation is disk swapping [38]. When the size of the main memory is not sufficient for the application running on the compute node, swapping to disk occurs and performance decreases dramatically. Swapping can have a particularly deleterious effect on performance in diskless clusters; all data accesses caused by swapping must travel through the network because the swap files are created on the centralized storage space.

5.2 Related Work

In this section, we present the related work and previous experiments done on diskless HPC implementations, compared to the traditional diskfull clusters.

Many benchmarks were conducted in the past to measure the performance of diskless HPC systems. Most these benchmarks [38,39,40] were done using the Ethernet LAN as a local cluster interconnect for communication, as well as for loading the diskless computes with the OS image. In our benchmark, however, we separate the compute nodes communication in a different Infiniband interconnect, while utilizing the LAN for booting the OS image.

Guler, et al. [39] configured two identical HPC clusters, except that one is diskless and one is diskfull, to compare performance and to identify what kind of applications is suitable for each configuration. On both clusters, the nodes had 2 GB of memory and dual Intel Xeon™ 2.4GHz processors. In the diskfull HPC cluster, each compute node had one Ultra3 SCSI hard disk, with the Red Hat® Linux 7.3 OS installed. The authors ran the first HPL benchmark on a single node of each cluster. One node was configured for the diskless HPC cluster and another for the standard HPC cluster. For larger problem sizes, the diskfull node performed approximately 5% better than the diskless one. Another HPL benchmark was run to measure the performance of 32 nodes on each cluster. This test allowed the team to determine whether any scalability issues would arise with the NFS server. Yet, no scalability, manageability, or operating problems occurred with this diskless configuration. Not only did the diskless configuration perform as well as the standard one, but the diskless cluster also outperformed the standard configuration by a few GFLOPS (around 2% increase). Moreover, the problem scaled very well from one node to 32 nodes with approximately 4.5 GFLOPS per node.

Chao-Tung Yang and Yao-Chang Chang [40] used a low cost Beowulf-type class HPC to prove the capability of their diskless cluster. The cluster consisted of one server node and eight computes as slave nodes. The server node had two Intel Pentium-III 690MHz

processors and 256MBytes of local memory, while the other eight nodes were dual Celeron-based SMP machines. Each individual processor was rated at 495MHz. The authors implemented a PVM-based matrix multiplication and also examined PVMPOV for parallel rendering. Their small-small sized cluster scaled very well when utilizing all the 8 compute nodes (16 CPUs). The authors, however, did not compare their diskless results with diskfull experiments.

James H. Laros and Lee H. Ward [43] performed their diskless HPC experiments on a 126 node test cluster that is comprised entirely of HP Alpha XP1000 nodes. Their cluster was built using the Bootable Hierarchy Architecture, a term they used to describe the layout of the NFS image used to support the diskless cluster. In their test system, the nodes served the role of leader or compute. The leader nodes are nodes that provide infrastructure services to other nodes. The compute nodes make up the bulk of the cluster. The authors' analysis only focused on the initialization of the cluster and the commands execution time when they are distributed among the diskless compute nodes. No experiments were done to demonstrate the performance of the cluster when running HPL or compute-intensive operations.

As described, many benchmarks were done in the past to measure the performance of diskless HPC systems. Most these benchmarks were done using the Ethernet LAN as a local cluster interconnect for communication as well as for loading the diskless computes with the OS image.

Our diskless HPC is distinguished from the earlier experiments in the following ways: First, our test cluster is more practical as it is using state-of-the-art hardware for nodes and advanced interconnect technology. In our experiment and setup, we separate the IPC communication among compute nodes from LAN communications. We use the popular

and industry standard Infiniband-interconnect technology for IPC communication. Gigabit Ethernet links are used only for management purposes and for having the compute nodes obtain their OS images to bootup. Second, our testbed out-scale other prior testbeds. Our cluster consists of a 126 compute nodes, with each node having a quad-core processor. These nodes with multi-core processors would impose high demands on the communication network. Third, in sharp contrast to other related and prior experimental work, we study and measure the performance of diskless clusters in terms of a variety of key metrics and measures of engineering and design importance. Such performance measures include execution speed, efficiency, network utilization, and disk swapping. For all of these measures, we study the impact of the cluster size on the performance measures. Fourth, the temperature and power consumption are also measured and reported in order to quantify the benefit of using diskless clusters in terms of power saving. Performance measurements are reported and compared for both diskfull and diskless clusters.

5.3 The Diskless Cluster Design

To perform our diskless vs. diskfull benchmark evaluation, we used the same cluster setup that was described in section 5.1 with some modifications. A disk node in the cluster was sharing a Linux ext3 file system as a network file system (NFS) among the cluster nodes. This file system contained the home directory of the test user that was launching the High Performance LINPACK (HPL) benchmarks as well as the HPL binary, Intel compilers, Intel Math Kernel Libraries (MKL), and Intel MPI libraries. The disk node was also

hosting the NFS_ROOT file system containing the operating system that will be shared among the diskless clients via network.

In case of diskless configuration, we had to increase the number of concurrent NFS threads running on the disk node hosting NFS_ROOT to handle the diskless clients' NFS requests, by adjusting the default value of "\$RPCNFSDCOUNT" variable in the NFS process file. This value depends greatly on the I/O load pattern, the network speed, the concurrency in the access and similar things. Red Hat recommends that systems administrators do a dynamic sizing of the number of nfsd threads [20]. During our tests, 64 concurrent threads were found to be the optimal number. Nevertheless, the NFS I/O load on the disk node went up to 55 (uptime command figure), or: $[(55 / 8 \text{ cores}) - 1 * 100 = 587\%]$ over-utilization when the diskless cluster first booted.

5.4 HPL Experimental Results

In this section, we present and discuss our HPL experimental results based on the same system and benchmark methodology explained in chapter 4. All reported measurements in this section are the average readings of three runs. The performance is measured and compared for both diskless and diskfull clusters while varying the cluster size. We study and measure the performance in terms of HPL efficiency, GFLOPS, HPL runtime and I/O node disk and network utilizations. We also examined the disk swapping effect on the diskless high performance cluster.

Figures 18, 19 and 20 show the HPC system efficiency, execution speed in GFLOPS and execution time (in seconds) of both diskfull and diskless configurations. HPL efficiency is obtained by dividing the theoretical peak speed (R_{peak}) by the maximal HPL speed

achieved (Rmax). As shown in Figure 18, diskless cluster provides comparable efficiency to the diskfull. In addition, as exhibited in both Figures 19 and 20, diskless slightly outperforms diskfull in terms of execution speed and run time.

It is important to note that the HPL efficiency was slightly reduced when using 126-node cluster for both diskfull and diskless configuration. The reason for this small reduction is due to the added communication introduced by the Infiniband top-level switch. For clusters of 32 and 64 nodes, the top-level switch is not involved and IPC communication among nodes is localized and only done via the leaf switch. However, when extending the cluster to 126 nodes, both top-level switch and the two leaf switches are involved in communication. As described earlier in, one leaf switch can support up to 72 nodes, as it connects 3 racks with each rack supporting 24 nodes.

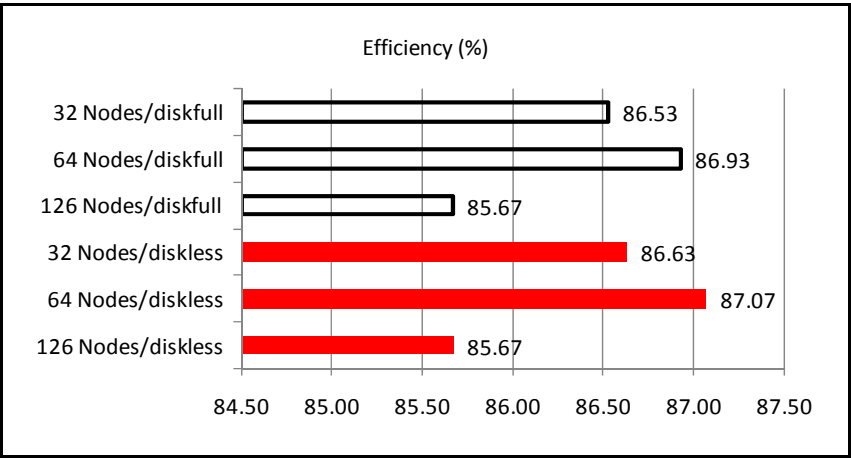


Figure 18: HPL efficiency for diskless and diskfull HPC.

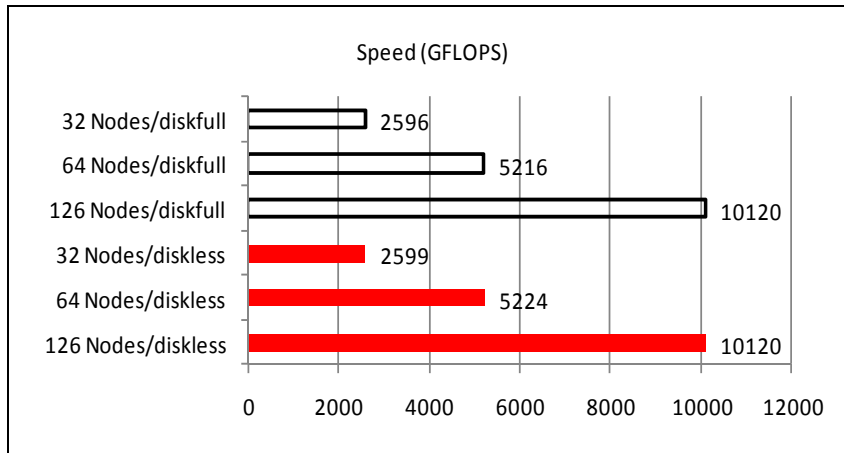


Figure 19: Execution speed in terms of GFLOPS for diskless and diskfull HPC.



Figure 20: HPL execution time for diskless and diskfull HPC.

It is observed from Figure 20 that the execution time of HPL increases when the cluster size increase, although larger cluster size would have more memory and more processing speed (i.e. GFLOPS). The reason for this increase is due to the way that HPL benchmark works. HPL provides three separate benchmarks that can be used to evaluate the performance of a dense system. The first is computing a 100 by 100 matrix, the second is for a 1000 by 1000 matrix, while the third benchmark, of a particular interest, is dependent on the algorithm chosen by the manufacturer and the size and speed in addition

to the available memory of the system being benchmarked [57]. The third benchmark was the one used. In other words, the benchmark execution size is made proportional to the size of the cluster in terms of memory, GFLOPS, and nodes. Large clusters will have larger benchmarks to run. This clearly explains the increase of execution time exhibited under clusters of 64 and 126 nodes.

Another experiment was performed where the cluster nodes were forced to swap to disk, by increasing the HPL required memory (i.e. N as an HPL input value) to 95%. The intension of this experiment was to measure the effect of disk swapping on the diskless cluster. While the diskfull system continued to run with typical swapping activities, Out-of-Memory (OOM) process was seen on the diskless compute nodes, causing the nodes to kill system processes randomly when they ran out of memory. The diskless HPC did not succeed running the benchmark when swapping is needed. Obviously, that is one limitation of running diskless HPC cluster.

We also measured the Gigabit Ethernet network utilization and disk I/O activities at the disk node throughout the experiment run time. It was noticed that obvious network and disk I/O activities occurred only when all the 126 nodes were booting up and loading the OS image via the network. However, after bootup, negligible activities were observed. Such observation is expected as access to disk node is needed only during the bootup of the 126 compute node, and IPC communication among computer nodes is carried out by the Infiniband links. The Red Hat Linux native *dstat* command was used to collect such statistics for a period of 5 minutes. Figure 21 illustrates the disk I/O activity on the image node while the diskless nodes are booting. As shown, the first read burst at 29s was caused by loading the kernel image into the diskless nodes, while the second burst at approximately 70s was caused by the start of actual loading OS files. Beyond 146s, the

OS image was entirely loaded into memory and minimal disk reads were taking place. On the other hand, disk writes continued as the diskless nodes were writing their states on the disk node, such as system and kernel logs (e.g. /var). These writes, however, did not exceed 8MB/s aggregate.

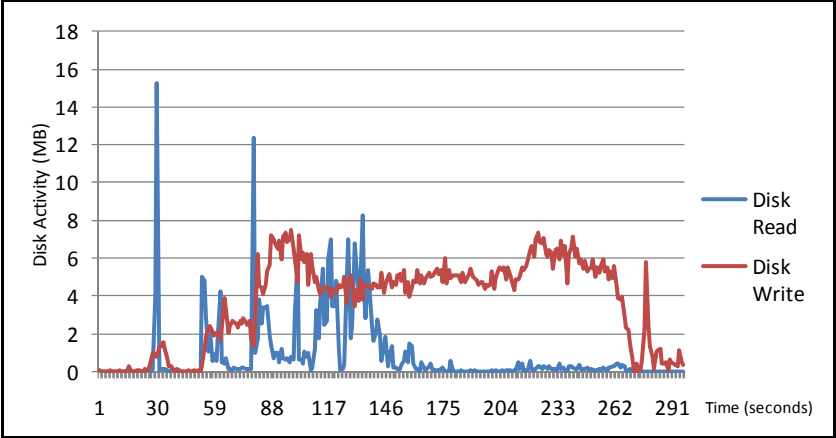


Figure 21: Disk I/O measured at the disk node during the bootup of diskless compute nodes.

Figure 22 shows the network IO on the disk node while booting the diskless compute nodes. As indicated, the network first high sending burst took place at 29s when the kernel image was being sent to the booting nodes to be loaded into their memory, while the second sending burst starting at 60s and lasting until 262s was caused by loading the actual OS files. These two bursts maxed to approximately 118MB/s, which is the maximum throughput of a 1Gbps connection. These bursts indicate a clear network contention on the disk node while the diskless nodes were booting up. In between the two bursts, the network activity goes down as the kernel image (initrd) scans for hardware in these diskless nodes, in which it does not need much of network activity. On the other hand and after completing the bootup process, the network activity decreased at 265s for

both network send and receive down to 500KB/s, as the diskless nodes had the OS image loaded into memory, and minimal access to disk node would be required. Such minimal access is primarily caused by Linux activities related to /proc and other virtual file systems for collecting and reporting system statistics.

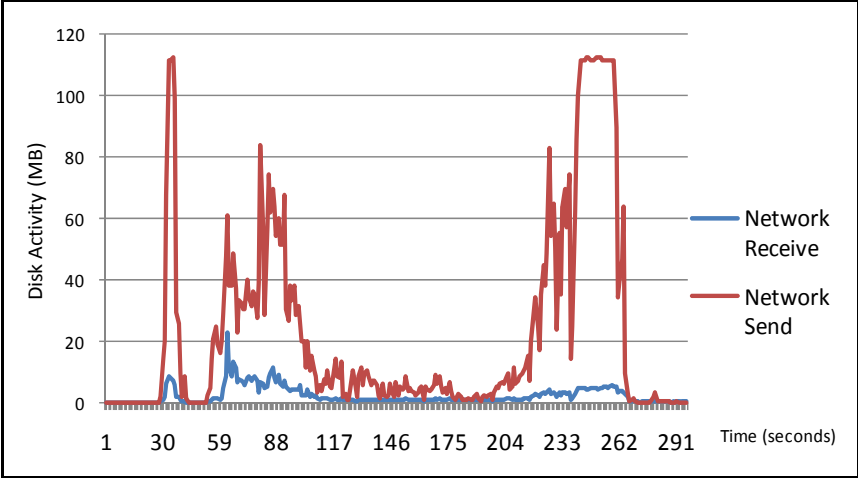


Figure 22: Network activities at the disk node during the bootup of diskless compute nodes.

During the time of HPL run on all 126 nodes, the temperature of both CPUs, the mother board’s temperature, and the power consumption for all 126 nodes were monitored while running diskless and on disk. DELL’s version of Intelligent Platform Management Interface (IPMI) tool [52] was used to collect such readings from all 126 nodes while the benchmarks were running on the nodes and fully utilizing the CPU and memory. In terms of temperature and heat dissipation, the diskfull and diskless readings were about the same at 18°C while performing the HPL test. In terms of power consumption, however, the diskless nodes operated with an average of 277 Watts per node, compared to 280 Watts per node for the diskfull configuration. That is about 2% saving in power. This

difference in power saving matches the hardware specifications of the published DELL internal disks power consumption [53] where they consume around 5 Watts per node. According to the United States' Department of Energy statistics for 2009, the average price for electricity in the USA is 10.01 cents per kW hour [58]. This would translate to an annual saving of U.S. \$31,567 for a diskless cluster consisting of 12,000 nodes compared to a diskfull cluster of the same size.

