High Performance and Grid Computing with Quality of Service Control

Sadiq M. Sait
Center for Communications and IT Research
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
sadiq@kfupm.edu.sa

Raed Al-Shaikh
EXPEC Computer Center (ECC)
Dhahran 31311, Saudi Arabia
Saudi Aramco
raed.shaikh@aramco.com

Abstract **- Up to writing this paper, existing High Performance Computing (HPC) systems do not provide proper quality of service (QoS) controls and reliability features because of two limitations: first, standard middleware 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. Second, modern high-speed interconnects such as Infiniband, Myrinet and Quadrics are optimized for performance rather than fault-tolerance and QoS control. The Data-Centric Publish-Subscribe (DCPS) model — 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. In this paper, we present our novel model of incorporating DDS QoS and reliability controls into HPC systems. Our results show that DDS integration into HPC adds considerable overheard in terms of performance and network utilization, when the application is mainly communication**

Index Terms— *HPC, QoS, Middleware, MPI.*

I. INTRODUCTION

In recent years, there has been a growing interest in the field of distributed computing, where diverse machines and subsystems 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 centers, and embedded or real-time environments [1].

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 [5]. 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 [3]. 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 configurations [4].

Despite the advantages of these distributed computing 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 any 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. 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) [6], 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 predefined QoS reliability controls in the more-specialized HPC environments and incorporate its most important QoS polices, which is one of the main aims of this research work.

The rest of the paper is organized as follows: in Section 2, we describe our motivation for this work. We then describe the general publish/subscribe framework in Data Distribution

Service in Section 3. In Section 4, we illustrate our model of integrating DDS into HPC. Section 5 presents our cluster setup and experimental results. We state our conclusion and future work in the last section.

II. 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 [7], we see that what used to be the #1 HPC cluster (i.e., The Jaguar Cluster) in June 2010 had 224,162 cores, while in November 2013, the #1 Tianhe-2 cluster had 3,120,000 cores. Notably, 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 [8, 9, 10]. B. Schroeder and G. Gibson [8] 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 1,100 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 [8].

In terms of storage and hard drive failures, the authors found 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 [11].

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 increase to achieve higher Petascale and even Exascale systems, it is expected that the system wide failure rate will increase [12].

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 2000, is taking only a few hours to run today. Therefore the simplest remedy for this failure rate is to resubmit the user job after offlining (i.e., fencing) the problematic node/core, rendering the wasted hours of the crashed job to be neglected. Therefore, 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. The goal of this study is to focus on the HPC QoS and reliability controls —and in different HPC layers —to benefit these very long users' jobs that require large-scale clusters to run.

In this paper, we present our work of adopting DDS standards into HPC, 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.

III. THE GENERAL PUBLIC/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 [6]. 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 1, 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. 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 [13] 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 it, using the keys, and delivers it to the recipients for processing.

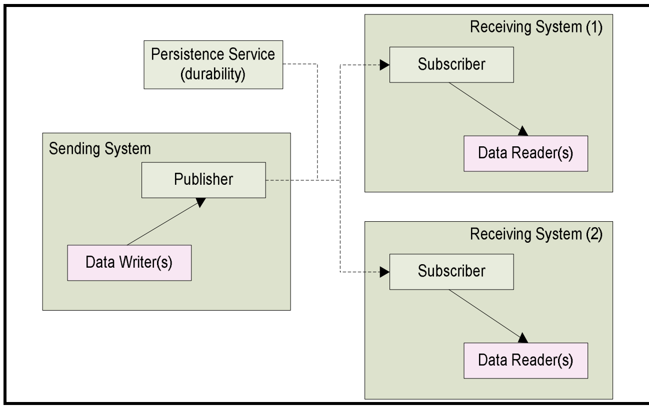

Figure 1: The general Publish-Subscribe model with persistence service

IV. THE HPC-DDS INTEGRATION MODEL

DDS QoS implementations usage has been limited to the generic heterogeneous computing environments. To the best of our knowledge, none of these attempts 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 2. 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 2: MPI vs. DDS layers

In DDS basic model, commutation between participants is achieved by having six essential entities, these are [15]: *DomainParticipant, DataWriter, DataReader, Publisher, Subscriber and Topic*. Working on these entities, our mapping of the DDS standard into HPC is illustrated in Figure 3 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.

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.

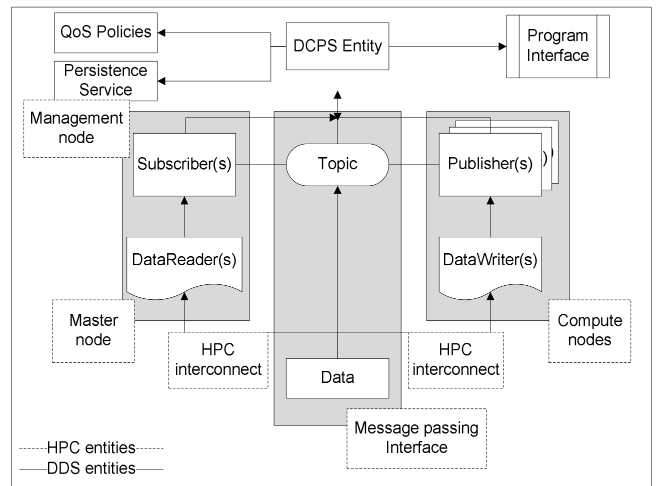

Figure 3: HPC-DDS integration model

Most node interactions in the basic DDS implementations are one-way communication, that is, from the publishers to subscribers. These publishers and subscribers have to reverse their roles to establish two-way communication. To mimic the two-way interaction between the master and compute nodes, and have it similar to the typical 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 Subscriber/DataReader for receiving the final data from the computes. Likewise, all compute nodes have the same structure for their two-way communication.

