Evaluating the Performance of Publish/Subscribe Platforms for Information Management in Distributed Real-time and Embedded Systems

Ming Xiong, Jeff Parsons, James Edmondson, Hieu Nguyen, and Douglas C Schmidt,

Vanderbilt University, Nashville TN, USA

Abstract[1]

Recent trends in distributed real-time and embedded (DRE) systems motivate the development of information management capabilities that ensure the right information is delivered to the right place at the right time to satisfy quality of service (QoS) requirements in heterogeneous environments. A promising approach to building and evolving large-scale and long-lived DRE information management systems are standards-based QoS-enabled publish/subscribe (pub/sub) platforms that enable participants to communicate by publishing information they have and subscribing to information they need in a timely manner. Since there is little existing evaluation of how well these platforms meet the performance needs of DRE information management, this paper provides two contributions: (1) it describes three common architectures for the OMG Data Distribution Service (DDS), which is a QoS-enabled pub/sub platform standard, and (2) it evaluates implementations of these architectures to investigate their design tradeoffs and compare their performance with each other and with other pub/sub middleware. Our results show that DDS implementations perform significantly better than non-DDS alternatives and are well-suited for certain classes of data-critical DRE information management systems.

Keywords: DRE Information Management; QoS-enabled Pub/Sub Platforms; Data Distribution Service;

1 Introduction

The OMG Data Distribution Service (DDS) [6] specification is a standard for QoS-enabled pub/sub communication aimed at mission-critical distributed real-time and embedded (DRE) systems. It is designed to provide (1) *location independence* via anonymous pub/sub protocols that enable communication between collocated or remote publishers and subscribers, (2) *scalability* by supporting large numbers of topics, data readers, and data writers, and (3) *platform portability and interoperability* via standard interfaces and transport protocols. Multiple implementations of DDS are now available, ranging from high-end COTS products to open-source community-supported projects. DDS is used in a wide range of DRE systems, including traffic monitoring [14], controlling unmanned vehicle communication with their ground stations [16], and in semiconductor fabrication devices [15].

Although DDS is designed to be scalable, efficient, and predictable, few researchers have evaluated and compared the performance of DDS implementations empirically for common DRE information management scenarios. Likewise, little published work has systematically compared DDS with alternative non-DDS pub/sub middleware platforms. This paper addresses this gap in the R&D literature by describing the results of the Pollux project, which is evaluating a range of pub/sub platforms to compare how their architecture and design features affect their performance and suitability of DRE information management. This paper also describes the design and application of an open-source DDS benchmarking environment we developed as part of Pollux to automate the comparison of pub/sub latency, jitter, throughput, and scalability.

The remainder of this paper is organized as follows: Section 2 summarizes the DDS specification and the architectural differences of three popular DDS implementations; Section 3 describes our ISISlab hardware testbed and open-source DDS Benchmark Environment (DBE); Section 4 analyzes the results of benchmarks conducted using DBE in ISISlab; Section 5 presents the lessons learned from our experiments; Section 6 compares our work with related research on pub/sub platforms; and Section 7 presents concluding remarks and outlines our future R&D directions.

2 Overview of DDS

2.1 Core Features and Benefits of DDS

The OMG Data Distribution Service (DDS) specification provides a data-centric communication standard for a range of DRE computing environments, from small networked embedded systems up to large-scale information backbones. At the core of DDS is the *Data-Centric Publish-Subscribe* (DCPS) model, whose specification defines standard interfaces that enable applications running on heterogeneous platforms to write/read data to/from a global data space in a DRE system. Applications that want to share information with others can use this global data space to declare their intent to publish data that is categorized into one or more topics of interest to participants. Similarly, applications can use this data space to declare their intent to become subscribers and access topics of interest. The underlying DCPS middleware propagates data samples written by publishers into the global data space, where it is disseminated to interested subscribers [6].

[1] This work was sponsored in part by the AFRL/IF Pollux project and Vanderbilt University's Summer Undergraduate Research program.

The DCPS model decouples the declaration of information access intent from the information access itself, thereby enabling the DDS middleware to support and optimize QoS-enabled communication.