#Nodes/State	Avg. Cluster Temp. (°C)	Avg. Cluster Power (Watts)
126 Nodes/diskfull	18	280
126Nodes/diskless	18	277

Table 4: Temperature and power consumption for diskfull vs. diskless HPC

Furthermore, selected tests were conducted to examine the behavior of diskless system under various conditions, one of which is the effect of NFS_ROOT crash while the cluster is being utilized. Particularly, the NFS service was stopped (i.e. NFS_ROOT) on the disk node that was serving the OS image to the compute nodes. The NFS crash caused the compute nodes to completely stall with no network access. This is expected because during HLP run, there was still access to disk node, but minimal in the range of 500 KB/s. The system was back to normal operation, however, when the NFS service was back online. This behavior is expected as the NFS protocol is stateless [54]. That is, the NFS server should not need to maintain any protocol state information about any of its clients in order to function correctly. With stateless servers, a client needs only retry requests until the server responds; it does not even need to know that the server has crashed, or the network temporarily went down.

5.5 BLAST Experimental Results

Our second mechanism to measure our HPC diskless performance is by using the Basic Local Alignment Search Tool (BLAST) [80], which is a suite of programs designed to search all available sequence databases for similarities between a protein or DNA query and known sequences, using sequence alignment technique. Sequence alignment provides an accurate mapping between the elements in the two strings. Given a pair of strings, there are many possible alignments, and each one can be assigned a quality score; by giving positive scores to exact letter matches and negative scores to substitutions (i.e., where one letter in a sequence is mapped to a different letter in the second sequence) and gaps (i.e., where mapped letters are the same, but they occur in different positions in the sequences). This scoring system for matches and substitutions is normally done in the form of a “scoring matrix” in which the entries reveal the biological impact of the corresponding matches or substitutions [81]. The global similarity score of a pair of sequences is simply the score of the best (i.e., highest-scoring) alignment of the two sequences. It is important to mention that even if the global similarity score is low, there may still be portions of the sequences that match extremely well, and such local alignments are often of higher biological interest than the best global alignment.

Clearly, projects such as BLAST and other Biomedical Informatics projects in general, require data analysts and computing expertise as well as medical research talents to analyze and manage billions of data elements. On top, it has been estimated that the collective amount of genetic information doubles every twelve to eighteen months. This increased volume of information boosts the amount of computation required when

comparing an unknown sequence to the databases of known sequences. For those reasons, we elected to use BLAST tool in order to evaluate the performance of our diskless cluster. In order to evaluate the performance, the benchmarks were run on the first diskless node for the serial BLAST tests, and ranging from one node and up to 32 nodes for the MPI experiments. We used both the NCBI BLAST [80] and mpiBLAST packages [82] for the tests. Both implementations are freely available, whereas mpiBLAST is the parallel implementation of the tool. The main benefit to using mpiBLAST versus the serial BLAST is performance. mpiBLAST can increase performance by several orders of magnitude [82] while still retaining identical results as output from the serial BLAST. Particularly, through the use of database fragmentation, mpiBLAST performs a BLAST search in parallel. Database fragmentation partitions a database into multiple fragments and by distributing the fragments across many computational-resources (e.g. cluster-nodes), where each fragment can be searched simultaneously. Furthermore, by segmenting the query into multiple, independent searches, multiple BLAST searches can be simultaneously performed.

In our experiments, we used two different databases against our two 560 and 1,410 nucleotides input sequences to examine the scalability of our BLAST runs, namely: the Drosoph database for having the Drosophila sequences with a size of 120MB, and the human genomes database with around 9.8GB of sequence records. Table 5 shows the performance benchmark of the serial BLAST using both the diskless and diskfull configurations.

Cluster Type	Database	Elapsed Time (560 nucleotides sequence input)	Elapsed Time (1,410 nucleotides sequence input)
Diskless	Drosoph	6.3 seconds	10.1 seconds
Diskfull	Drosoph	6.5 seconds	10.5 seconds
Diskless	Human genomes	212 seconds	280 seconds
Diskfull	Human genomes	221 seconds	289 seconds

Table 5: Serial BLAST comparison using the two cluster configurations

In this serial BLAST benchmark, the diskless runs slightly superseded the diskfull configuration in all iterations. Specifically, the diskless cluster performed around 3% better than the diskfull version. The rationale behind this slight increase is the elimination of disk accesses when referencing the OS was needed.

No. of nodes	Diskless (560 nucleotides)	Diskfull (560 nucleotides)	Diskless (1,140 nucleotides)	Diskfull (1,140 nucleotides)
	time	time	time	time
1	7.2	7.6	11.7	12.4
2	3.9	4	9	9.5
4	2.5	2.6	7.9	8.1
8	0.7	0.73	3	3.1
16	0.3	0.31	1.5	1.5
32	0.2	0.2	0.2	0.2

Table 6: mpiBLAST performance benchmark using Drosoph database

Table 6 shows the performance of the mpiBLAST code using the Drosoph database. In this test, the mpiBLAST was compiled using MVAPICH while retaining all the default options. In addition, the database was fragmented prior in each run to a number of partitions that is equal to the number of the cluster nodes, in order to achieve the optimal performance.

It is noticeable that the diskless-runs on a single node took around 7.2 seconds when using the 560-nucleotides sequence, whereas it took only 6.3 seconds when using the serial BLAST (in fact this observation applies to all single MPI-node runs vs. serial BLAST runs). This effect is due to the fact that the MPI-based BLAST code has more routines and functions to call, making the code more complex, and thus more time to run. Another observation is the degradation in performance increase rate when reaching 6 nodes. This degradation is related to the additional communication overhead with respect to the computation time. This communication is lessened in the Human genome database runs as the computation time gets larger with respect to the communication overhead. Similar to the serial BLAST tests, the diskless cluster outperformed the diskfull setup in both database runs.

No. of nodes	Diskless (560 nucleotide seq.) time	Diskfull (560 nucleotide seq.) time	Diskless (1,140 nucleotide seq.) time	Diskfull (1,140 nucleotide seq.) time
1	230	239	296	303
2	121.7	127	164	168
4	85	88	105	108.1
8	22	25	35	37.2
16	15	16.1	23.2	24
32	4	4.3	5	5.3

Table 7: mpiBLAST performance benchmark using the Human genome database

Table 7 presents the performance using the 9.8GB Human genome database. Again, both the 560 and 1,140 nucleotides sequences were used in the runs. Overall, the performance of the diskless setup supersedes the diskfull cluster by around 2-4%. This percentage was lessened when exceeding 12 nodes as the MPI communication overhead became the main

contributor to the running time. It is also noticeable that that performance scalability is somewhat linear when using up to 12-15 nodes.

5.6 Conclusion and Future Work

Diskless HPC clusters are becoming a compelling alternative with greater benefits when compared to diskfull clusters, particularly in terms of reducing power consumption and failure rate. In this chapter, we have presented a design and a configuration of a state-of-the-art diskless cluster using Infiniband-interconnect technology. Our cluster consisted of 126 compute nodes equipped with quad-core processors. We measured and evaluated the performance of such a cluster in terms of key metrics which include overall efficiency, execution speed (in GFLOPS), and execution time. We also measured temperature and power consumption. These measurements of diskless cluster were compared to its respective diskfull cluster, considering three cluster sizes of 32, 64, and 126 compute nodes. Our results show that diskless clusters yield comparable performance to diskfull clusters, and in some cases outperform the diskfull. In terms of power consumption, diskless clusters clearly win with power saving of at least 3 Watts per node. On the other hand, diskless clusters have shortcomings. For one, diskless clusters require ample of RAM. It was demonstrated that if compute nodes are forced to perform disk swapping by decreasing their available memory, the compute nodes will freeze. Another obvious shortcoming is that the disk node in a diskless cluster can be a single point of failure. However, these two shortcomings can be alleviated by increasing the RAM of compute nodes and by having more reliable disk nodes that use advanced network storage technologies such as NAS and RAID technology.

As a future study, we plan to expand the size of the Infiniband diskless cluster to include 512 compute nodes, and then investigate its performance. We also plan to evaluate diskless cluster performance when using other popular benchmarks such as the Pallas MPI benchmarking tool [56] which gives more insight on MPI behavior and performance. We are also considering measuring the performance of diskless clusters when using 10Gbps Ethernet for IPC communication instead of Infiniband.

Chapter 6

QoS and Performance Evaluation of the Infiniband Interconnect

With the ever increasing number of scattered clusters around the globe, there has been a growing need to look for ways to interconnect these smaller clusters into larger, and potentially more powerful, HPC systems. Controlling the QoS for applications in such heterogeneous environments, however, has been one of today's challenges in this field, which in turn, has led system administrators to explore QoS for high performance networks.

Considerably, the Infiniband Architecture (IBA) Specification provides six different routing algorithms to better provide QoS and optimize the HPC internetwork traffic [84]. In this chapter, we present these algorithms and then evaluate QLogic's dispersive routing using a large-scale Infiniband cluster, equipped with Intel's latest Westmere processor. Our results show that whilst the default MinHop algorithm suits most of the serial and

point-to-point benchmarks, the dispersive routing algorithm exhibits improved performance when running specific computational and parallel transfer routines.

6.1 InfiniBand and QoS

IBA supports QoS at two levels: at the routing level, which is the scope of this chapter, and at the link level.

Fundamentally, IBA has three mechanisms to support QoS at the link level. These are: using service levels (SL), virtual lanes (VL), and virtual lane arbitration (VLArb). The Infiniband administrator may define up to 16 service levels (SLs) using the Subnet Manager (SM), but by default, this definition does not specify what characteristics or traffic type of each service level. Therefore, it depends on the implementation or the administrator how to partition the different existing traffic types among these SLs.

Once the SLs are defined, the network maintains a Service Level to Virtual Lane (SL2VL) mapping table that specifies to which VL we need to send packets belonging to one SL. The network maintains virtual lane arbitration (VLArb) table which defines two priority levels: high and low priorities. In each level, a weighted round-robin scheme of arbitration between the virtual lanes is defined. These two priority tables together ensure that each packet will be forwarded according to its designated SL across the network. A ceiling can also be defined on the number of credits that can be sent by all the high priority entries combined before allowing the low priority entries to be sent. This mechanism prevents starvation for those applications using the low priority queues.

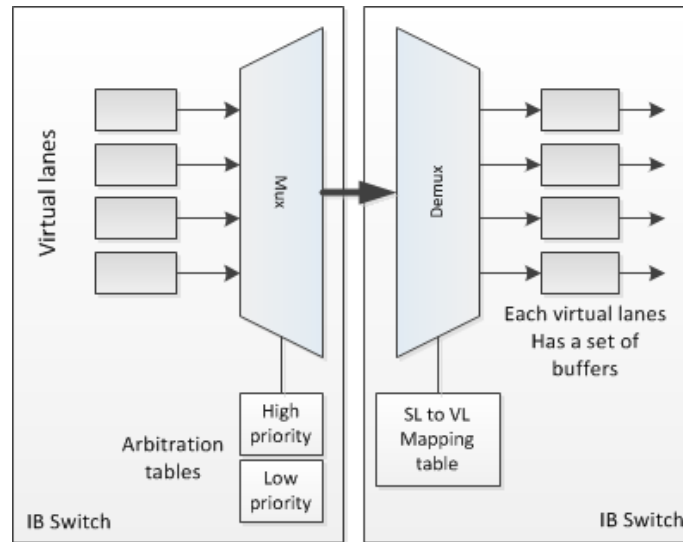


Figure 23: Infiniband Service Levels to Virtual Lanes mapping

Another newly introduced QoS layer in the Infiniband is the Direct Access Programming Library (DAPL) [83]. This library allows the MPI to use multiple fabrics independently and seamlessly. The rationale behind developing this library is that over the past several years, multiple networks appeared that provide Remote Direct Memory Access (RDMA) capabilities, such as the Infiniband, Myrinet and Quadrics. Some of these interconnects define their own APIs and of them do not define APIs at all. In addition, users of these networks who develop applications that take advantage of the RDMA semantics want to have a common set of APIs for all the networks. The work DAPL fills this need, as shown in figure 24. It is important to mention that depending on the design of the MPI layer, the performance impact of using the underlying DAPL might be negligible [83].

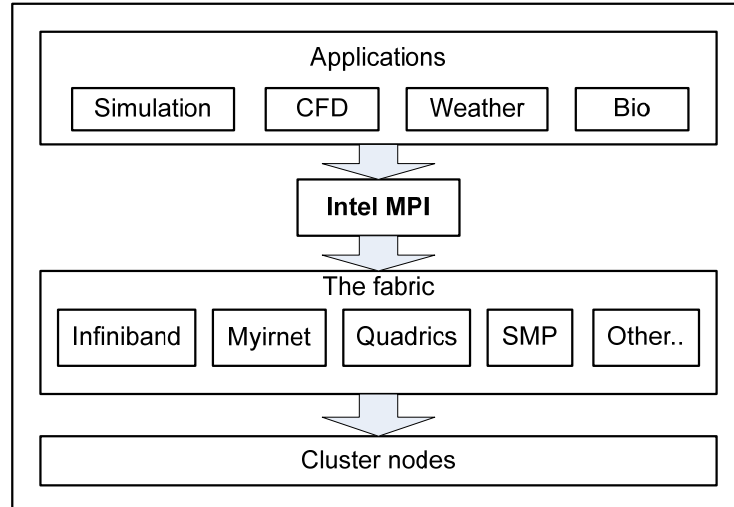


Figure 24: Infiniband DAPL Architecture

6.2 Infiniband Routing Algorithms

In this section, we briefly describe the five standard Infiniband routing algorithms, as specified by the Infiniband Architecture (IBA) Specifications, as well as QLogic's proprietary Dispersive Routing. The Mellanox manual [84] describes the first five routing implementation in more details.

The Min Hop Algorithm

The Min Hop algorithm is based on the minimum hops to each cluster node where the path length is optimized. The Min Hop is default algorithm, where it is activated when no other routing mechanism is specified. The Min Hop algorithm is simply based on two routines: computation of MinHop tables on every IB switch and Linear Forwarding Table (LFT) output port assignment for the forwarding mechanism.

The UPDN Algorithm

The Infiniband Up/Down (UPDN) routing algorithm is designed to avoid loops on the network. It is based on the minimum hops to each node, but it is bound to ranking rules. Loop-deadlock may occur when it is no longer possible to send data between any two hosts connected through the loop. Therefore, the UPDN routing algorithm should be used if the subnet is not a pure Fat Tree, and there is a potential for deadlocks.

In short, the UPDN algorithm initialization is done as follows: first, the firmware does an auto-detect for root nodes in the fabric. Then, the ranking process assigns all root switch nodes a rank of #0. Using the Breadth-First-Search (BFS) algorithm, the remaining nodes in the subnet are ranked incrementally. Then, another BFS algorithm is run in each node in the subnet. During the BFS process, the Forwarding Database (FDB) table of each switch node traversed by BFS gets updated, in reference to the root node of each sub-tree. At the end of the process, the updated FDB tables ensure loop-free paths through the subnet.

Fat-tree Routing Algorithm

The fat-tree algorithm can be selected if the Infiniband network is balanced or almost semi-balanced fat-tree of various types. As in UPDN, fat-tree is also an option to avoid credit-loop-deadlocks.

If the root Globally Unique Identified (GUID) file is not manually provided, the topology has to be pure fat-tree network for the algorithm to be initialized automatically. In addition, the tree rank should be between two and eight, and switches of the same rank should have the same number of UP-going port groups, unless they are root switches. Similarly, switches of the same rank should have the same number of DOWN-going port groups, unless they are leaf switches.

If the root GUID file is given as an argument, however, then the topology doesn't need to be pure fat-tree, and it should only comply with two conditions: first, the tree rank should be between two and eight, and that all the compute nodes have to be at the same tree level. The subnet manager performs a light sweep of the fabric it is managing every 10 seconds (by default). Thus, it is important to mention that if the IB networks do not comply with the above constraints (due to a link failure), the topology is no longer a “pure” fat-tree and the network will fall back to the default Min Hop routing.

LASH Routing Algorithm

LASH (Layered SHortest) routing algorithm is a deterministic shortest path routing algorithm, providing a deadlock-free routing within the Infiniband network. The concept assumes the presence of virtual channels divided into virtual networks (layers) to avoid deadlocks. Other than that, LASH routing requires no special functionality within the switches.

When computing the routing function, LASH analyzes the network topology for the shortest-path routes between all pairs of <sources, destinations> and groups these paths into virtual layers. Then, the algorithm begins a Virtual Level (VL) assignment process where a physical route is assigned to a layer if the addition of that route does not cause deadlock within that layer. This is achieved by maintaining a channel dependency graph for each virtual layer. If a deadlock exists, the algorithm creates a new virtual layer and continues the assignment process. It is noted that once this stage is complete, it is highly likely that the first layers processed will contain more paths than the latter ones. Therefore, to better balance the use of layers, LASH moves paths from one layer to another so that the number of paths in each layer is balanced.