A. Implemented Quality of Service Policies

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 can be

found in the Object Management Group Specification Document [6].

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) 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 a fast-but-unreliable "best effort" to a 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_LIMITSQoS,willbe 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. I.e., the DDS middleware will attempt to find other paths and retransmit data-samples to reconstruct a correct snapshot of the DataWriter history before it is accessible by the recipients.

If the reliability type is set to "BEST_EFFORT," the service will not resend the missing data-samples, but will ensure that datasent 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 "1," 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.

V. EXPERIMENTAL SETUP AND METHODOLOGY

To evaluate the feasibility and impact of adopting the DDS QoS into HPC, we implemented two applications with different computational models. The first program "the prime numbers search," which is a modified version of Blaise'sMPI Prime [14], 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 second application "Node-to-Node Streaming" represents the communication-bound type of applications, in which it is used for streaming large amounts of data between compute nodes.

In both experiments, 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 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 for it to work in our design.

A. The Cluster Design

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.

B. *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. The primes search algorithm is purely computation-bound and requires minimal inter-node communication, since the collective communications calls on the nodes are used to collect the only two data elements requiring communication: the number of primes found andthe largest prime, regardless of the interval size.

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. 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 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_interval_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 reverse 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_interval_result." It then 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:

```
my_interval_start = (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.

1) *Performance Evaluation and Results*

In this section, we present our experimental results using the described 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 1 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 communicated at runtime.

Table 1: MPI vs. DDS Primes Search runtime while varying the number of nodes

	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 2 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.

Table 2: MPI vs. DDS Primes Search runtime while varying the input size on 32 nodes

	Number of elements to be searched (*in millions*)						
	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 3 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).

Table 3: The delay when engaging a new node in DDS while running Primes Search

	Number of elements to be searched (*in millions*)				
	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.

C. 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 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.

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 4 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.

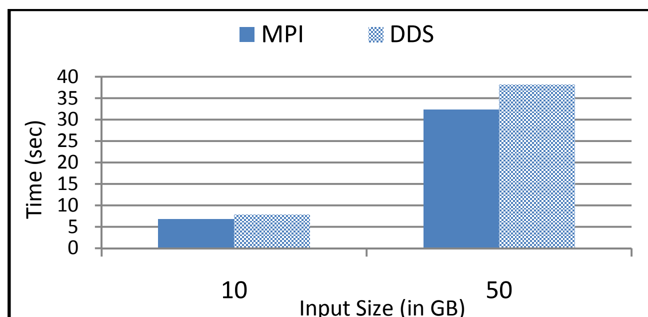

Figure 4: Node-to-Node Throughput

Similar to the matrix multiplication application, the QoS policy DDS_SYNCHRONOUS_PUBLISH_MODEhad 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 adds additional overhead as indicated in the test.

Figure 5 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 resulted in unrealistic readings due to the excessive storage

access overhead. This test is also not applicable to the MPI implementation, due to the absence of the fault-tolerance feature.

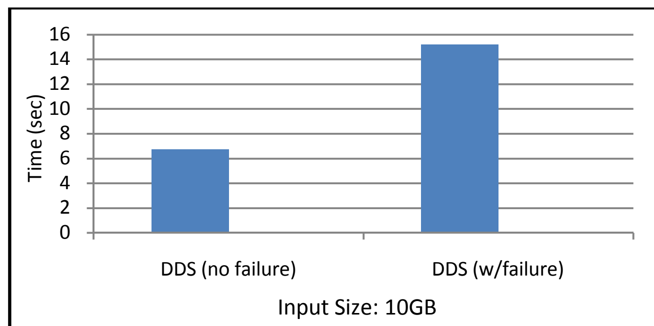

Figure 5: Failing the receiver in DDS while running the Node-to-Node application

VI. CONCLUSION

In this paper, we presented our work of adopting the DDS standard into HPC to circumvent the MPI shortcomings and provide QoS for HPC applications. As demonstrated in our tests, DDS integration into HPC adds considerable overheard 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.

ACKNOWLEDGMENT

The authors would like to thank King Fahd University for Petroleum and Minerals (KFUPM) and the EXPEC Computer Center (ECC) at Saudi Aramco for their invaluable support and contributions to this study.

REFERENCES

[1] 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.

[2] 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.

[3] F. Pister, L. Hess and V. Lindenstruth, "Fault Tolerant Grid and Cluster Systems," Kirchhoff Institute of Physics (KIP), University Heidelberg, Germany.

[4] 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.

[5] Jeffery Steinman, "The Wrap VI Simulation Kernel," Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation, 2005.

[6] "Data Distribution Service for Real-time Systems, v1.0," Object Management Group Specification Document, Dated 2004-12-02, available at http://www.omg.org.

[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] B. Schroeder, G. Gibson, "Understanding Failures in Petascale Computers," Journal of Physics: Conference Series 78 (2007),SciDAC 2007.

[9] B. Murphy and T. Gent, "Measuring System and Software Reliability Using an Automated Data Collection Process." Quality and Reliability Engineering International, 11(5), 1995.

[10] 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).

[11] 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.

[12] Y. Wang, S. Yang, Alan Grigg, Julian Johnson, "DDS Based Framework for Remote Integration over the Internet," in the 7th Annual Conference on Systems Engineering Research 2009 (CSER 2009).

[13] G. Pardo-Castellote, "DDS Spec Outfits Publish-Subscribe Technology for the GIG," COTS Journal, April 2005.

[14] The MPI Prime Search. Available at: https://computing.llnl.gov/tutorials/mpi/

[15] G. Pardo-Castellote, "OMGData-Distribution Service: Architectural Overview,"MILCOM'03 Proceedings of the 2003 IEEE Conference on Military Communications - Volume I, 2003.