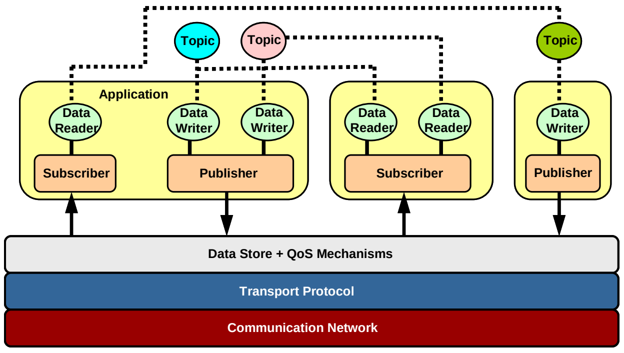

Figure 1: Architecture of DDS

The following DDS entities are involved in a DCPS-based application, as shown in Figure 1:

- **Domains**. DDS applications send and receive data within a *domain*. Only participants within the same domain can communicate, which helps isolate and optimize communication within a community that shares common interests.
- **Data Writers/Readers and Publishers/Subscribers**. Applications use *data writers* to publish data values to the global data space of a domain and data readers to receive data. A *publisher* is a factory that creates and manages a group of data writers with similar behavior or QoS policies. A *subscriber* is a factory that creates and manages data readers.
- **Topics**. A *topic* connects a data writer with a data reader. Communication happens only if the topic published by a data writer matches a topic subscribed to by a data reader. Communication via topics is anonymous and transparent, *i.e.*, publishers and subscribers need not be concerned with how topics are created nor who is writing/reading them since the DDS DCPS middleware manages these issues.

The remainder of this subsection describes the benefits of DDS relative to conventional pub/sub middleware and client/server-based SOA platforms.

Figures 2 and 3 show DDS capabilities that make it better suited than other standard middleware platforms as the basis of DRE information management. Figure 2(A) shows that DDS has *fewer layers* than conventional SOA standards, such as CORBA, .NET, and J2EE, which can reduce latency and jitter significantly, as shown in Section 4. Figure 2(B) shows that DDS supports many *QoS policies*, such as the lifetime of each data sample, the degree and scope of coherency for information updates, the frequency of information updates, the maximum latency of data delivery, the priority of data delivery, the reliability of data delivery, how to arbitrate simultaneous modifications to shared data by multiple writers, mechanisms to assert and determine liveliness, parameters for filtering by data receivers, the duration of data validity, and the depth of the 'history' included in updates.

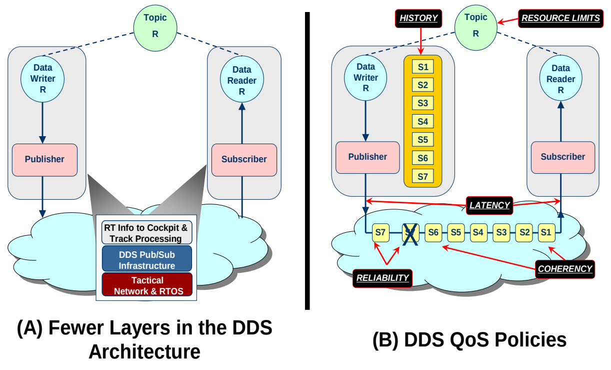

Figure 2: DDS Optimizations & QoS Capabilities

These QoS policies can be configured at various levels of granularity (*i.e.*, topics, publishers, data writers, subscribers, and data readers) thereby allowing application developers to construct customized contracts based on the specific QoS requirements of individual entities. Since the identity of publishers and subscribers are unknown to each other, the DCPS middleware is responsible for determining whether QoS policies offered by a publisher are compatible with those required by a subscriber, allowing data distribution only when compatibility is satisfied.

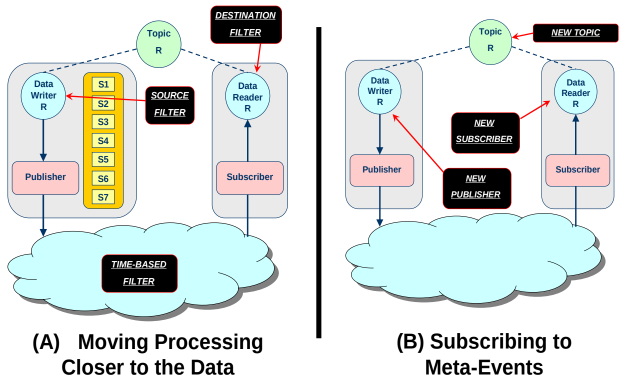

Figure 3: DDS Filtering & Meta-event Capabilities

Figure 3(A) shows how DDS can *migrate processing closer to the data source*, which reduces bandwidth in resource-constrained network links. Figure 3(B) shows how DDS enables clients to *subscribe to meta-events* that they can use to detect dynamic changes in network topology, membership, and QoS levels. This mechanism helps DRE information management systems adapt to environments that change continuously.

2.2 Alternative DDS Implementations

The DDS specification defines a wide range of QoS policies (outlined in Section 2.1) and interfaces used to exchange topic instances between participants. The specification intentionally does not address how to implement the services or manage DDS resources internally, so DDS providers are free to innovate. Naturally, the communication models, distribution architectures, and implementation techniques used by DDS providers significantly impact application behaviour and QoS, *i.e.*, different choices affect the suitability of DDS implementations and configurations for various types of DRE information management applications.