It has been shown that for both regular and irregular topologies, LASH outperforms Up/Down [84]. The reason for this is that LASH distributes the traffic more evenly through a network via the VLs, avoiding the bottleneck issues related to a root node and always routes shortest-path.

DOR Routing Algorithm

The Dimension Order Routing algorithm is based on the Min Hop algorithm and so implements shortest paths. Instead of sending traffic via multiple paths that have the same shortest distance, DOR chooses among the available shortest paths based on an ordering of dimensions. Each port must be consistently cabled to represent a hypercube dimension or a mesh dimension. When there are multiple links between any two switches or nodes, they still represent only one dimension and traffic is balanced across them. This way, the algorithm is kept simple and dead-lock free, however, it eliminates path diversity in a mesh network and thus lowers throughput. Without path diversity, the routing algorithm is unable to route around faults in the network or avoid areas of contention.

QLogic's Dispersive Routing Algorithm

QLogic introduced Infiniband Dispersive Routing algorithm as part of its Infiniband Fabric Suite (IFS) version 6.0. The algorithm's idea is to optimize routing the HPC interconnection traffic using multiple paths between the HCA adapters. That is, instead of sending all the internetwork packets to a destination on a single path, the algorithm monitors multiple paths to a destination through the Infiniband Fabric Suite (IFS), distributes traffic over those paths, and ensures that the packets are reassembled in the proper order for processing at their destination. The technique also maximizes the speed of transfer, ensuring that the InfiniBand fabric is operating efficiently.

6.3 Performance Evaluation and Results

In this section, we discuss our measurement criteria and interpret the obtained IMB benchmark results. In order to evaluate the performance of the dispersive routing and compare it with the default MinHop algorithm, the benchmarks were run on the cluster starting with 8 and up to 1,512 processes of the entire 126 (remember that each node has 6x2 cores).

In table 8, we used IMB Ping Pong test, which is the classical pattern for measuring startup and throughput of a single message sent between two processes. In this test, we compared the latency for the two different algorithms. Noticeably, they are about the same (~200 ns). Further, both algorithms are capable of delivering up to 3100MB/s with a message size of 16M.

	128KB	256KB	512KB	1MB	2MB	4MB	8MB
Dispersive Routing	1625	1935	2325	2681	2820	2993	3039
MinHop Routing	1622	1933	2336	2684	2817	2992	3036

Table 8: Infiniband routing Ping Pong Test (in MB/s)

Tables 9 and 10 show the performance during the parallel transfer benchmarks. In the Pallas send recv test, each process sends to the right and receives from the left neighbor in the chain. The turnover count is two messages sample (1 in, 1 out) for each process. Pallas Exchange test, on the other hand, is a communications pattern that often occurs in grid splitting algorithms. The group of processes is seen as a periodic chain, and each

process exchanges data with both left and right neighbor in the chain. It is observed that both algorithms perform the same in these two tests.

	128KB	256KB	512KB	1MB	2MB	4MB	8MB
Dispersive Routing	1100	1351	1535	1544	1577	1622	1603
MinHop Routing	1078	1344	1539	1551	1580	1618	1608

Table 9: Infiniband routing SendRecv Test (in MB/s)

	128KB	256KB	512KB	1MB	2MB	4MB	8MB
Dispersive Routing	1255	1343	1362	1360	1366	1385	1420
MinHop Routing	1248	1338	1360	1370	1374	1402	1455

Table 10: Infiniband routing Exchange Test (in MB/s)

The following set of tests measure the time needed to communicate between a group of processes in different behaviors. Figure 25 shows IMB Allreduce test. Allreduce reduces vectors of length L float items from every process to a single vector and distributes it to all processes. As shown in the figure, the time increases as we increase the message size for all type of interconnects. In this test, the dispersive routing algorithm performs slightly better when the message size exceeds 1MB.

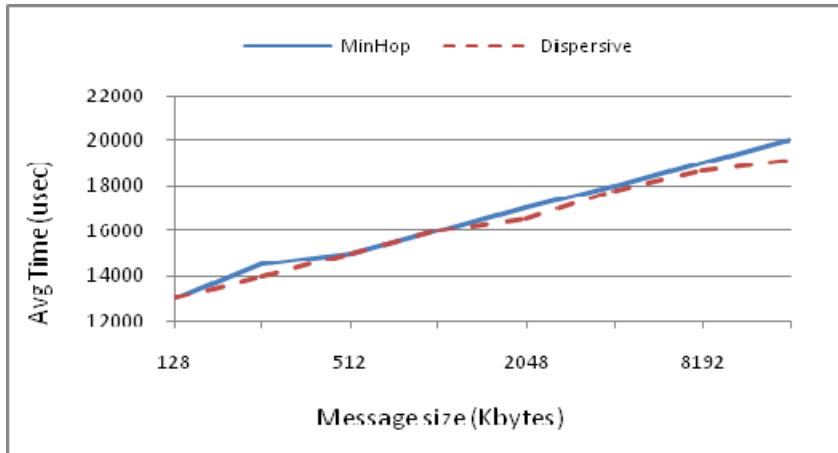


Figure 25: Infiniband routing AllReduce Test

Reduce test, which also reduce vectors of length L float items from every process to a single vector but in the root process. The root of the operation is changed cyclically. Clearly, dispersive routing performs better when the message size exceeds 1MB, as shown in figure 26. This observation - as well as AllReduce test - support the fact that dispersive routing works well in the case of parallel communication, or when the switch handles intensive broadcast transfers.

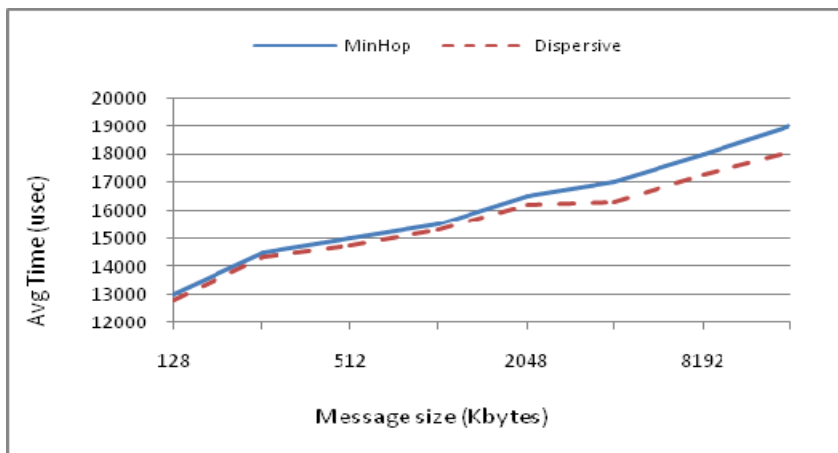


Figure 26: Infiniband routing Reduce Test

Figure 27 shows the same case for Reduce Scatter test, which as well reduces vectors (of length float items) from every process to a single vector but, this time, the L items are split as evenly as possible between all processes. On All Gather test, as in figure 28, every process sends X bytes and receives the gathered $X * (\#processes)$ bytes from the senders.

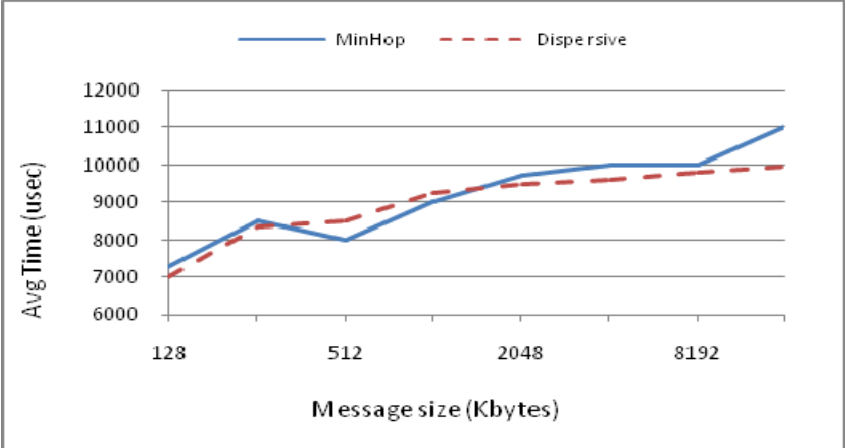


Figure 27: Infiniband routing Reduce Scatter Test

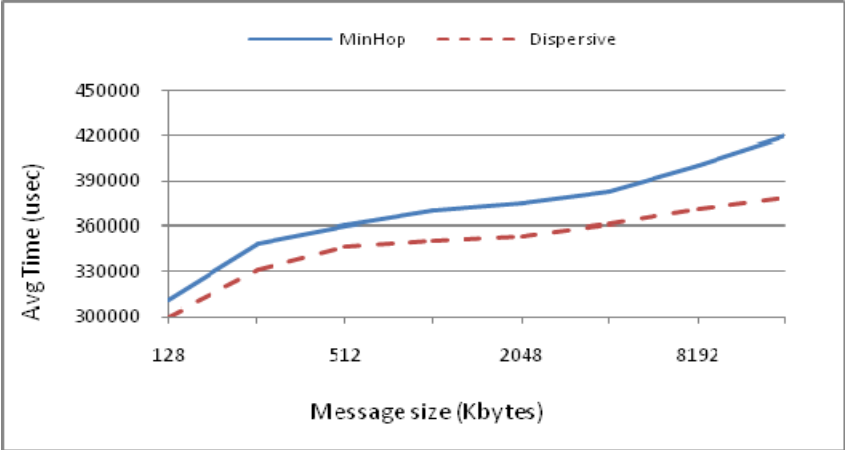


Figure 28: Infiniband routing All Gather Test

In Pallas Bcast test, the root process broadcasts X bytes to all other processes, Dispersive routing is still faster than MinHop, see figure 29. In particular, as we increase the message

size, the number of performance difference increases. For large message size (8MB) we see the time to do bcast using dispersive routing is about 14330 usec, while the time using MinHop is about 15000 usec.

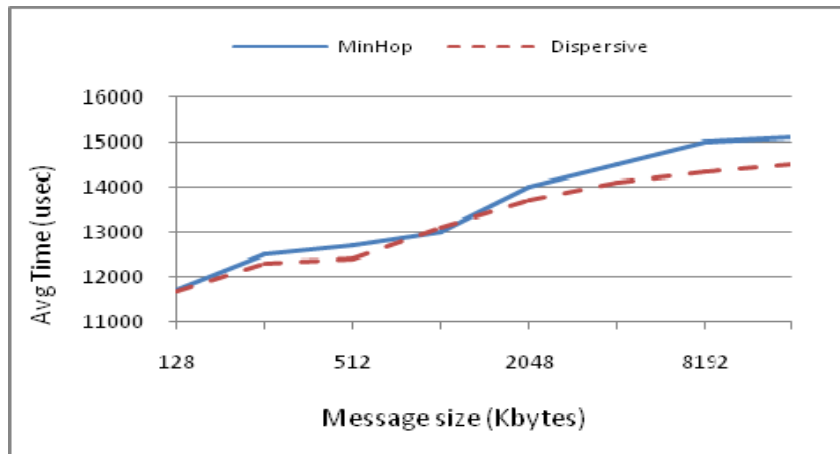


Figure 29: Infiniband routing Bcast Test

6.4 Conclusion

As high performance clusters (HPC) expand in terms of nodes count and processing cores, internetwork congestion and bottlenecks at the host and network levels become one of the main challenges in clustered computing. Considerably, the Infiniband Architecture (IBA) Specification provides six different routing algorithms to better optimize the HPC internetwork traffic. In this chapter, we presented these algorithms and then evaluated QLogic's dispersive routing using a large-scale Infiniband cluster, equipped with Intel's latest Westmere processor. The chapter presented the cluster configuration and evaluates its performance using HPL and Intel MPI (IMB) benchmarks. Our results show that whilst the default MinHop algorithm suits most of the point-to-point benchmarks, the

dispersive routing algorithm exhibits improved performance when running specific computational and parallel transfer routines.

Chapter 7

Importing DDS-QoS into HPC and Grid

Computing

In this chapter, we present our work of adopting DDS standards into HPC, in order to circumvent the MPI shortcomings and provide QoS and reliability controls in the middleware layer. It is also part of this research to examine the effect of adopting DDS QoS on HPC, in terms of scalability, performance and fault-tolerance, by testing three different computational models. All of our tests were conducted using state-of-art HPC technologies, such as the Quad Data Rate Infiniband interconnect and multi-core processors.

7.1 The General Publish-Subscribe Framework in Data Distribution Services

Data Distribution Service is a specification of a publish/subscribe middleware for distributed systems, created for the need to standardize a data-centric programming model for distributed systems [22]. The DDS standard, which is maintained by the Object Management Group's (OMG), offers a portable and scalable middleware infrastructure, designed for heterogeneous computing environments, to facilitate data transfers between data publishers and subscribers. DDS also provides various quality-of-service (QoS) policies that span across the multiple communication layers.

The DDS standard implements the publish/subscribe communication model for sending and receiving signals, commands, or even user-defined data between the nodes in the environment. As shown in figure 30, nodes that are sending data create "topics" of certain data types. These topics can be thought of as dedicated "data channels" that participants can join and share data through them. Using these topics, the different samples (which are different versions of data related to that topic) are communicated between the senders and recipients in the domain. The communication speed depends primarily on the DDS implementation and the communication medium that is used, where some DDS implementations [23] are reported to achieve latency as low as 65 microseconds between nodes, and high throughput up to 950Mbps, where any node can be a publisher, subscriber, or both simultaneously.

To enhance scalability, topics may have several independent data channels that are tagged with "keys." This technique allows recipients to subscribe to different data flows with a

single subscription. When data is received, the middleware arranges using the keys and deliver it to the recipients for processing.

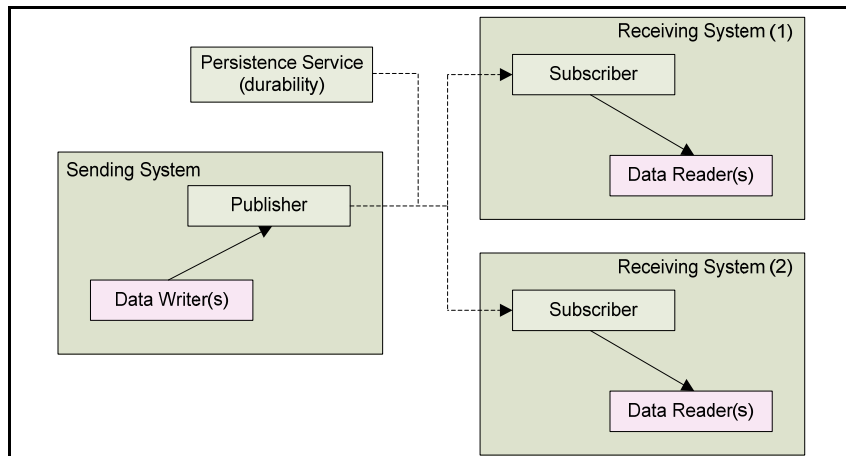


Figure 30: The general Publish-Subscribe model with persistence service

The main advantage of the DDS model is that applications can be entirely distributed. Also, the middleware handles the interaction and communication between the entities in the environment through the use of DDS libraries. Unlike MPI or other distributed computing middleware, the applications do not need information about the node in the domain, including their existence or locations. These features are achieved by the use of the automatic-discovery mechanisms as one of the DDS QoS parameters, and by specifying the behavior used when sending and receiving messages, including:

- Determining who should receive the messages,
- Where recipients are located,
- What happens if messages cannot be delivered,
- Maximum-waiting and minimum-separation times

7.1.1 QoS in DDS

As described earlier, having control over Quality of Service (QoS) is one of the most important features of the DDS standard. Each group of senders and receivers in the system can define independent QoS policies, whereas the middleware assures if the QoS agreement can be satisfied, thus establishing the communication or indicating an incompatibility error. Some information about some important QoS policies is highlighted below, and more policies are listed in Appendix 1:

Deadline: Data publishers may set the speed at which they can send data by offering certain update deadlines. By setting a deadline, the sender assures to send new updates at a minimum rate. Recipient on the other end may then request data at that or any slower rate.

Liveliness: This QoS determines whether an entity or a node is “active” (i.e. alive). The application can also be informed via a listener when an Entity is no longer responsive.

Strength: The middleware can have several publishers that are sending data using the same defined topic, each with its own “strength” indicator. Recipients receive the samples from the strongest active publisher. This technique provides automatic failover to the system, in a way that if a publisher with high strength parameter fails, then all subscribers can immediately switch to the second strong publisher in the same domain.