Table 1: Supported DDS Communication Models

Impl	Unicast	Multicast	Broadcast
DDS1	Yes (default)	Yes	No
DDS2	No	Yes	Yes (default)
DDS3	Yes (default)	No	No

By design, the DDS specification allows DCPS implementations and applications to take advantage of various communication models, such as unicast, multicast, and broadcast transports. The communication models supported for the three most popular DDS implementations we evaluated are shown in Table 1.[2] DDS1 supports unicast and multicast, DDS2 supports multicast and broadcast, whereas DDS3 supports only unicast. These DDS implementations all use layer 3 network interfaces, *i.e.*, IP multicast and broadcast, to handle the network traffic for different communication models, rather than more scalable multicast protocols, such as Richocet [5], which combine native IP group communication with proactive forward error correction to achieve high levels of consistency with stable and tunable overhead. Our evaluation also found that these three DDS implementations have different architectural designs, as described in the remainder of this section.

2.2.1 Federated Architecture

The federated DDS architecture shown in Figure 4 uses a separate DCPS daemon process for each network interface. This daemon must be started on each node before domain participants can communicate. Once started, it communicates with DCPS daemons running on other nodes and establishes data channels based on reliability requirements (*e.g.*, reliable or best-effort) and transport addresses (*e.g.*, unicast or multicast). Each channel handles communication and QoS for all the participants requiring its particular properties. Using a daemon process decouples the applications (which run in a separate user process) from DCPS configuration and communication-related details. For example, the daemon process can use a configuration file to store common system parameters shared by communication endpoints associated with a network interface, so that changing the configuration does not affect application code or processing.

In general, a federated architecture allows applications to scale to a larger number of DDS participants on the same node, *e.g.*, by bundling messages that originate from different DDS participants. Moreover, using a separate daemon process to mediate access to the network can (1) simplify application configuration of policies for a group of participants associated with the same network interface and (2) prioritize messages from different communication channels.

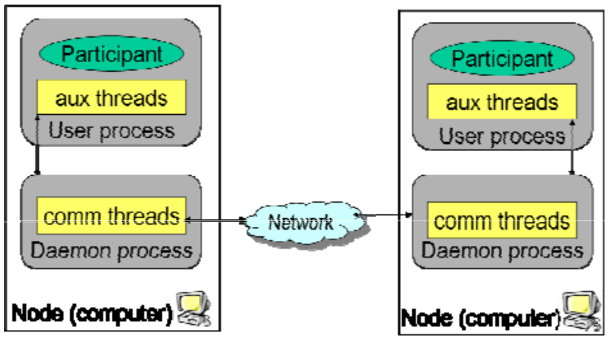

Figure 4: Federated DDS Architecture

A disadvantage of the daemon-based approach, however, is that it introduces an extra configuration step—and possibly another point of failure. Moreover, applications must cross extra process boundaries to communicate, which can increase latency and jitter, as shown in Section 4.

2.2.2 Decentralized Architecture

The decentralized DDS architecture shown in Figure 5 places the communication- and configuration-related capabilities into the same user process as the application itself. These capabilities execute in separate threads (rather than in a separate daemon process) that the DCPS middleware library uses to handle communication and QoS.

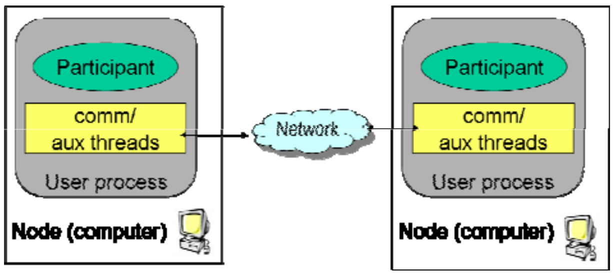

Figure 5: Decentralized DDS Architecture

The advantage of a decentralized architecture is that each application is self-contained, without needing a separate daemon. As a result, latency and jitter are reduced, and there is one less configuration and failure point. A disadvantage, however, is that specific configuration details, such as multicast address, port number, reliability model, and parameters associated with different transports, must be defined at the application level, which is tedious and error-prone. This architecture also makes it hard to buffer data sent between mul-

[2] The specific DDS product names are "shrouded" pending final approval from the companies that produce them. The actual names will appear in the published paper.

tiple DDS applications on a node, and thus does not provide the same scalability benefits as the federated architecture described in Section 2.2.1.

2.2.3 Centralized Architecture

The centralized architecture shown in Figure 6 uses a single daemon server running on a designated node to store the information needed to create and manage connections between DDS participants in a domain. The data itself passes directly from publishers to subscribers, whereas the control and initialization activities (such as data type registration, topic creation, and QoS value assignment, modification and matching) require communication with this daemon server.