Durability: Publishers can declare "durability," a parameter that determines how long previously published data is saved. Late-joining subscribers to durable publications can then be updated with past values.

Lifespan: The purpose of this QoS is to avoid delivering outdated data to the recipients. Each data sample written by the publisher has an associated ‘expiration time’ beyond

which the data should not be delivered to any application or recipient. Once the sample expires, the data will be deleted from the caches as well as from the transient and persistent information storages.

Resource Limits: This QoS policy controls the resources that the middleware can use in order to meet the requirements imposed by the application and other QoS settings. For example, if the Publishers' objects are communicating samples faster than they are ultimately taken by the Subscriber's objects, the middleware will eventually hit against some of the QoS-imposed resource limits. Note that this may occur when just a single Subscriber cannot keep up with its corresponding Publisher. The behavior in this case depends on the setting for Data Distribution Service for Real-time Systems. That is, if reliability is set to BEST_EFFORT, then the middleware is allowed to drop samples. If the reliability, however, is set to RELIABLE, the middleware will block the Publisher or discard the sample at the Subscriber in order not to lose existing samples.

Partition: This QoS control permits partitioning the global domain into other smaller logical partition, possibly into different smaller domains. Therefore, in order for the recipient to receive data from the publisher, not only the Topic must match, but also they must subscribe into the same partition, or "domain".

Other QoS parameters exist to control the resources of the entire system, suggest latency budgets, set delivery order, attach user data, prioritize messages, set resource utilization limits and partition the system into namespaces.

7.2 The HPC-DDS Integration Model

To the best of our knowledge, the DDS QoS implementations usage has been limited to the generic heterogeneous computing environments. None of these attempts, however, were done specifically to incorporate DDS QoS policies into HPC batch jobs, and replace the de facto standard MPI middleware. Thus, one of the main aims of this study is to research the feasibility of incorporating DDS QoS policies into HPC environments and take advantage of the well-established reliability and fault-tolerance features in DDS.

When comparing the properties of both DDS and HPC, a number of DDS standards and requirements are similar to those for HPC architectures, as shown in figure 31. In particular, both DDS and HPC deal with intercommunication models, middleware layers, hardware infrastructure, and timing-related and QoS issues.

Moderate QoS	MPI	DDS	Extensive QoS
Moderate QoS	Compilers	Compilers	Moderate QoS
Limited to moderate QoS	Interconnect Subnet Manager	Interconnect Subnet Manager	Limited to moderate QoS
Limited QoS	HPC Interconnect (Fabric)	HPC Interconnect (Fabric)	Limited QoS
Limited QoS	HPC Hardware	HPC Hardware	Limited QoS

Figure 31: MPI vs. DDS layers

In DDS basic model, commutation between participants is achieved by having six essential entities, these are [90]: *DomainParticipant*, *DataWriter*, *DataReader*, *Publisher*,

Subscriber, and Topic. Working on these entities, our mapping of the DDS standard into HPC is illustrated in figure 32 and can be described as follows: the HPC master node is represented by the DDS Publisher/DataWriter entity, since its main responsibility in conventional HPC systems is reading data from input sources, sending the partial data to compute nodes (Subscribers/DataReaders in DDS), and then collecting the results back. Typically, HPC environments use one master node for their message passing communication, and therefore, we apply the same concept by having one Publisher/DataWriter in all of our DDS-HPC applications.

Similarly, compute nodes are represented as Subscribers/DataReaders and they act as the worker nodes. The association of a DataWriter with DataReader objects (or Master to compute nodes in HPC terms) is done by means of Topics, which act as the messaging interface “or channels” between all the entities, similar to the message passing interface (MPI) in conventional HPC systems. Samples in the Topics are transferred by utilizing the communication medium, which is represented by the high speed interconnect in the HPC systems.

As to control the QoS policies and add the Persistence Service in our DDS-HPC model, we dedicate an additional node to host the Persistence Service libraries. In our integration, we utilize the standard HPC management node for this task.

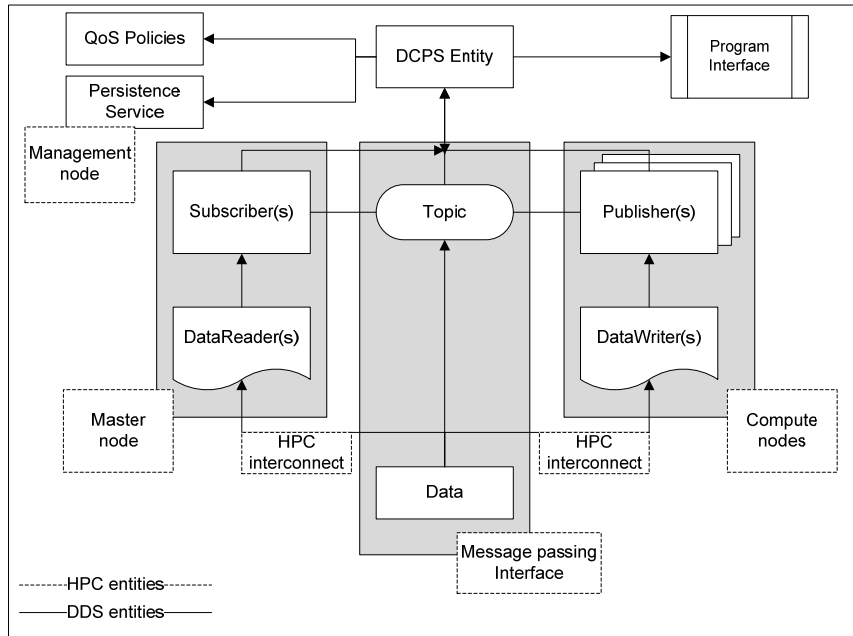


Figure 32: HPC-DDS integration model

Most of nodes' interactions in the conventional DDS implementations are one-way communication, that is, from the publishers to subscribers. These publishers and subscribers have to reverse their roles in order to establish two-way communication in the HPC environment. To mimic the two-way interaction between the master and compute nodes and have it similar to the MPI-based systems, we spawn two threads in the Publisher/DataWriter (i.e. master node), where thread 0 acts as the publisher for sending data, while thread 1 acts as a Subscribers/DataReaders for receiving the final data from the computes. Likewise, all compute nodes have the same structure for their two-way communication. Figure 33 shows our general flowchart for implementing parallel programs using the HPC architecture by following the DDS model, while figure 34 shows our generic pseudo-code for porting parallel programs to this architecture.

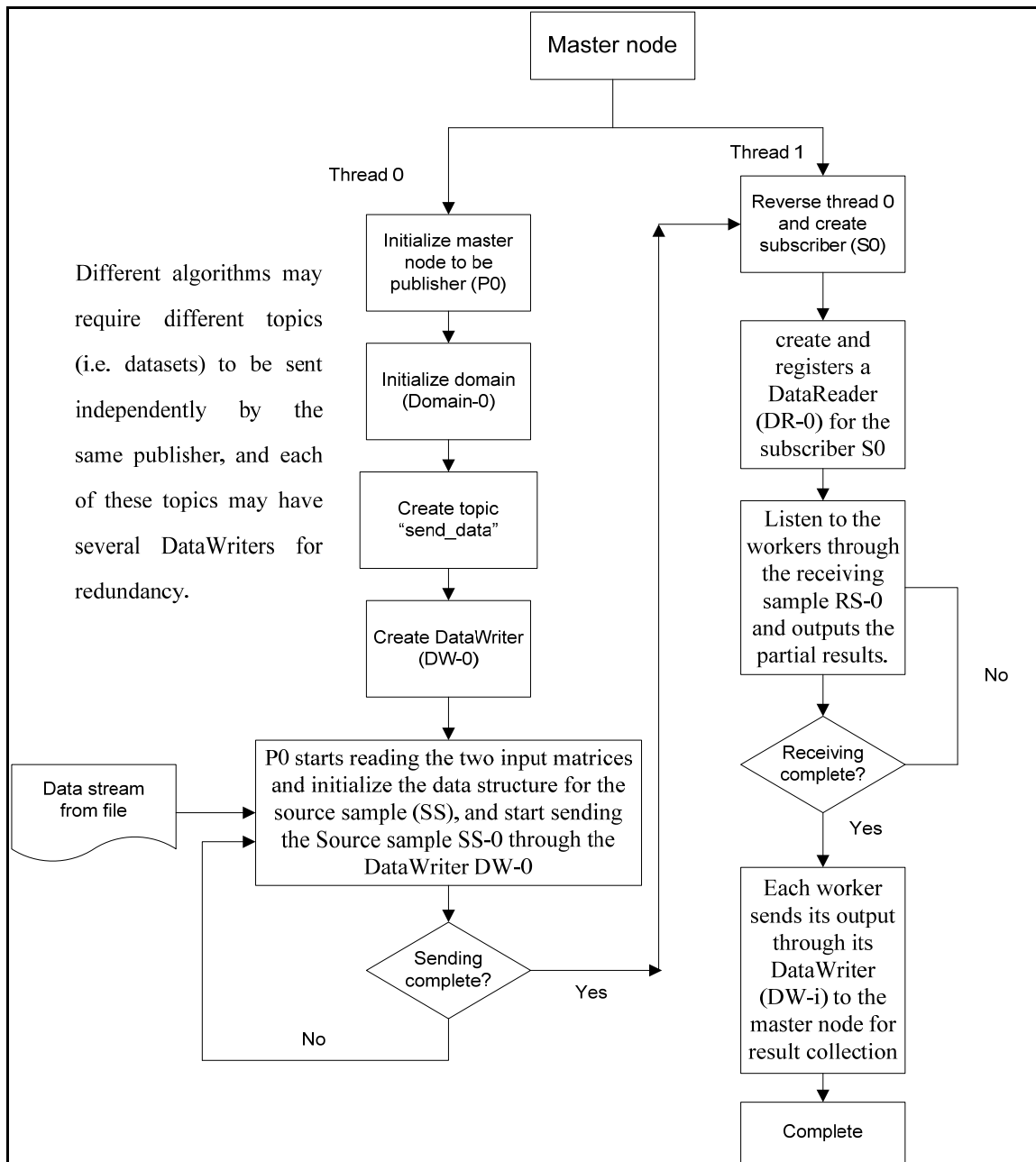


Figure 33: The general HPC-DDS flowchart for implementing parallel programs

```

Master node:
  Thread 0:
    Create an instance of publisher P0 with selected QoS profile in domain: Domain-0
    Create a DDS topic [name: send_data]
    Create and register a DataWriterDW-0 for the publisher P0 that uses the created
  topic
    Create an instance of the topic (data sample)
    Read the input data [from input]
    Initialize the data structure Source Sample SS (data parameters, no. of workers).
    Publish the Source sample SS-0 through DataWritersDW-0

  Thread 1:
    Create an instance of subscriber S0 with selected QoS profile in domain: Domain-0
    Create and register a DataReaderDR-0 for the subscriber S0 that uses topic [name: Recv_result]
    While (Result sample RS-0 not complete)
      If new data from sample RS received
        Get worker number i
        Output the processed result
      End if
    End while

Worker nodes (Wi):
  Create an instance of subscriber Si with selected QoS profile in domain: Domain-0
  Create and register a DataReader DR-i for the subscriber Si that uses topic [name: send_data]
  While (!timeout && data !received)
    If sample SS-0 received
      Get worker number
      Do partial computation in Wi
    End if
  End while

  Create an instance of publisher Pi with selected QoS profile in domain: Domain-0
  Create a DDS topic [name: Recv_result]
  Create and register a DataWriter DW-i for the publisher Pi that uses the created topic
  Create an instance of the topic (data sample)

  Initialize the data structure Result Sample RSi: (the result data, number of workers, i).
  Publish the Result sample RS-0 through DataWriters DW-i

```

Figure 34: The HPC programming pseudo-code using DDS paradigm

7.2.1 Implemented Quality of Service Policies

In order to enable the DDS reliability QoS on our DDS-HPC design, we adopted three main QoS policies in our implementation; these are: durability, reliability and history.

During the execution of our DDS-HPC implementation, the independent “persistence service” is run on a separate physical server (i.e. the management node) in order to support the “durability” QoS policy. This persistence service saves the published data samples so that they can be delivered to subscribing recipients that join the system at a later time, even if the publishing application has already terminated. The persistence service can use a file system or a relational database to save the status of the system. In case of a failure in the persistence service, the system administrator may initialize a new instance of the service (or reboot the down service if possible) to resume the functionality of the system without losing the current status. The newly initialized persistence service would read the written checkpointed status as defined in the policy file.

The second QoS policy, which is reliability, indicates the level of reliability requested by a DataReader or offered by a DataWriter. Data senders may set different settings of reliability, indicated by the number of past issues they can keep in their storage (or memory) for the purpose of retrying transmissions. Subscribers may then demand differing levels of reliable delivery, ranging from fast-but-unreliable "best effort" to highly reliable in-order delivery. This provides per-data stream reliability control. In case the reliability type is set to “RELIABLE”, the write operation on the DataWriter might be blocked if there is a possibility that the data can be lost, or if the resources limits specified in the RESOURCE_LIMITS QoS to be consumed. In these cases, the RELIABILITY

option “max_blocking_time” configures the maximum duration the write operation may block.

Further, if the reliability type is set to “RELIABLE”, data-samples generated from a single DataWriter cannot be made available to the receiving nodes if there are older data-samples that have not been delivered yet due to a communication issue. In other words, the DDS middleware will attempt to find other paths and retransmit data-samples in order to reconstruct a correct snapshot of the DataWriter history before it is accessible by the recipients.

On the other hand, if the reliability type is set to “BEST_EFFORT”, the service will not resend the missing data-samples, but will ensure that data sent will be stored in the DataReader(s) history, in the same order they were created by the DataWriter. Therefore, the recipient node may lose some data-samples but it will never see the value of a data-object change from a newer value to an older value.

The third policy, history, controls the reaction of the middleware when the value of an instance changes before it is finally communicated to some of its existing DataReader entities. If the type is set to “KEEP_LAST”, then the middleware will only attempt to keep track the latest values of the data and discard the older ones. In this case, the value controls the maximum number of values the middleware will maintain and deliver. The default (and most frequently used setting) for this QoS is one, indicating that only the most recent value should be delivered.

If the history type is set to “KEEP_ALL”, then the middleware will attempt to keep and deliver all the values of the sent data to existing recipients. Similar to the RELIABILITY QoS, the resources that the middleware can use to retain the different values are limited by the settings of the RESOURCE_LIMITS QoS. If the limit is reached, then the reaction

of the middleware will depend on the RELIABILITY QoS. That is, if the reliability is set to “BEST_EFFORT”, then the old values will be dropped, while if reliability is set to “RELIABLE”, then the middleware will block the sender until it can send the ongoing old values first to all recipients.

```

<datawriter_qos>
  <reliability>
    <kind>RELIABLE_RELIABILITY_QOS</kind>
  <max_blocking_time>
    <sec>60</sec>
  </max_blocking_time>
</reliability>
<history>
  <kind>KEEP_ALL_HISTORY_QOS</kind>
</history>
<durability>
  <kind>PERSISTENT_DURABILITY_QOS</kind>
</durability>
<protocol>
  <rtps_reliable_writer>
    <min_send_window_size>50</min_send_window_size>
    <max_send_window_size>50</max_send_window_size>
  </rtps_reliable_writer>
</protocol>
</datawriter_qos>

<datareader_qos>
  <reliability>
    <kind>RELIABLE_RELIABILITY_QOS</kind>
  </reliability>
<history>
  <kind>KEEP_ALL_HISTORY_QOS</kind>
</history>
<durability>
  <kind>PERSISTENT_DURABILITY_QOS</kind>
</durability>
</datareader_qos>

```

Figure 35: The QoS policy file for our DDS-HPC design.

Figure 35 shows the implemented QoS with their values. Other DDS QoS policies were set to their default values, since they are either not applicable to the HPC implementation, or not suitable for the type of applications (i.e. single master/publisher, multiple computes/subscribers) that we have tested in our experiments. Appendix 1 shows these QoS policies and their default values.

7.3 Experimental Setup and Methodology

In order to evaluate the feasibility and impact of adopting the DDS QoS into HPC, we implemented three applications with different computational models. The first program “the parallel matrix multiplication” presents the hybrid type of parallel applications where it involves both intensive communication between nodes as well as relatively high computational power, and that the amount of communication between the master and compute nodes is proportional to the size of the input matrices. On the other hand, the second program “the prime numbers search”, which is a modified version of Blaise’s MPI Prime [91], represents the computation-bound type of applications, since the collective calls in the program are used to reduce two data elements only; these are: the number of primes found and the largest prime in the sequence. The third application “Node-to-Node Streaming” represents the communication-bound type of applications, in which it is used for streaming large amount of data between compute nodes. The complete pseudo-codes for the three applications can be found in appendices 2, 3 and 4 of this dissertation work.