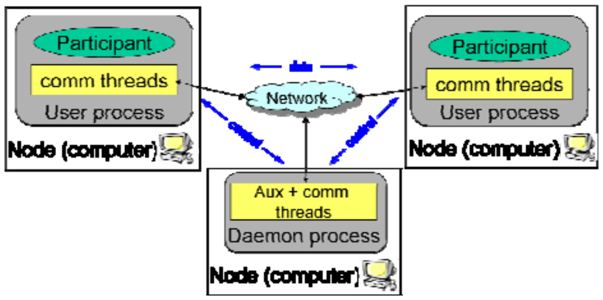

Figure 6: Centralized DDS Architecture

The advantage of the centralized approach is its simplicity of implementation and configuration since all control information resides in a single location. The disadvantage, of course, is that the daemon is a single point of failure, as well as a potential performance bottleneck in a highly loaded system.

The remainder of this paper investigates how the architecture differences described above can affect the performance experienced by DRE information management applications.

3 Methodology for Pub/Sub Platform Evaluation

This section describes our methodology for evaluating pub/sub platforms to determine how well they support various classes of DRE information management applications, including systems that generate small amounts of data periodically (which require low latency and jitter), systems that send larger amount of data in bursts (which require high throughput), and systems that generate alarms (which require asynchronous, prioritized delivery).

3.1 Evaluated Pub/Sub Platforms

In our evaluations, we compare the performance of the C++ implementations of DDS shown in Table 2 against each other. We also compare these three DDS implementations against three other pub/sub middleware platforms, which are shown in Table 3.

Table 2: DDS Versions Used in Experiments

Impl	Version	Distribution Architecture
DDS1	4.1c	Decentralized Architecture
DDS2	2.0 Beta	Federated Architecture
DDS3	8.0	Centralized Architecture

Table 3: Other Pub/Sub Platforms in Experiments

Platform	Version	Summary
CORBA Notification Service	TAO 1.5	OMG data interoperability standard that enables events to be sent & received between objects in a decoupled fashion
SOAP	gSOAP 2.7.8	W3C standard for an XML-based Web Service
JMS	J2EE 1.4 SDK/ JMS 1.1	Enterprise messaging standards that enable J2EE components to communicate asynchronously & reliably

We compare the performance of these pub/sub mechanisms by using the following metrics:

- **Latency**, which is defined as the roundtrip time between the sending of a message and reception of an acknowledgment from the subscriber. In our test, the roundtrip latency is calculated as the average value of 10,000 round trip measurements.
- **Jitter**, which is the standard deviation of the latency.
- **Throughput**, which is defined as the total number of bytes received per unit time in different 1-to-n (*i.e.*, 1-to-4, 1-to-8, and 1-to-12) publisher/subscriber configurations.

We also compare the performance of the DDS asynchronous listener-based and synchronous waitset-based subscriber notification mechanisms. The listener-based mechanism uses a callback routine (the *listener*) that the DDS service invokes when data is available. The waitset-based mechanism sets up a sequence (the *waitset*) containing user-defined conditions and a designated application thread will sleep on the waitset until these conditions are met.

3.2 Benchmarking Environment

3.2.1 Hardware and Software Infrastructure

The computing nodes we used to run our experiments are hosted on ISISlab [19], which is a testbed of computers and network switches that can be arranged in many configurations. ISISlab consists of 6 Cisco 3750G-24TS switches, 1 Cisco 3750G-48TS switch, 4 IBM Blade Centers each consisting of 14 blades (for a total of 56 blades), 4 gigabit network IO modules and 1 management modules. Each blade has two 2.8 GHz Xeon CPUs, 1GB of ram, 40GB HDD, and 4 independent Gbps network interfaces. In our tests, we used up to 14 nodes (1 pub, 12 subs, and a centralized server in the case of DDS3). Each blade ran Fedora Core 4 Linux, version 2.6.16-1.2108_FC4smp. The DDS applications were run in the Linux real-time scheduling class to minimize extraneous sources of memory, CPU, and network load.

3.2.2 DDS Benchmark Environment (DBE)

To facilitate the growth of our tests both in variety and complexity, we created the *DDS Benchmarking Environment* (DBE), which is an open-source framework for automating our DDS testing. The DBE consists of (1) a repository that contains scripts, configuration files, test ids, and test results, (2) a hierarchy of Perl scripts to automate test setup and execution, (3) a tool for automated graph generation, and (4) a shared library for gathering results and calculating statistics.

The DBE has three levels of execution designed to enhance flexibility, performance, and portability, while incurring low overhead. Each level of execution has a specific purpose: the top level is the user interface, the second level manipulates the node itself, and the bottom level is comprised of the actual executables (*e.g.*, publishers and subscribers for each DDS implementation). DBE runs the actual test executables rather than by the DBE scripts. For example, if Ethernet saturation is reached during our testing, the saturation is accomplished by relevant DDS data transmissions, not by DBE test artifacts.