To perform benchmark evaluation, we used the same cluster setup that was described in section 5.1. In all of our three implementations, we attempted to make our programming structure as close as possible to the typical MPI model, where we have single master/several computes hierarchy. This approach was followed in order to have a fair comparison between the two programming models in terms of runtime and complexity. Further and similar to MPI, a copy of the DDS libraries were placed in every master and compute nodes of the cluster in order for it to work in our design.

7.3.1 The Matrix Multiplication Application

In order to evaluate the performance of the DDS over HPC and compare it with MPI, we implemented the parallel matrix multiplication algorithm using both paradigms (i.e. DDS and MPI) and evaluated them on the mentioned HPC cluster. Beside it is computationally intensive with $O(n^3)$ iterations, we represented the hybrid type of applications by the matrix multiplication algorithm since it is a fundamental operation in many numerical linear algebra applications. Its efficient implementation on parallel computers is an issue of prime importance when providing such systems with scientific software libraries [70].

The implementation starts by designating the master node of the cluster as the main publisher. This node, in turn, spawns two threads using OpenMP to parallelize its two main functions: the first thread is responsible for initializing the node to be a publisher (P0) with selected QoS profile, which is predefined in an XML file as shown in figure 35. The thread also specifies the domain where all the publishers and subscribers would work on, which is domain-0 in our implementation. Specifying the domain is necessary in order to allow multiple groups of publishers and subscribers to work independently, segmenting the cluster into several smaller sub-clusters, if needed.

Next, a topic with the name 'send_matrix_data' is created and a DataWriter (DW-0) is initialized under P0 using the created topic. The reason for this hierarchy is that there exist algorithms (i.e. other than the matrix multiplication application) that would require different topics (i.e. datasets) to be sent independently by the same publisher, and each of these topics may have several DataWriters for redundancy.

After that, the publisher reads the two matrices from input and initializes the data structure for the source sample (SS) by defining the matrices dimension and the number

of workers. The source sample then starts sending the Source sample SS-0 through the DataWriter DW-0.

The second thread on the master node reverses the function of thread 0 by creating an instance of a subscriber S0 with selected QoS profile in Domain-0, in preparation to receive the partial results from the workers (workers act as subscribers at the beginning and then publishers at the end). Specifically, it creates and registers a DataReader (DR-0) for the subscriber S0 (the workers) that uses topic 'recv_matrix_result'. It then listens to the workers through the receiving sample RS-0 and outputs the partial results.

On the subscribers' side (i.e. the workers), each node initiates itself as a subscriber to the main publisher P0, assigns an ID to itself (Wi), and starts receiving the rows and columns for computation. The distribution of which rows go to which node is done dynamically in a way that is determined by first identifying the row-wise range taken by each node using the formula:

$$\text{row_range_max} = [(\text{no. of total rows} / \text{no. of workers}) * i] - 1.$$

And then identifying the minimum rows range using the formula:

$$\text{row_range_min} = \text{row_range_max} - [(\text{no. of total rows} / \text{no. of workers}) - 1]$$

Subsequently, the computation starts by the three nested-for loops, similar to the matrix multiplication implementation via MPI, using the formula:

$$\text{result_matrix}[x][y] += \text{matrix1}[x][z] * \text{matrix2}[z][y]$$

Each worker then sends its output through its DataWriter (DW-i) to the master node for result collection.

In case of a node failure on the workers' side, the system administrator may initiate a new node with the same ID of the failed worker. The new worker would read the written checkpointed status as defined in the policy, re-read the sample from the persistence service, and resume the operation of the system.

It is important to mention that as a requirement for the durability QoS, all sent topics require DataWriters to match the configuration of the persistent QoS policy configuration with the DataReaders. As a consequence, a DataWriter that has an incompatible QoS with respect to what the topic specified will not send its data to the persistent service, and thus its status will not be saved. Similarly, a DataReader that has an incompatible QoS with respect to the specified in the topic will not get data from it.

7.3.1.1 Performance Evaluation and Results

In this section, we present our experimental results using the HPC system illustrated in section 5.1. All measurements reported in this section are the average readings of three runs. The performance is measured and compared for both MPI and DDS implementations while varying the cluster size. In our tests, we measured the performance in terms of scalability, total runtime, execution version initialization times, communication overhead and fault recovery delay (for DDS-based runs).

Figure36 shows our benchmarks to test the scalability and runtime of MPI and DDS while varying the number of nodes. During our first trials with DDS, the `DDS_ASYNCHRONOUS_PUBLISH_MODE` QoS was used when sending and receiving the matrices elements between the nodes. This QoS option, however, did not scale well when extending the matrices size beyond 60 elements. Applying the `DDS_SYNCHRONOUS_PUBLISH_MODE` QoS, on the other hand, enabled us to

enlarge the matrices up to 500 elements. The latter QoS policy reduces the amount of time the application thread spends sending data. Typically, it is used along with the `DDS_PublishModeQosPolicy` and `DDS Flow Controller` policies to send large data reliably, specify how DDS sends application data on the network, and instruct the middleware to use its own thread to send data, instead of the user thread. In that case, each `DDS Publisher` spawns a single asynchronous publishing thread (`DDS_AynchronousPublisherQosPolicy::thread`) to serve all its asynchronous `DDS DataWriter` instances.

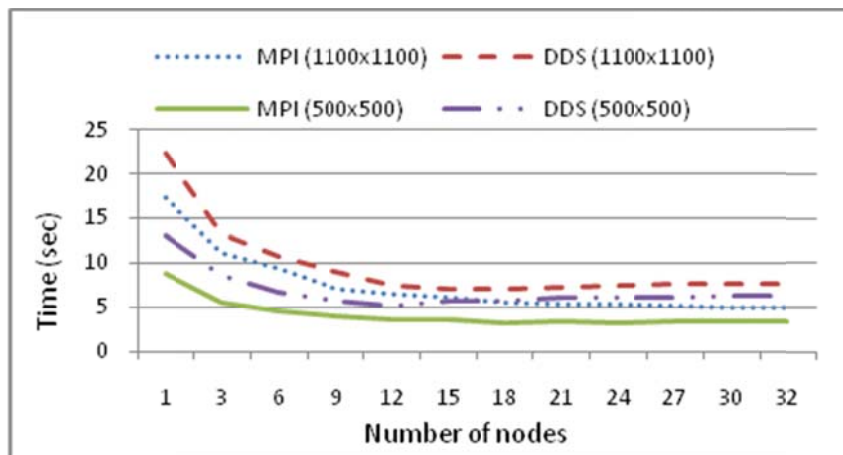


Figure 36: MPI vs. DDS Mat. Mult. runtime when varying the number of nodes

In DDS terms, "large data" means that the data that cannot be sent as a single packet by a network transport. According to RTI manual [77], the application must be configured to fragment the data and send it asynchronously when sending data larger than 63K over UDP/IP.

Another tweak that was done to extend the size of the computed matrices was to enlarge the "stack reserve size" to 32MB (default is 1MB) when compiling the DDS-based code.

The reserve value specifies the total stack allocation in virtual memory for the running program. This option enabled the code to compute up to 1100x1100 size matrices. The application, however, did not perform well when enlarging the matrices beyond 1100 elements.

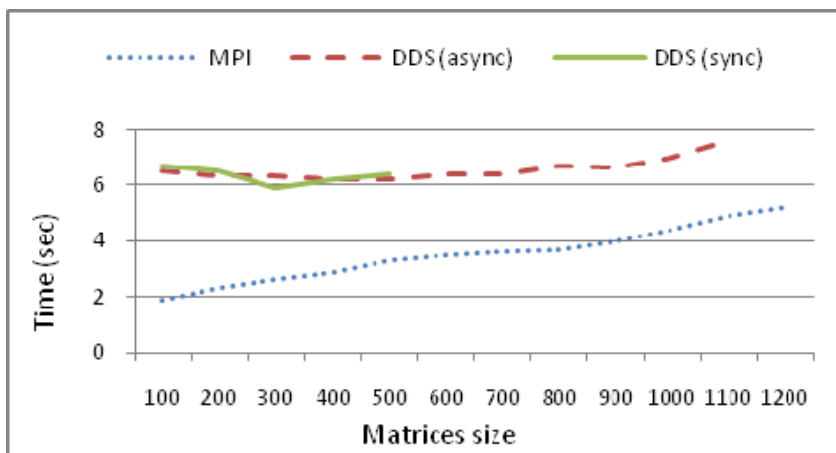


Figure 37: MPI vs. DDS Mat. Mult. runtime when varying the matrices size on 32 nodes

Figure 37 also demonstrates the runtime of both the DDS and MPI implementations while varying the matrices size. Clearly, the MPI version outperformed the DDS version by taking around 3.3 seconds to compute 500x500 size matrices, compared with 6.2 seconds using DDS. This performance difference was more observable when extending the matrices size up to 1100.

Another observation when looking at the MPI implementation is the slight increase in the run time when multiplying the 500x500 size matrices on 32 nodes. This increase is related to the additional communication overhead with respect to the computation time. This communication is lessened in the 1100x1100 multiplication as the computation time gets larger with respect to the communication overhead.

To magnify the effect of MPI communication overhead with respect to computation time, we extended the MPI matrix multiplication benchmark runs to 126 nodes. Figure 38 shows the effect of this communication overhead as the number of nodes increases. Obviously, finding the right combination of nodes to gain the best performance depends on the nature of the application and can only be found by performing empirical runs. In this range on nodes, DDS shows lesser overhead bounce than MPI.

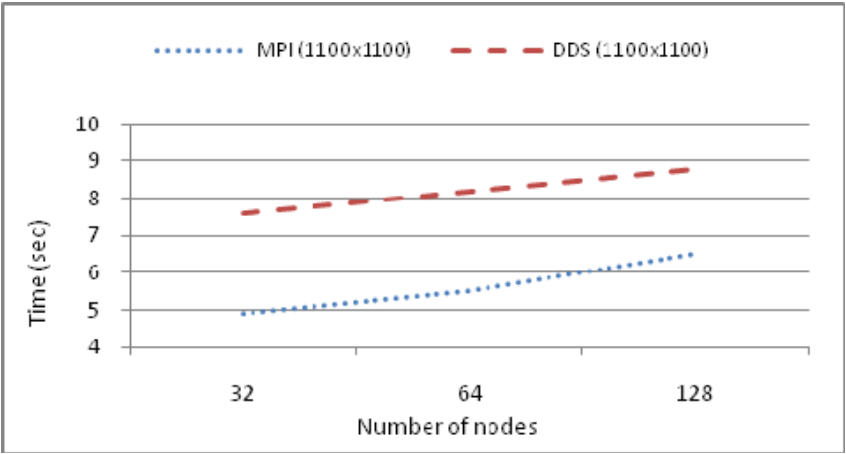


Figure 38: The communication overhead for computing 1100x1100 matrices

As shown in table 11, most of the communication overhead when using the DDS-based code was spent in the preparation phase (ratio: 2.86 for preparation-to-computation times) in the 500x500 test. This overhead is especially evident when the matrices size is small, as it took 5.8 seconds to initialize the publishers, the DataWriters, the instances, and then send 100x100 matrices to the compute nodes. Again, the ratio of the preparation-to-computation time was lessened when increasing the matrices size up to 1100x1100 elements.

	DDS (100x100)	DDS (500x500)	DDS (1100x1100)
Preparation time + Communication	5.8 sec.	4.3 sec.	4.8 sec.
Computation time	0.9 sec.	1.5 sec.	2.8 sec.
Total	6.7 sec.	6.2 sec.	7.6 sec.
Ratio	6.45	2.86	1.71

Table 11: Communication overhead ratio in DDS while running on 32 nodes

Figure 39 shows the delay in engaging a new node in the DDS domain, replacing a crashed node, while using the persistent service and durability, reliability and history QoS. This test is not applicable to the MPI implementation. As indicated in the figure, the delay is proportional to the size of the matrices as the persistent service needs to re-send all the previously published instances to this new node. At the 100x100 matrices benchmark, the overhead to re-send matrices data was about 0.8 seconds, while this overhead has increased to about 2 seconds in the 1100x1000 test.

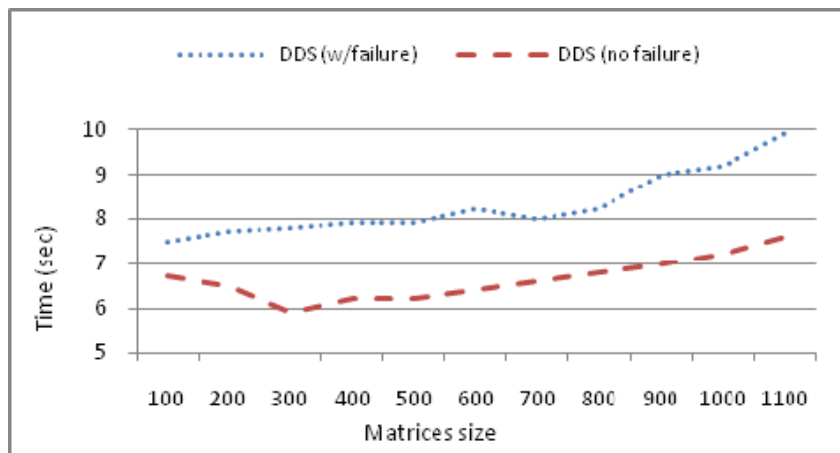


Figure 39: Network delay when engaging a new node in DDS while running Mat. Mult.

Figure 40 illustrates the network utilization when computing the matrices using DDS and MPI implementations. As illustrated in the figure, the total Megabytes sent and received for MPI implementation is around 11.5MB when computing the 1100x1100 matrices, while it is around 15.7MB for the DDS implementation. It is also noticeable that the DDS implementation has lower negative scalability-overhead when the matrix size increases, as opposed to MPI implementation.

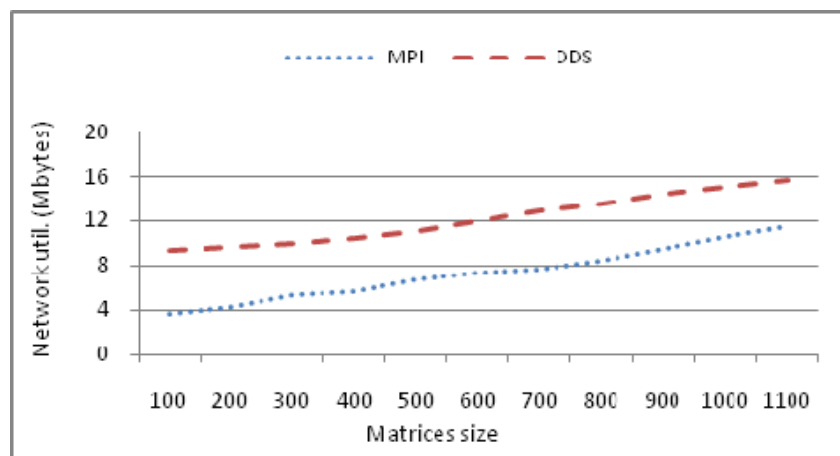


Figure 40: The network utilization for running Matrix Multiplication on 32 nodes

7.3.2 The Primes Search Application

The goal of this parallel application is to pass a large interval of integers, divide it evenly among the compute nodes, and search for prime numbers in each sub-interval while finding the largest prime. We implemented the primes search algorithm using both paradigms (i.e. DDS and MPI) and evaluated them on the mentioned HPC cluster. Unlike the parallel matrix-multiplication experiment, this application is purely computation-bound and requires minimal inter-nodes communication, since the collective communications calls on the nodes are used to collect the only two data elements

requiring communications: the number of primes found and the largest prime, regardless of the interval size.

Following our model of adopting DDS into HPC, the application starts by designating the master node as the main publisher, and spawning two threads using OpenMP to parallelize its two main functions: the first thread is responsible for initializing the node to be a publisher (P0) with the given QoS profile, while the second thread is set for receiving the results from the compute nodes.

A topic with the name 'send_interval_data' is created and a DataWriter (DW-0) is initialized under P0 using the created topic. The master node (P0) reads the interval start and end points for searching the primes, and initializes the data structure for the source sample (SS) by defining the number of workers, and the size of the interval to be searched. The source sample then starts sending the Source sample SS-0 through the DataWriter DW-0.

The second thread on the master node reverses the function of thread 0 by creating an instance of a subscriber S0 with selected QoS profile in Domain-0, in preparation to receive the partial results from the workers. It uses topic 'recv_primes_result' and listens to the workers through the receiving sample RS-0 and outputs the partial results.

On the subscribers' side, each node initiates itself as a subscriber to the main publisher P0, assigns an ID to itself (Wi), and starts receiving the sub-intervals for searching the primes. The distribution of which sub-interval goes to which compute node to search for primes is determined by first identifying the start of the sub-interval for each compute node using the formula:

$$\text{my_interval_start} = (\text{myID} * 2) + 1$$

Where myID is the node ID in the cluster nodes' sequence.

Then, the subsequent elements for each sub-interval are calculated by adding apace starting from my_interval_start, where the pace is equal to the number of compute nodes:

```
pace= LastNode_ID
for (n=my_interval_start; n<=last_element; n=n+pace)
PrimeTest(n)
```

The Workers test the odd numbers in their intervals and up to the last element in the sub-interval, using the function:

```
PrimeTest {
SqrRoot = (int) sqrt(n);
    for (i=3; i<=SqrRoot; i=i+2)
        if ((n%i)==0)
            print "prime n found";
        else print "n is composite";
    }
```

Running through each sub-interval, each worker sends its output through its DataWriter (DW-i) to the master node for results collection.

Similar to the matrix multiplication example, in case of a node failure on the workers' side, the system administrator may initiate a new node with the same ID of the failed worker. The new worker would read the written checkpointed status as defined in the QoS policy, re-read the sample from the persistence service, and resume the operation of the system.

7.3.2.1 Performance Evaluation and Results

In this section, we present our experimental results using the motioned HPC system. In our benchmarks, we measured the performance in terms of scalability, total runtime, execution version initialization times, and fault recovery delay (for DDS-based runs). We

skipped the communication overhead and the network utilization benchmarks since inter-nodes communication is insignificant in this application. Similar to our previous experiments, all measurements reported in this section are the average readings of three runs.

Table 12 shows our benchmark to assess the scalability and runtime of MPI and DDS, while varying the number of nodes and fixing the search interval to 500 million integers. Clearly, the two MPI and DDS implementations have comparable results when scaling the Primes Search application up to 32 nodes. The slight DDS performance degradation is due to the QoS parameters that are communicated at the runtime.

	Number of nodes				
	2	4	8	16	32
MPI	517.2 sec.	258.1 sec.	130.5 sec.	66.3 sec.	33.9 sec.
DDS	517.5 sec.	258.7 sec.	131.1 sec.	67.0 sec.	34.7 sec.

Table 12: MPI vs. DDS Primes Search runtime while varying the number of nodes

Table 13 presents our benchmark while varying the size of the interval to be searched while fixing the number of compute nodes to 32. The benchmark shows that both MPI and DDS implementations scaled almost linearly when processing up to 500 million elements input. Another observation is the sustained performance when the number of nodes was fixed at 32 while the size of the interval was reduced to 10 million elements. The reason for this linear performance is the minimal interaction between the nodes (i.e.

communication overhead), regardless of the number of computes and the size of the interval to be searched.

	Number of elements to be searched (<i>in millions</i>)						
	10	50	100	200	300	400	500
MPI	0.61 sec.	3.15 sec.	6.59 sec.	13.2 sec.	19.46 sec.	26.7 sec.	33.9 sec.
DDS	0.62 sec.	3.21 sec.	6.73 sec.	13.5 sec.	19.83 sec.	27.28 sec.	34.7 sec.

Table 13: MPI vs. DDS Primes Search runtime while varying the input size on 32 nodes

Table 14 presents the added delay in engaging a new node in the DDS domain, replacing a crashed node while the application is running. This test is not applicable to the MPI implementation, due to the lack of the fault-tolerance feature. As indicated in the table, the delay is almost constant since the sent data from the publisher to the newly engaged node has a fixed size on all tests (the sub-interval, and the number of workers).

	Number of elements to be searched (<i>in millions</i>)				
	100	200	300	400	500
DDS (no failure)	6.73 sec.	13.5 sec.	19.83 sec.	27.28 sec.	34.7 sec.
DDS (w/failure)	7.21 sec.	14.1 sec.	20.20 sec.	27.53 sec.	35.1 sec.

Table 14: The delay when engaging a new node in DDS while running Primes Search

7.3.3 The Node-to-Node Streaming Application

The goal of this application is to send large random data from one node in the cluster to another and send it back, for the purpose of simulating point-to-point communication using the high speed Infiniband interconnect. This application represents the native communication-bound type of applications and it heavily relies on the HPC interconnect throughput.

Using the same described DDS-HPC model, the application begins by designating the master node as the main publisher, and spawning two threads using OpenMP to parallelize its two main functions: the first thread is responsible for initializing the node to be a publisher (P0) with selected QoS profile, while the second thread is set for receiving the results from the compute node.

The first thread creates a topic with the name 'send_stream_data' and initializes a DataWriter (DW-0). Then, the input file is read and the data structure is initialized for the source sample (SS) while specifying the number of workers (one node in this application). The second thread on the master node reverses the function of thread 0 by creating an instance of a subscriber S0 with selected QoS profile in Domain-0, in preparation to receive the file back from the worker. It uses topic 'recv_stream_result' and listens to the worker through the receiving sample RS-0 and outputs the results.

On the receiver side, the node initiates itself as a subscriber to the main publisher P0, assigns an ID to itself (W0), in preparation to start receiving the data stream.

The routine for creating large data is done by the following functions:

```

const long int K = 1048576;
const long int msgsize = 32*K;

// Initialize X and Y
for (i=0; i<msgsize; i++) {
    X[i] = 1;
    Y[i] = 2;
}

```

While sending the receiving data is done through the following MPI function:

```

MPI_Comm_rank(MPI_COMM_WORLD, &ID);

if (ID == 0) {
MPI_Send(X, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD);
MPI_Recv (Y, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);

} else { /* ID == 1 */

MPI_Recv (Y, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
MPI_Send (Y, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD);

}

```

The complete pseudo-code for the HPC-DDS version can be found in Appendix 4.

7.3.3.1 Performance Evaluation and Results

This section presents our experimental results for testing the Node-to-Node streaming application. In the experiments, we tested both DDS and MPI implementations by streaming 5GB and 10GB of data, while all reported measurements are the average readings of three runs.

Figure 41 illustrates our benchmark to evaluate the scalability and runtime of MPI and DDS, using 10GB and 50GB of streamed data. Clearly, the MPI superseded the DDS implementation with the use of the low-level MPI_Send and MPI_Recv functions, where it was capable of achieving maximum one-way throughput of 1,519MB/s with the 50GB test, compared to a maximum throughput of 1,323MB/s for the DDS implementation.

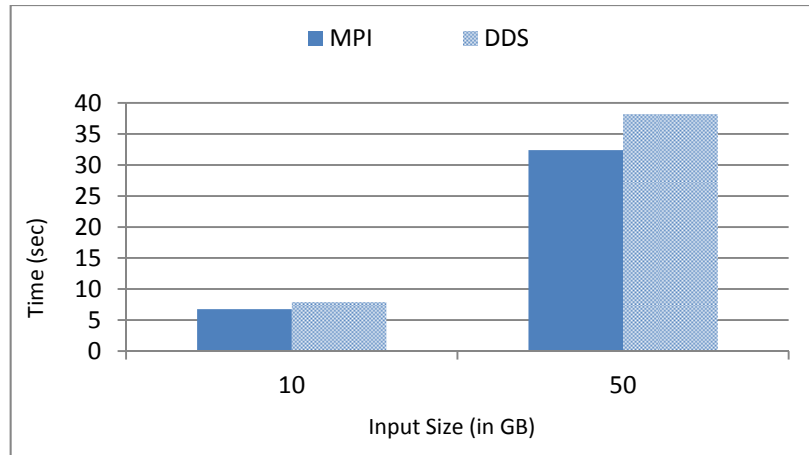


Figure 41: Node-to-Node Throughput

Similar to the matrix multiplication application, the QoS policy `DDS_SYNCHRONOUS_PUBLISH_MODE` had to be set to enable sending large data and instruct the middleware to use its own thread to send data, instead of the user thread. The synchronous communication, on the other hand, adds additional overhead as indicated in the test.

Figure 42 shows the elapsed time for engaging a new node in the DDS domain, replacing a crashed node, and resending the data again. By setting the `DURABILITY` QoS to `TRANSIENT`, the DataWriter stores all the sent samples in memory and resends them to the new node once it joins the domain. This setting was only applicable to the 10GB input size, since the 50GB input can only be stored in the DataWriter's permanent storage (by setting the `DURABILITY` QoS to `PERSISTENT`). Using the `PERSISTENT` setting, however, resulted in unrealistic readings due to the excessive storage access overhead. This test is also not applicable to the MPI implementation, due to the lack of the fault-tolerance feature.

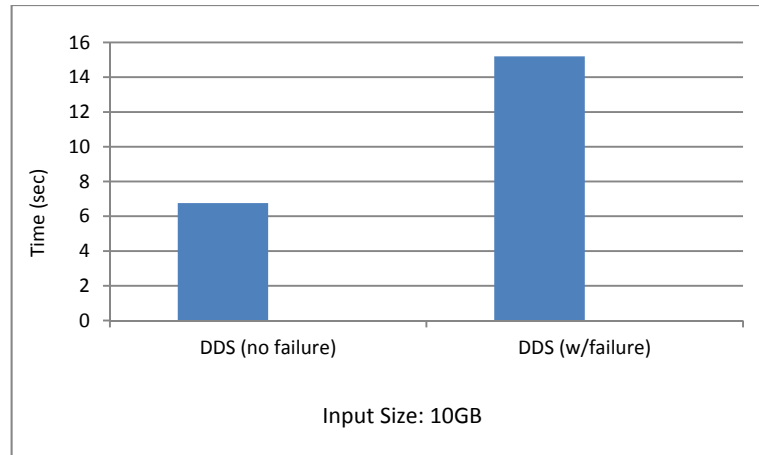


Figure 42: Failing the receiver in DDS while running the Node-to-Node application

7.4 Conclusion

In this chapter, we presented our work of adopting DDS standard into HPC in order to circumvent the MPI shortcomings and provide QoS for HPC applications. As demonstrated in our tests, DDS integration into HPC adds considerable overhead in terms of performance and network utilization when the application is mainly communication-bound, while the performance is comparable to those MPI-based applications when the program is computation-bound. In both cases, the solution is a viable option for those applications in which QoS is considered a priority, or for those HPC batch jobs that would run on commodity hardware, where the probability of failure is not negligible.

Chapter 8

Conclusion and Future Work

8.1 Overview

The ever increasing demand for computing power in scientific applications has accelerated the process of deploying HPC systems that deliver Peta-scale performance. Current HPC systems that are capable of running large-scale parallel applications may span multi-thousands of nodes. For parallel applications, the failure probability increases significantly with the increase in number of nodes. Thus, ignoring failures or system reliability can have severe effect on the performance of the HPC cluster, and quality of service.

In this research work, we investigated the reliability and QoS controls in the high performance computing environments through examining the Diskless HPC clusters and the Infiniband routing and QoS techniques as specified by the Infiniband Architecture Specifications (IBA), using a Westmere-based HPC cluster. Then, we presented our work

of adopting DDS reliability QoS into HPC. This integration provided the ability to control QoS properties on HPC and Grids that affect performance, reliability, and fault-tolerance. To the best of our knowledge, this is the first research focusing on the integration of DDS QoS policies into HPC computing.

8.2 Conclusion

Our results obtained show that while controlling QoS and reliability in HPC can be a challenging task, we were able to achieve comparable performance when QoS and other reliability-related techniques were enabled in specific HPC layers, such as the Infiniband interconnection and the HPC cluster storage disks.

Looking at the diskless HPC, the setup proved to be a compelling alternative with greater benefits when compared to diskfull clusters. In particular, the tests showed that diskless clusters yield similar GFLOPS performance to diskfull clusters, and in some cases outperform the diskfull. In terms of power consumption, diskless clusters clearly win with power saving of at least 3 Watts per node. That said, the biggest shortcoming of this setup is the need for large memory nodes to host the OS, as it was demonstrated that if compute nodes are forced to perform disk swapping by decreasing their available memory, the compute nodes will freeze. Another obvious shortcoming is that the disk node in a diskless cluster can be a single point of failure. However, these two shortcomings can be alleviated by increasing the RAM of compute nodes and by having more reliable disk nodes that use advanced network storage technologies such as NAS and RAID technology.

In terms of HPC-DDS integration, we adopted three main QoS policies in our implementation that affect reliability; these are: DURABILITY, RELIABILITY and HISTORY, while keeping all other non-used policies at their default values. We also used the DDS Persistence Service to maintain a backup of the communication data and have a centralized control of the QoS policies. Our results showed that our model adds up to 20% overhead in terms of performance and network utilization when the application is mainly communication-bound, due to the additional QoS parameters send during nodes communication, while the performance is comparable to those MPI-based applications when the program is computation-bound, as it was presented in the Primality Test application. In both cases, the solution is considered a viable option for those parallel applications that would last for several days or even weeks to run or for those jobs where QoS is considered a priority. Furthermore, the solution is feasible for those HPC batch jobs that would run on commodity hardware, where the probability of failure is not negligible.

8.3 Future Work

Research studies are continuing to discuss QoS on the different layers of the distributed and high performance computing systems.

In terms of diskless HPC, we plan to expand the size of the Infiniband diskless cluster to include 512 compute nodes, and then investigate its performance. We also plan to evaluate diskless cluster performance when using other popular benchmarks such as the Pallas MPI benchmarking tool which gives more insight on MPI behavior and

performance. We are also considering measuring the performance of diskless clusters when using 10Gbps Ethernet for IPC communication instead of Infiniband.

In terms of DDS integration into HPC, we plan to import other advance MPI-based applications and examine the effect of QoS on their reliability and performance. We believe that there are other compute-intensive applications that require large clusters and long run-times, where they can benefit from the addition of QoS properties, despite the added overhead to their performance.

Appendix 1: The DDS QoS (as defined in www.omg.org)

QoS Policy	Value	Meaning	Concerns
TOPIC_DATA	A sequence of octets: "value"	User data not known by the middleware, but distributed by means of built-in topics. The default value is an empty (zero sized) Sequence.	Topic
GROUP_DATA	A sequence of octets: "value"	User data not known by the middleware, but distributed by means of built-in topics. The default value is an empty (zero sized) sequence.	Publisher, Subscriber
DURABILITY	A "kind": VOLATILE, TRANSIENT_LOCAL, TRANSIENT, or PERSISTENT	This policy expresses if the data should 'outlive' their writing time.	Topic, DataReader, DataWriter
	VOLATILE	The Service does not need to keep any samples of data-instances on behalf of any <i>DataReader</i> that is not known by the <i>DataWriter</i> at the time the instance is written. In other words the Service will only attempt to provide the data to existing subscribers. This is the default kind.	
	TRANSIENT_LOCAL, TRANSIENT	The Service will attempt to keep some samples so that they can be delivered to any potential late joining <i>DataReader</i> . Which particular samples are kept depends on other QoS such as HISTORY and RESOURCE_LIMITS. For TRANSIENT_LOCAL, the service is only required to keep the data in the memory of the <i>DataWriter</i> that wrote the data and the data is not required to survive the <i>DataWriter</i> . For TRANSIENT, the service is only required to keep the data in memory and not in permanent storage; but the data is not tied to the lifecycle of the <i>DataWriter</i> and will, in general, survive it. Support for TRANSIENT kind is optional.	
	PERSISTENT	[optional] Data is kept on permanent storage, so that they can outlive a system session.	
DURABILITY_SERVICE	A duration "service_cleanup_delay" A HistoryQosPolicy Kind "history_kind" And three integers: history_depth, max_samples, max_instances, max_samples_per_instance	Specifies the configuration of the durability service. That is, the service that implements the DURABILITY kind of TRANSIENT and PERSISTENT	Topic, DataWriter

	service_cleanup_delay	Control when the service is able to remove all information regarding a data-instance. By default, zero	
	history_kind, history_depth	Controls the HISTORY QoS of the fictitious DataReader that stores the data within the durability service. The default settings are history_kind=KEEP_LAST history_depth=1	
	max_samples, max_instances, max_samples_per_instance	Control the RESOURCE_LIMITS QoS of the implied <i>DataReader</i> that stores the data within the durability service. By default they are all LENGTH_UNLIMITED.	
PRESENTATION	An “access_scope”: INSTANCE, TOPIC, GROUP And two booleans: “coherent_access” “ordered_access”	Specifies how the samples representing changes to data instances are presented to the subscribing application. This policy affects the application’s ability to specify and receive coherent changes and to see the relative order of changes. <i>access_scope</i> determines the largest scope spanning the entities for which the order and coherency of changes can be preserved. The two Booleans control whether coherent access and ordered access are supported within the scope <i>access_scope</i> .	Publisher, Subscriber
	INSTANCE	Scope spans only a single instance. Indicates that changes to one instance need not be coherent nor ordered with respect to changes to any other instance. In other words, order and coherent changes apply to each instance separately. This is the default <i>access_scope</i> .	
	TOPIC	Scope spans to all instances within the same <i>DataWriter</i> (or <i>Data Reader</i>), but not across instances in different <i>DataWriter</i> (or <i>Data Reader</i>).	
	GROUP	[optional] Scope spans to all instances belonging to <i>DataWriter</i> (or <i>DataReader</i>) entities within the same <i>Publisher</i> (or <i>Subscriber</i>).	
DEADLINE	A duration “period”	<i>DataReader</i> expects a new sample updating the value of each instance at least once every deadline <i>period</i> . <i>DataWriter</i> indicates that the application commits to write a new value (using the <i>DataWriter</i>) for each instance managed by the <i>DataWriter</i> at least once every deadline <i>period</i> . It is inconsistent for a <i>DataReader</i> to have a DEADLINE <i>period</i> less than its TIME_BASED_FILTER’s <i>minimum_separation</i> . The default value of the deadline <i>period</i> is infinite.	Topic, DataReader, DataWriter
LATENCY_BUDGET	A duration “duration”	Specifies the maximum acceptable delay from the time the data is written until the data is inserted in the receiver’s application-cache and the receiving application is notified of the fact. This policy is a hint to the Service, not something that must be monitored or enforced.	Topic, DataReader, DataWriter