4 Empirical Results

This section analyzes the results of benchmarks conducted using DBE in ISISlab. We first evaluate 1-to-1 roundtrip latency performance of DDS pub/sub implementations within a single blade and compare them with the performance of non-DDS pub/sub implementations in the same configuration. We then analyze the results of 1-to-n scalability throughput tests for each DDS implementations, where n is 4, 8, or 12 blades on ISISlab. All graphs of empirical results use logarithmic axes since the latency/throughput of some pub/sub implementations cover such a large range of values that linear axes display unreadably small values over part of the range of payload sizes.

4.1 Latency and Jitter results

Benchmark design. Latency is an important measurement to evaluate DRE information management performance. Our test code measures roundtrip latency for each pub/sub middleware platform described in Section 3.1. We ran the tests on both simple and complex data types to see how well each platform handles the extra (de)marshaling overhead introduced for complex data types. The IDL structure for the simple and complex data types are shown below.

```
// Simple Sequence Type
Struct data
{ long index; sequence<octet> data; }

// Complex Sequence Type
struct Inner { string info; long index; };
typedef sequence<Inner> InnerSeq;
struct Outer
{ long length; InnerSeq nested_member; };
typedef sequence<Outer> ComplexSeq;
```

The publisher writes a simple or complex data sequence of a certain payload size. When the subscriber receives the topic data it replies with a 4-byte acknowledgement. The payload length for both simple and complex data ranges from 4 to 16,384 by powers of 2.

The basic building blocks of the JMS test codes consist of administrative objects used by J2EE Application server, a message publisher, and a message subscriber. JMS supports both point-to-point and publish/subscribe message domains. Since we are measuring the latency of one publisher to one subscriber, we chose a point-to-point message domain that uses a synchronous message queues to send and receive data.

TAO's Notification Service uses a Real-time CORBA threadpool lane to create a priority path between publisher and subscriber. The publisher creates an event channel and specifies a lane with priority 0. The subscriber also chooses a lane with priority 0. The Notification Service then matches the lane for the subscriber and sends the event to the subscriber using the CLIENT_PROPAGATED priority model. .

The publisher test code measures latency by timestamping the data transmission and subtracting that from the timestamp value when it receives the subscriber ack. This test is to evaluate how fast data gets transferred from one pub/sub node to another at different payload sizes. To eliminate factors other than differences among the middleware implementations that affect latency/jitter, we tested a single publisher sending to a single subscriber running in separate processes on the same node. Since each node has two CPUs the publisher and subscriber can execute in parallel.

Results. Figures 7 and Figure 8 compare latency/jitter results for simple/complex data types for all pub/sub platforms. These figures show how DDS latency/jitter is much lower than conventional pub/sub middleware for both simple and complex data types. In particular, DDS1 has much lower latency and jitter than the other pub/sub mechanisms.

Analysis. There are several reasons for the results in Figures 7 and 8. As discussed in Section 2.1, DDS has fewer layers than other standard pub/sub platforms, so it incurs lower latency and jitter. Since in each test the publisher and subscriber reside on the same blade, DDS1 and DDS2 both switch to shared memory transport to improve performance. The particularly low latency of DDS1 stems largely from its mature implementation, as well as its decentralized architecture shown in Figure 5, in which publishers communicate to subscribers without going through a separate daemon process. In contrast, DDS2's federated architecture involves an extra hop through a pair of daemon processes (one on the publisher and one on the subscriber), which helps explain why its latency is higher than DDS1's when sending small simple data types.