		The Service is not required to track or alert the user of any violation. The default value of the <i>duration</i> is zero indicating that the delay should be minimized.	
OWNERSHIP	A “kind” SHARED EXCLUSIVE	[optional] Specifies whether it is allowed for multiple <i>DataWriters</i> to write the same instance of the data and if so, how these modifications should be arbitrated	Topic DataReader, DataWriter
	SHARED	Indicates shared ownership for each instance. Multiple writers are allowed to update the same instance and all the updates are made available to the readers. In other words there is no concept of an “owner” for the instances. This is the default behavior if the OWNERSHIP QoS policy is not specified or supported.	
	EXCLUSIVE	Indicates each instance can only be owned by one <i>DataWriter</i> , but the owner of an instance can change dynamically. The selection of the owner is controlled by the setting of the OWNERSHIP_STRENGTH QoS policy. The owner is always set to be the highest-strength <i>DataWriter</i> object among the ones currently “active” (as determined by the LIVELINESS QoS).	
OWNERSHIP_ STRENGTH	An integer “value”	[optional] Specifies the value of the “strength” used to arbitrate among multiple <i>DataWriter</i> objects that attempt to modify the same instance of a data-object (identified by <i>Topic + key</i>). This policy only applies if the OWNERSHIP QoS policy is of <i>kind</i> EXCLUSIVE. The default value of the <i>ownership_strength</i> is zero.	DataWriter
LIVELINESS	A “kind”: AUTOMATIC, MANUAL_BY_ PARTICIPANT, MANUAL_BY_TOPIC and a duration “lease_duration”	Determines the mechanism and parameters used by the application to determine whether an <i>Entity</i> is “active” (alive). The “liveliness” status of an Entity is used to maintain instance ownership in combination with the setting of the OWNERSHIP QoS policy. The application is also informed via listener when an <i>Entity</i> is no longer alive. The <i>DataReader</i> requests that liveliness of the writers is maintained by the requested means and loss of liveliness is detected with delay not to exceed the <i>lease_duration</i> . The <i>DataWriter</i> commits to signalling its liveliness using the stated means at intervals not to exceed the <i>lease_duration</i> . Listeners are used to notify the <i>DataReader</i> of loss of liveliness and <i>DataWriter</i> of violations to the liveliness contract. The default <i>kind</i> is AUTOMATIC and the default value of the <i>lease_duration</i> is infinite.	Topic, DataReader, DataWriter
	AUTOMATIC	The infrastructure will automatically signal liveliness for the <i>DataWriters</i> at least as often as required by the <i>lease_duration</i>	
	MANUAL modes	The user application takes responsibility to signal liveliness to the Service using one of the mechanisms described, “LIVELINESS,” on	

		page 113. Liveliness must be asserted at least once every <i>lease_duration</i> otherwise the Service will assume the corresponding <i>Entity</i> is no longer “active/alive.”	
	MANUAL_BY_PARTICIPANT	The Service will assume that as long as at least one <i>Entity</i> within the <i>DomainParticipant</i> has asserted its liveliness the other <i>Entities</i> in that same <i>DomainParticipant</i> are also alive.	
	MANUAL_BY_TOPIC	The Service will only assume liveliness of the <i>DataWriter</i> if the application has asserted liveliness of that <i>DataWriter</i> itself.	
TIME_BASED_FILTER	A duration "minimum_separation"	Filter that allows a <i>DataReader</i> to specify that it is interested only in (potentially) a subset of the values of the data. The filter states that the <i>DataReader</i> does not want to receive more than one value each <i>minimum_separation</i> , regardless of how fast the changes occur. It is inconsistent for a <i>DataReader</i> to have a <i>minimum_separation</i> longer than its <i>DEADLINE_period</i> . By default <i>minimum_separation</i> =0 indicating <i>DataReader</i> is potentially interested in all values.	DataReader
PARTITION	A list of strings “name”	Set of strings that introduces a logical partition among the topics visible by the <i>Publisher</i> and <i>Subscriber</i> . A <i>DataWriter</i> within a <i>Publisher</i> only communicates with a <i>DataReader</i> in a <i>Subscriber</i> if (in addition to matching the <i>Topic</i> and having compatible QoS) the <i>Publisher</i> and <i>Subscriber</i> have a common partition name string. The empty string (“”) is considered a valid partition that is matched with other partition names using the same rules of string matching and regular-expression matching used for any other partition name. The default value for the PARTITION QoS is a zero-length sequence. The zero-length sequence is treated as a special value equivalent to a sequence containing a single element consisting of the empty string.	Publisher, Subscriber
RELIABILITY	A “kind”: RELIABLE, BEST_EFFORT and a duration “max_blocking_time”	Indicates the level of reliability offered/requested by the Service.	Topic, DataReader, DataWriter
	RELIABLE	Specifies the Service will attempt to deliver all samples in its history. Missed samples may be retried. In steady-state (no modifications communicated via the <i>DataWriter</i>) the middleware guarantees that all samples in the <i>DataWriter</i> history will eventually be delivered to all the <i>DataReader</i> objects. Outside steady state the HISTORY and RESOURCE_LIMITS policies will determine how samples become part of the history and whether samples can be discarded from it. This is the default value for <i>DataWriters</i> .	

	BEST_EFFORT	Indicates that it is acceptable to not retry propagation of any samples. Presumably new values for the samples are generated often enough that it is not necessary to re-send or acknowledge any samples. This is the default value for <i>DataReaders</i> and <i>Topics</i> .	
	max_blocking_time	The value of the <i>max_blocking_time</i> indicates the maximum time the operation <i>DataWriter::write</i> is allowed to block if the <i>DataWriter</i> does not have space to store the value written. The default <i>max_blocking_time</i> =100ms.	
TRANSPORT_PRIORITY	An integer “value”	This policy is a hint to the infrastructure as to how to set the priority of the underlying transport used to send the data. The default value of the <i>transport_priority</i> is zero.	Topic, DataWriter
LIFESPAN	A duration “duration”	Specifies the maximum duration of validity of the data written by the <i>DataWriter</i> . The default value of the lifespan <i>duration</i> is infinite.	Topic, DataWriter
DESTINATION_ORDER	A “kind”: BY_RECEPTION_TIMESTAMP, BY_SOURCE_TIMESTAMP	Controls the criteria used to determine the logical order among changes made by <i>Publisher</i> entities to the same instance of data (i.e., matching <i>Topic</i> and <i>key</i>). The default kind is BY_RECEPTION_TIMESTAMP.	Topic, DataReader, DataWriter
	BY_RECEPTION_TIMESTAMP	Indicates that data is ordered based on the reception time at each <i>Subscriber</i> . Since each subscriber may receive the data at different times there is no guaranteed that the changes will be seen in the same order. Consequently, it is possible for each subscriber to end up with a different final value for the data.	
	BY_SOURCE_TIMESTAMP	Indicates that data is ordered based on a timestamp placed at the source (by the Service or by the application). In any case this guarantees a consistent final value for the data in all subscribers.	
HISTORY	A “kind”: KEEP_LAST, KEEP_ALL And an optional integer “depth”	Specifies the behavior of the Service in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing subscribers. This QoS policy controls whether the Service should deliver only the most recent value, attempt to deliver all intermediate values, or do something in between. On the publishing side this policy controls the samples that should be maintained by the <i>DataWriter</i> on behalf of existing <i>DataReader</i> entities. The behavior with regards to a <i>DataReader</i> entities discovered after a sample is written is controlled by the DURABILITY QoS policy. On the subscribing side it controls the samples that should be maintained until the application “takes” them from the Service.	Topic, DataReader, DataWriter
	KEEP_LAST and optional	On the publishing side, the Service will only	

	integer “depth”	attempt to keep the most recent “depth” samples of each instance of data (identified by its key) managed by the <i>DataWriter</i> . On the subscribing side, the <i>DataReader</i> will only attempt to keep the most recent “depth” samples received for each instance (identified by its <i>key</i>) until the application “takes” them via the <i>DataReader’s take</i> operation. KEEP_LAST is the default <i>kind</i> . The default value of <i>depth</i> is 1. If a value other than 1 is specified, it should be consistent with the settings of the RESOURCE_LIMITS QoS policy.	
	KEEP_ALL	On the publishing side, the Service will attempt to keep all samples (representing each value written) of each instance of data (identified by its <i>key</i>) managed by the <i>DataWriter</i> until they can be delivered to all subscribers. On the subscribing side, the Service will attempt to keep all samples of each instance of data (identified by its <i>key</i>) managed by the <i>DataReader</i> . These samples are kept until the application “takes” them from the Service via the <i>take</i> operation. The setting of <i>depth</i> has no effect. Its implied value is LENGTH_UNLIMITED.	
RESOURCE_LIMITS	Three integers: max_samples, max_instances, max_samples_per_instance	Specifies the resources that the Service can consume in order to meet the requested QoS.	Topic, DataReader, DataWriter
	max_samples	Specifies the maximum number of data-samples the <i>DataWriter</i> (or <i>DataReader</i>) can manage across all the instances associated with it. Represents the maximum samples the middleware can store for any one <i>DataWriter</i> (or <i>DataReader</i>). It is inconsistent for this value to be less than <i>max_samples_per_instance</i> . By default, LENGTH_UNLIMITED.	
	max_instances	Represents the maximum number of instances <i>DataWriter</i> (or <i>DataReader</i>) can manage. By default, LENGTH_UNLIMITED.	
	max_samples_per_instance	Represents the maximum number of samples of any one instance a <i>DataWriter</i> (or <i>DataReader</i>) can manage. It is inconsistent for this value to be greater than <i>max_samples</i> . By default, LENGTH_UNLIMITED.	
WRITER_DATA_LIFECYCLE	A boolean: “autodispose_unregistered_instances”	Specifies the behavior of the <i>DataWriter</i> with regards to the lifecycle of the data-instances it manages.	DataWriter

Appendix 2: The Matrix-Multiplication Pseudo-code Using DDS

```
Master node:
  Thread 0:
    Create an instance of publisher P0 with selected QoS profile in domain: Domain-0
    Create a DDS topic [name: send_matrix_data]
    Create and register a DataWriter DW-0 for the publisher P0 that uses the created topic
    Create an instance of the topic (data sample)
    Read the 2 matrices [dimensions, matrix elements, number of workers] from input
    Initialize the Source Sample SS: (the 2 matrices, the mxm dimension, no. of workers).
    Publish the Source sample SS-0 through DataWriters DW-0

  Thread 1:
    Create an instance of subscriber S0 with selected QoS profile in domain: Domain-0
    Create and register a DataReader DR-0 for the subscriber S0 that uses topic [name:
    recv_matrix_result]
    While (Result sample RS-0 not complete)
      If new data from sample RS received
        Get worker number i
        Output the row,column result in its matrix cell
      End if
    End while

Worker nodes (Wi):
  Create an instance of subscriber Si with selected QoS profile in domain: Domain-0
  Create and register a DataReader DR-i for the subscriber Si that uses topic [name: send_matrix_data]
  While (!timeout && data !received)
    If sample SS-0 received
      Get worker number
      row_range_max (Wi) = [(no. of total rows / no. of workers) * i] - 1
      row_range_min (Wi) = row_range_max - [(no. of total rows / no. of workers) - 1]
      for x from row_range_max (Wi) to row_range_min (Wi)
        for y from 0 to last_column (matrix#1)
          for z from 0 to last_row (matrix#2)
            result_matrix [x][y] += matrix1[x][z] * matrix2[z][y]
          end for
        end for
      end for
    End if
  End while

  Create an instance of publisher Pi with selected QoS profile in domain: Domain-0
  Create a DDS topic [name: recv_matrix_result]
  Create and register a DataWriter DW-i for the publisher Pi that uses the created topic
  Create an instance of the topic (data sample)

  Initialize the data structure Result Sample RSi: (the result matrix, number of workers, i).
  Publish the Result sample RS-0 through DataWriters DW-i
```

Appendix 3: The Primes Search Pseudo-code Using DDS

```
Master node:
  Thread 0:
    Create an instance of publisher P0 with selected QoS profile in domain: Domain-0
    Create a DDS topic [name: send_interval_data]
    Create and register a DataWriterDW-0 for the publisher P0 that uses the created topic
    Create an instance of the topic (data sample)
    Read the complete interval from file
    Initialize the data structure Source Sample SS: (the interval size, no. of workers).
    Publish the Source sample SS-0 through DataWritersDW-0

  Thread 1:
    Create an instance of subscriber S0 with selected QoS profile in domain: Domain-0
    Create and register a DataReaderDR-0 for the subscriber S0 that uses topic [name:
    recv_primes_result]
    While (Result sample RS-0 not complete)
      If new data from sample RS received
        Get worker number i
        Output the largest prime found
      End if
    End while

Worker nodes (Wi):
Create an instance of subscriber Si with selected QoS profile in domain: Domain-0
Create and register a DataReader DR-i for the subscriber Si that uses topic [name: send_interval_data]
While (!timeout && data !received)
  If sample SS-0 received
    Get worker number (Wi)
    mystart = (Wi*2)+1
    For every element n in the partial interval:
      SqrRoot = integer:sqrt(n);
      for i from 1 to SqrRoot
        i = i +2
        if ((n%i)==0)
          output the prime number n
          else "it is composite"
      End if
    End while

Create an instance of publisher Pi with selected QoS profile in domain: Domain-0
Create a DDS topic [name: recv_primes_result]
Create and register a DataWriter DW-i for the publisher Pi that uses the created topic
Create an instance of the topic (data sample)

Initialize the data structure Result Sample RSi: (the interval size, number of workers, i).
Publish the Result sample RS-0 through DataWriters DW-i
```

Appendix 4: The Node-to-Node Pseudo-code Using DDS

```
Master node:
  Thread 0:
    Create an instance of publisher P0 with selected QoS profile in domain: Domain-0
    Create a DDS topic [name: send_stream_data]
    Create and register a DataWriterDW-0 for the publisher P0 that uses the created topic
    Create an instance of the topic (data sample)
    Initialize the input data
      for (i=0; i<msgsize; i++) {
        X[i] = 1;
        Y[i] = 2;
      }
    Initialize the data structure Source Sample SS: (the input size, no. of workers).
    Publish the Source sample SS-0 through DataWritersDW-0

  Thread 1:
    Create an instance of subscriber S0 with selected QoS profile in domain: Domain-0
    Create and register a DataReaderDR-0 for the subscriber S0 that uses topic [name:
    rcv_stream_result]
    While (Result sample RS-0 not complete)
      If new data from sample RS received
        Set worker number = 0
        Output the values of x[0] and y[0] for confirmation
      End if
    End while

Worker node (W0):
  Create an instance of subscriber Si with selected QoS profile in domain: Domain-0
  Create and register a DataReaderDR-0 for the subscriber S0 that uses topic [name: send_stream_data]
  While (!timeout && data !received)
    Wait for SS-0 data to completion
  End while
  If sample SS-0 received
    Set worker number (W0)
  End if

  Create an instance of publisher P0 with selected QoS profile in domain: Domain-0
  Create a DDS topic [name: rcv_primes_result]
  Create and register a DataWriter DW-0 for the publisher Pi that uses the created topic
  Create an instance of the topic (data sample)

  Initialize the data structure Result Sample RS0: (the input size, number of workers).
  Publish the Result sample RS-0 through DataWriters DW-0
```