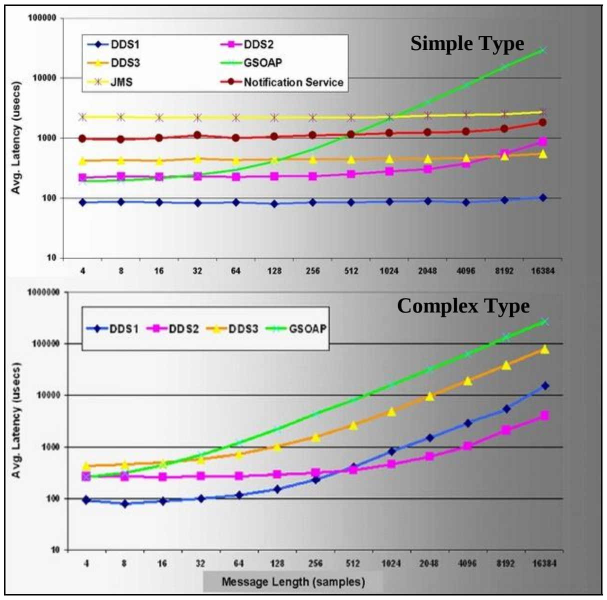

Figure 7: Simple Latency Vs Complex Latency

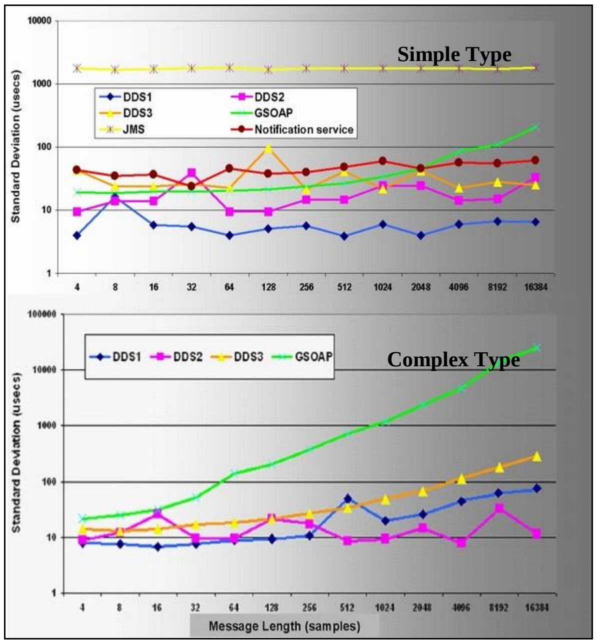

Figure 8: Simple Jitter Vs Complex Jitter

Figures 7 and 8 also show that sending complex data types incurs more overhead for all pub/sub implementations, particularly as data size increases. For complex data types we observe that although DDS1 outperforms DDS2 for smaller payloads, DDS2 catches up at 512 bytes and performs better for payload sizes above 512 bytes. This behavior is explained by the fact that DDS1 performs (de)marshaling even in the local blade, whereas DDS2 optimizes local blade performance by making a simple C++ call and bypassing (de)marshaling. The benefits of this DDS2 optimization, however, are apparent only for large data sizes on the same blade, as shown by the curve in Figure 7 comparing DDS1 and DDS2. We have run the same latency tests on two different blades and the results shown in [19] indicate that DDS1 performs better for all payload sizes tested since remote communication obviates the DDS2 optimization

Interestingly, the latency increase rate of GSOAP is nearly linear as payload size increases, *i.e.*, if the payload size doubles, the latency nearly doubles as well. GSOAP's poor performance with large payloads stems largely from its XML representation for sequences, which (de)marshals each element of a sequence using a verbose text-based format rather than (de)marshaling the sequence in blocks as DDS and CORBA do.

4.2 Throughput Results

Benchmark design. Throughput is another important performance metric for DRE information management systems. The primary goals of our throughput tests were therefore to measure how each DDS implementation handles scalability of subscribers and how different communication models (*e.g.*, unicast, multicast, and broadcast) affect performance, and how synchronous (waitset-based). To maximize scalability in our throughput tests, the publisher and subscriber(s) reside in different processes on different blades.

The remainder of this subsection first evaluates the performance results of three DDS implementations as we vary the communication models they support, as per the constraints outlined in Table 1. We then compare the multicast performance of DDS1 and DDS2, as well as the unicast performance of DDS1 and DDS3, which are the only common points of evaluation currently available. We focus our throughput tests on the DDS implementations and did not measure throughput for the other pub/sub platforms because our results in Section 4.1 show they were significantly outperformed by DDS.

The remainder of this section presents our throughput results. For each figure, we include a small text box with a brief description of the DDS QoS used for the current test. Any parameter that is not mentioned in the box uses the default value specified in the DDS specification [6]. Note that we have chosen the best-effort QoS policy for data delivery reliability, as mentioned in Section 1. The reason we focus on best-effort is that we are evaluating DRE information management scenarios where (1) information is updated frequently and (2) the overhead of retransmitting lost data samples is acceptable.

4.2.1 DDS1 Unicast/Multicast

Results. Figure 9 shows the results of our scalability tests for DDS1 unicast/multicast with 1 publisher and multiple subscribers. This figure shows that unicast performance degrades when scaling up the number of subscribers, whereas multicast throughput scales up more gracefully.

Analysis. Figure 9 shows that the overhead of unicast degrades performance as the number of subscribers increases since the middleware sends a copy to each subscriber. It also indicates how multicast improves scalability since only one copy of the data is delivered, regardless of the number of subscribers in the domain.

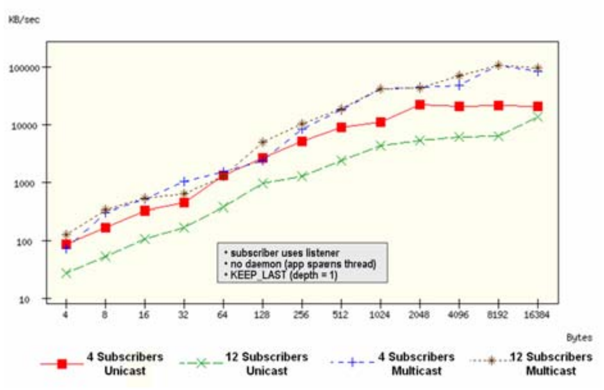

Figure 9: DDS1 Unicast vs Multicast

4.2.2 DDS2 Broadcast/Multicast

Results. Figure 10 shows the scalability test results for DDS2 broadcast/multicast with 1 publisher and multiple subscribers (*i.e.*, 4 and 12 blades). This figure shows that both multicast and broadcast scales well as the number of subscribers increases and multicast performs slightly better than broadcast.

Analysis. Figure 10 shows that sending messages to a specific group address rather than every subnet node is slightly more efficient. Moreover, using broadcast instead of multicast may be risky since sending messages to all blades can saturate network bandwidth and blade processors.

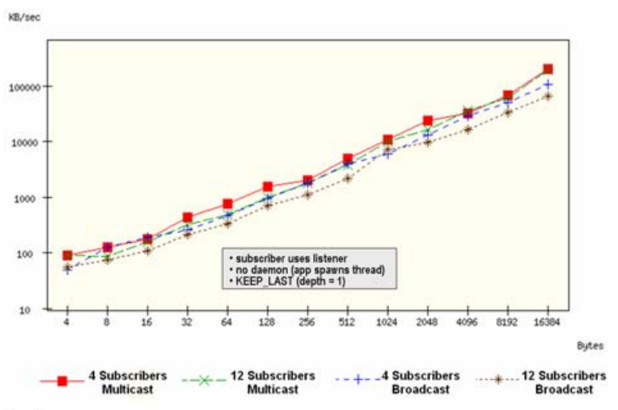

Figure 10: DDS2 Multicast vs Broadcast

4.2.3 Comparing DDS Implementation Performance

Results. Figure 11 shows multicast performance comparison of DDS1 and DDS2 with 1 publisher and 12 subscriber blades. Since DDS3 does not support multicast, we omit it from the comparison. Figure 12 shows unicast performance comparison of DDS1 and DDS3. Since DDS2 does not support unicast, we also omit it from this comparison.

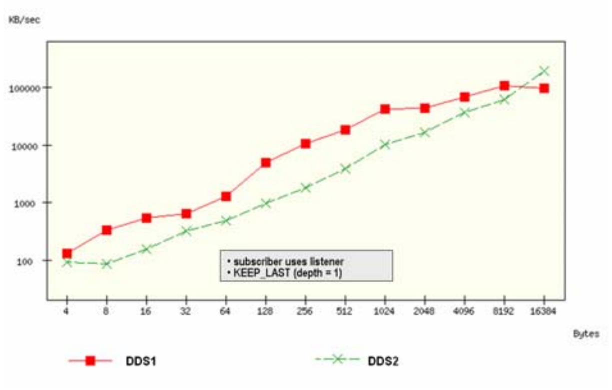

Figure 11: 1-12 DDS1 Multicast vs. DDS2 Multicast

Analysis. Figures 11 and 12 indicate that DDS1 outperforms DDS2 for smaller data sizes. As the size of the payloads increase, however, DDS2 performs better. It appears that the difference in the results stems from the different distribution architectures (decentralized and federated, respectively) used to implement DDS1 and DDS2.

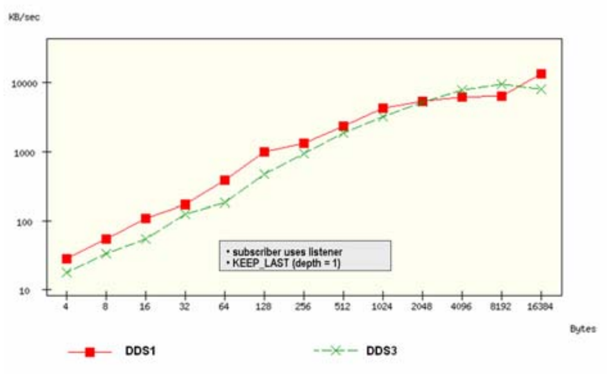

Figure 12: 1-12 DDS1 Unicast vs. DDS3 Unicast

5 Key Challenges and Lessons Learned

This section describes the challenges we encountered when conducting the experiments presented in Section 4 and summarizes the lessons learned from our efforts.

5.1 Resolving DBE Design and Execution Challenges

Challenge 1: Synchronizing Distributed Clocks

Problem. It is hard to precisely synchronize clocks among applications running on blades distributed throughout ISISlab. Even when using the Network Time Protocol (NTP), we still experienced differences in time that led to inconsistent results and forced us to constantly repeat the synchronization routines to ensure the time on different nodes was in sync. We therefore needed to avoid relying on synchronized clocks to measure latency, jitter, and throughput.