REFERENCES

- [1] N. Wang, Schmidt, D.C., V. Hag, H. Corsaro, A., "Toward an adaptive data distribution service for dynamic large-scale network-centric operation and warfare (NCOW) systems", in Military Communications Conference, 2008. MILCOM 2008. IEEE, Nov. 2008.
- [2] S.Spetka, S. Tucker, G. Linderman, "Information Management for High Performance Autonomous Intelligent Systems", in Performance Metrics for Intelligent Systems Workshop, PerMIS '07, Courtyard Gaithersburg Washingtonian Center, Gaithersburg, MD, August, 2007.
- [3] Spetka, S.E., Ramseyer, G.O., Linderman, R.W., "Grid Technology and Information Management for Command and Control", 10th International Command and Control Research and Technology Symposium, the Future of C2, McLean, Virginia, VA, June, 2005.
- [4] Eui-Nam Huh, "Sensor Event Processing on Grid", Technical report. Department of Computer Engineering Kyung Hee University 2005.
- [5] Jeffery Steinman, "The Wrap VI Simulation Kernel", Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation, 2005.
- [6] K.H. Kim, "Wide-area Real-Time Distributed Computing in a Tightly Managed Optical Grid: An Optiputer Vision", in: Proceedings of the 18th International Conference on Advanced Information Networking and Applications, AINA '04, vol. 2, March, IEEE Computer Society, 2004.
- [7] P. Grace, G. Coulson, G. Blair, et al. "GRIDKIT: Pluggable Overlay Networks for Grid Computing", in Proceedings Distributed Objects and Applications (DOA'04), Lecture Notes in Computer Science 3291, Springer-Verlag. ISBN: 3-540-23662-7.
- [8] S. Oh, J. Kim, G. Fox, "Real-Time Performance Analysis for Publish/Subscribe Systems", Future Generation Computer Systems, Sep., 2009.
- [9] Y. Wang¹, S. Yang¹, Alan Grigg², Julian Johnson, "DDS Based Framework for Remote Integration over the Internet", in the 7th Annual Conference on Systems Engineering Research 2009 (CSER 2009).
- [10] L. Srinivasan, J. Treadwell, "An Overview of Service-oriented Architecture, Web Services and Grid Computing", HP Software Global Business Unit, Nov. 2005.
- [11] Java Business Integration. Available at: www.rl.af.mil/programs/jbi/
- [12] Globus Toolkit. Available at: www.globus.org/toolkit/
- [13] R. Eduardo, M. Gil, B. Joao, "The use of real-time publish-subscribe middleware in networked vehicle systems", 1st IFAC Workshop on Multivehicle Systems (MVS'06), Brazil, Oct., 2006.

- [14] D. Prabu, et al., “An Efficient Run Time Interface for Heterogeneous Architecture of Large Scale Supercomputing System”, World Academy of Science, Engineering and Technology, 2006.
- [15] F. Pister, L. Hess and V. Lindenstruth, “Fault Tolerant Grid and Cluster Systems”, Kirchhoff Institute of Physics (KIP), University Heidelberg, Germany.
- [16] I. Haddad, C. Leangsuksun, R. Libby, T. Liu, Y. Liu, S. Scott, “Highly Reliable Linux HPC Clusters: Self-awareness Approach”, Proc. of the 2nd International Symposium on Parallel and Distributed Processing and Applications, 2004.
- [17] A. Azagury, D. Dolev, G. Gofit, John M. Marberg, J. Satran, “Highly Available Cluster: A Case Study”, FTCS 1994: 404-413.
- [18] Fagg, G., Dongarra, J., “Building and using a Fault Tolerant MPI implementation”, Int’l Journal of High Performance Applications and Supercomputing, 2004.
- [19] A. Gidenstam, B. Koldehofe, M. Papatriantafidou, and P. Tsigas, “Dynamic and Fault-Tolerant Cluster Management”. In Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing, pages 237–244, Aug. 2005
- [20] W. Gropp, E. Lusk, “Fault Tolerance in MPI Programs”, Journal of High Performance Computing and Applications, 2003.
- [21] J. Mugler, T. Naughton, S. Scott, C. Leangsuksun, “OSCAR Clusters”, Proceeding of The Linux Symposium 2003, July 23rd-26th, 2004.
- [22] “Data Distribution Service for Real-time Systems, v1.0,” Object Management Group Specification Document, Dated 2004-12-02, available at <http://www.omg.org>.
- [23] G. Pardo-Castellote, “DDS Spec Outfits Publish-Subscribe Technology for the GIG,” COTS Journal, April 2005.
- [24] Adams, J., Laverell, D., Ryken, M. MBH’99: A Beowulf Cluster Capstone Project, Proceedings of the 14th Annual Midwest Computer Conference, Whitewater, WI, March 2000.
- [25] Open DDS Specifications. Available at: www.opendds.org
- [26] J. Neelamegam, S. Chakravarthi, M. Apte, A. Skjellum, “PromisQoS: An Architecture for Delivering QoS to High-Performance Applications on Myrinet Clusters”, 28th Annual IEEE International Conference on Local Computer Networks (LCN’03), Oct., 2003.
- [27] A. Hafid, G.Bochmann, and B. Kerherve. “A Quality of Service Negotiation Procedure for Distributed Multimedia Presentational Applications”. In HPDC ’96, pages 330–339, 1996.
- [28] H. Chu, K Nahrstedt, “A CPUServiceClasses for Multimedia Applications”,. In IEEE Multimedia Systems Journal, 1999.
- [29] I. Foster, A. Roy, and V. Sander. “A Quality of Service Architecture That Combines Resource Reservation and Application Adaptation”, In Proceedings of the 8th International Workshop on Quality of Service (IWQOS), pages 181–188, Pittsburgh, PA, June 2000.

- [30] I. Cardei, R. Jha, M. Cardei, and A. Pavan. “Hierarchical Architecture for Real-Time Adaptive Resource Management”, In IFIP/ACM International Conference on Distributed Systems Platforms, pages 415–434, 2000.
- [31] N.J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. “A Gigabit-per-second Local Area Network”, IEEE Micro 15, pp 29-36 Feb., 1995
- [32] HariSubramoni, Ping Lai, and Dhabaleswar K. Panda, “Designing QoS Aware MPI for InfiniBand”, Technical Report, Department of Computer Science and Engineering, The Ohio State University.
- [33] The Open Fabric Alliance, available at: <http://www.openfabrics.org/>
- [34] The Open Grid Services Architecture (OGSA). Available at: <http://www.globus.org/ogsa/>
- [35] M. Ghuson, R. AlShaikh, M. Baddourah, “Performance Evaluation of Myrinet and Cisco Infiniband Using Intel MPI Middleware”, the 9th LCI International Conference on High Performance Computing, NCSA, University of Illinois, USA, May 2008.
- [36] The Green Top500 List. Available at: <http://www.green500.org/>
- [37] J. Sloan, “High performance Linux clusters with OSCAR, Rocks, openMosix, and MPI”, O’Reilly Publication, 2005.
- [38] J. Laros and L. Ward, “Implementing Scalable Diskless Clusters Using the Network File System”, Proceedings of the Los Alamos Computer Science Institute (LACSI) Symposium 2003, USA, October, 2003.
- [39] B. Guler, M. Hussain; T. Leng, and V. Mashayekhi, “The Advantages of Diskless HPC Clusters using NAS”, DELL Inc., Nov. 2002.
- [40] C. Yang and Y. Chang, “A Linux PC Cluster with Diskless Slave Nodes for Parallel Computing”, High-Performance Computing Laboratory, Department of Computer Science and Information Engineering, Tunghai University, Jan, 2003.
- [41] C. Engelmann, H. Ong and S. Scott, “Evaluating the Shared Root File System Approach for Diskless High-Performance Computing Systems”, Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing (LCI-09), Colorado, 2009.
- [42] Terry Jones, Andrew Tauferner, Todd Inglett, et al., “HPC Colony: Linux at Large Node Counts Report from Experiments Conducted on Sixth BGW Day”, August 10, 2007
- [43] J. Laros, C. Segura and N. Dauchy, “A Minimal Linux Environment for High Performance Computing Systems”, The 10th World Multi-Conference on Systemics, Cybernetics and Informatics, Florida, July 2006, pp.130-138.
- [44] C. Lu., “Scalable Diskless Checkpointing for Large Parallel Systems”, MSc. Thesis, University of Illinois at Urbana-Champaign, 2002.

- [45] B. Maher, “Techniques to Build a Diskless Boot Linux Cluster of JS21 Blades”, IBM Red Book, 2006.
- [46] T. Morgan JR., “DRBL: Diskless Remote Boot in Linux”, Master’s Capstone Project on High Performance Computing, April, 2006.
- [47] Z. Chen, G. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. “Building fault survivable MPI programs with FT-MPI using diskless Checkpointing”, Proceedings of the 10th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP), Chicago IL, June 2005, pp.213-223.
- [48] TOP500 Supercomputers. Available at: <http://www.top500.org>
- [49] HPL - High-Performance Linpack Benchmark. Available at: <http://www.netlib.org/benchmark/hpl>
- [50] S. Frank and R. Haskin, “GPFS: A Shared-Disk File System for Large Computing Clusters”, Proceedings of 1st Conference on File and Storage Technologies (FAST), USA, Jan., 2002, pp. 231–244.
- [51] P. Reisner and L. Ellenberg, “Replicated Storage with Shared Disk Semantics”, Proceedings of the 12th International Linux System Technology Conference (Linux-Kongress), Germany, Oct, 2005, pp.111-119.
- [52] DELL IPMI. Available at: <http://linux.dell.com/ipmi.shtml>
- [53] DELL Blades Server for HPC M610. Available at: <http://www.dell.com/us/en/enterprise/servers/server-poweredge-m610>.
- [54] C. Juszczak, “Improving the Write Performance of an NFS Server”, Proceedings of the USENIX Winter 1994 Technical Conference, USENIX, Association Berkeley, CA, USA, pp. 20-20, 1994.
- [55] Red Hat Knowledge Base: The Optimal Number of nfsd Threads. Available at: <http://kbase.redhat.com/faq/docs/DOC-2237>
- [56] Pallas Benchmarking tools. Available at: <http://people.cs.uchicago.edu/~hai/vcluster/PMB/>
- [57] J. Dongarra, J. Luszczek, and A. Petit, “The LINPACK benchmark: past, present and future”, in the Journal of Concurrency and Computation: Practice and Experience, 2003, pp. 803-820.
- [58] Energy Information Administration, USA Department of Energy http://www.eia.doe.gov/cneaf/electricity/epm/table5_6_b.html.
- [59] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda, “Host-Assisted Zero-Copy Remote Memory Access Communication on InfiniBand”, Int’l Parallel and Distributed Processing Symposium (IPDPS 04), April, 2004.
- [60] C. Bell, D. Bonachea, Y. Cote and et al. “An Evaluation of Current High-Performance Networks”, Int’l Parallel and Distributed Processing Symposium (IPDPS’03), April 2003.

- [61] J. Liu, B. Chandrasekaran, J. Wu and et al., “Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics”, Supercomputing, ACM/IEEE, pages 58- 58, Nov. 2003.
- [62] Myrinet, Myricom. Available at: <http://www.myri.com>
- [63] R. Fatoohi, K. Kardys, S. Koshy and el at. “Performance evaluation of high-speed interconnects using dense communication patterns”, Parallel Computing Volume 32, Issue 11-12, pages 794-807, 2006.
- [64] Intel Inc. Available at: <http://www.intel.com>
- [65] Portland PGI. Available at: <http://www.pgroup.com/>
- [66] The top500 supercomputers. Available at: <http://www.top500.org>
- [67] MVAPICH: MPI over InfiniBand and iWARP. Available at: <http://mvapich.cse.ohio-state.edu>
- [68] T. Typou, V. Stefanidis, P.D. Michailidis and K.G, “ Margaritis, Implementing Matrix Multiplication on an MPI Cluster of Workstations”, in Proceedings of the 1st In’t Conference "From Scientific Computing to Computational Engineering" (IC-SCCE'2004), Athens, Greece, vol. II, pp. 631-639, 2004
- [69] Lawrence Livermore National Laboratory – OpenMP tutorial. Available at: <https://computing.llnl.gov/tutorials/openMP/>
- [70] Simple matrix multiplication on MPI. Available at: <http://sushpa.wordpress.com/2008/05/20/simple-matrix-multiplication-on-mpi/>
- [71] A. Boukerche, R. Al-Shaikh, “Towards Highly Available and Scalable High Performance clusters”, as a special issue on Network-Based Computing in the Journal of Computer and System Sciences (in conjunction with IPDPS’06), 2006.
- [72] B. Madani, R. Al-Shaikh, “Towards RTPS Models for High Performance and Grid Computing”, the 14th International Conference on Petroleum Data Integration, Information and Data Management, USA, TX, May, 2010.
- [73] M. Al-Mulhem, R. Al-Shaikh, “Performance Evaluation of Intel and Portland Compilers Using Intel Westmere Processor”, in the 2nd IEEE conference on Intelligent Systems, Modeling and Simulation, Phnom Penh, Cambodia, January, 2011.
- [74] K Salah, R. Al-Shaikh, M. Sindi, “Towards Green Computing Using Diskless High Performance Clusters”, in the 7th IEEE Int’l Conference on Network and Service Management (CNSM’11), Paris, France, October 2011.
- [75] B. Madani, R. Al-Shaikh, “Performance Modeling and MPI Evaluation Using Westmere-based Infiniband HPC Cluster”, the 4th IEEE European Symposium in Mathematical Modeling and Computer Simulation”, Pisa, Italy, November, 2010.
- [76] Cluster Resources. Available at: <http://www.clusterresources.com>
- [77] Real-Time Innovations (RTI). Available at: <http://www.rti.com>

- [78] A. Agbaria, R. Friedman, “Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations”, In the 8th IEEE International Symposium on High Performance Distributed Computing, 1999.
- [79] L. Dikken, F. Linden, J. Vasseur and P. Sloot, “DynamicPVM: Dynamic Load Balancing on Parallel Systems”, In W. Gentsch and U. Harms, editors, High Performance Computing and Networking, pp. 273-277, April 1994, Springer, LNCS 797.
- [80] NCBI BLAST, available at: <http://blast.ncbi.nlm.nih.gov/Blast.cgi>
- [81] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, W. Feng, “Massively parallel genomic sequence search on the Blue Gene/P architecture”, SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing.
- [82] A. Darling, L. Carey, and W. Feng, “The Design, Implementation, and Evaluation of mpiBLAST”, 4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with ClusterWorld Conference & Expo, June 2003.
- [83] L. Chai, R. Noronha, P. Gupta, G. Brown and D. K. Panda, "Designing a Portable MPI-2 over Modern Interconnects Using uDAPL Interface", Recent Advances in Parallel Virtual Machine and Message Passing Interface”, Lecture Notes in Computer Science, 2005, Volume 3666/2005, pp. 200-208.
- [84] Mellanox OFED User’s Manual. Available at: www.mellanox.com
- [85] QLogic Infiniband. Available at: www.qlogic.com
- [86] D.Buntinas, B.Goglin, D.Goodell, G. Mercier, S.Moreaud, “Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis”, International conference on Parallel Processing (ICPP’09), Vienna, 2009.
- [87] B. Schroeder, G. Gibson, “Understanding Failures in PetascaleComputers,” Journal of Physics: Conference Series 78 (2007),SciDAC 2007.
- [88] B. Murphy and T. Gent, “Measuring System and Software Reliability Using an Automated Data Collection Process”. Qualityand Reliability Engineering International, 11(5), 1995.
- [89] B. Schroeder, G. Gibson, “A Large Scale Study of Failures in High-performance-computing Systems”, Int’l Symposium on Dependable Systems and Networks (DSN 2006). IEEE Transactions on Dependable and Secure Computing (TDSC).
- [90] G. Pardo-Castellote, “OMG Data-Distribution Service: Architectural Overview”, MILCOM’03 Proceedings of the 2003 IEEE conference on Military communications - Volume I, 2003.
- [91] The MPI Prime Search. Available at: <https://computing.llnl.gov/tutorials/mpi/>

Vitae

- Raed Abdullah Al-Shaikh.
- Saudi Nationality.
- Born in August 4, 1978.
- Earned a Bachelor of Science degree in Computer Engineering from King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia, in June 2001.
- Earned a Master in Business Administration (MBA) degree from the University of Bahrain (UOB), Issa Town, Bahrain, in December 2004.
- Earned a Master degree in Computer Science (MCS) from the University of Ottawa (UofO), Ottawa, Canada, in May 2006.
- E-mail: raed.shaikh@aramco.com