Solution. For our latency experiments, we have the subscriber send a minimal reply to the publisher, and use on the clock on the publisher side to calculate the roundtrip time. For throughput, we use the subscriber's clock to measure the time required to receive a designated number of samples. Both methods provide us with common reference points and minimize timing errors through the usage of effective latency and throughput calculations based on a single clock.

Challenge 2: Automating Test Execution

Problem. Achieving good coverage of a test space where parameters can vary in several orthogonal dimensions leads to a combinatorial explosion of test types and configurations. Manually running tests for each configuration and each middleware implementation on each node is tedious, error-prone, and time-consuming. The task of managing and organizing test results also grows exponentially along with the number of distinct test configuration combinations.

Solution. The DBE described in Section 3.2.2 stemmed from our efforts to manage the large number of tests and the associated volume of result data. Our efforts to streamline test creation, execution and analysis are ongoing, and include work on several fronts, including a hierarchy of scripts, several types of configuration files, and test code refactoring.

Challenge 3: Handling Packet Loss

Problem. Since our DDS implementations use the UDP transport, packets can be dropped at the publisher and/or subscriber side. We therefore needed to ensure that the subscribers get the designated number of samples despite packet loss.

Solution. One way to solve this problem is to have the publisher send the number of messages subscribers expect to receive and then to stop the timer when the publisher is done. The subscriber could then use only the number of messages that were actually received to calculate the throughput. However, this method has two drawbacks: (1) the publisher must send extra notification messages to stop the subscribers, but since subscribers may not to receive this notification message the measurement would never happen and (2) the publisher stops the timer, creating a distributed clock synchronization problem discussed in Challenge 1 that could affect the accuracy of the evaluation. To address these drawbacks we therefore adopted an alternative that ensures subscribers a deterministic number of messages by having the publishers "oversend" an appropriate amount of extra data.. With this method, we avoid extra "pingpong" communication between publishers and subscribers. More importantly, we can measure the time interval entirely at the subscriber side without relying on the publisher's clock. The downside of this method is that we had to conduct experiments to determine the appropriate amount of data to oversend.

Challenge 4: Ensuring Steady Communication State

Problem. Our benchmark applications must be in a steady state when collecting statistical data.

Solution. We send primer samples to "warm up" the applications before actually measuring the data. This warmup period allows time for possible discovery activity related to other subscribers to finish, and for any other first-time actions, on-demand actions, or lazy evaluations to be completed, so that their extra overhead does not affect the statistics calculations.

5.2 Summary of Lessons Learned

Based on our test results, experience developing the DBE, and numerous DDS experiments, we learned the following:

- **DDS Performs significantly better than other pub/sub implementations.** Figure 7 in Section 4.1 shows that even the slowest DDS was about twice as fast as non-DDS pub/sub services. Figure 7 and Figure 8 show that DDS pub/sub middleware scales better to larger payloads compared to non-DDS pub/sub middleware. This performance margin is due in part to the fact that DDS decouples the information intent from information exchange. In particular, XML-based pub/sub mechanisms, such as SOAP, are optimized for transmitting strings, whereas the data types we used for testing were sequences. GSOAP's poor performance with large payloads is due to the fact that GSOAP (de)marshals each element of a sequence, which may be as small as a single byte, while DDS implementations send and receive these data types as blocks.
- **Individual DDS architectures and implementations are optimized for different use cases**. Figures 7 and 11 show that DDS1's decentralized architecture is optimized for smaller payload sizes compared to DDS2's federated architecture. As payload size increases, especially for the complex date type in Figure 7, DDS2 catches up and surpasses DDS1 in performance on the same blade. When the publisher and subscriber run on different blades, however, DDS1 outperforms DDS for all tested data sizes.

- **Apples-to-apples comparisons of DDS implementations are hard**. The reasons for this difficulty fall into the following categories: (1) *no common transport protocol* – the DDS implementations that we investigated share no common application protocol, *e.g.*, DDS1 uses a RTPS-like protocol on top of UDP, DDS2 will add RTPS support soon, and DDS3 simply implements raw TCP and UDP, (2) *no universal support for unicast/broadcast/multicast* – Table 1 shows the different mechanisms supported by each DDS implementations, from which we can see DDS3 does not support any group communication transport, making it hard to maintain performance as the number of subscribers increases, (3) *DDS applications are not yet portable*, which stem partially from the fact that the specification is still evolving and vendors use proprietary techniques to fill the gaps (a portability wrapper façade would be a great help to any DDS application developer, and a huge help to our efforts in writing and running large numbers of benchmark tests), and (4) *arbitrary default settings for DDS implementations,* which includes network-specific parameters not covered by the DDS specifications that can significant impact performance.

6 Related Work

To support emerging DRE information management systems, pub/sub middleware in general, and DDS in particular, have attracted an increasing number of research efforts (such as COBEA [20] and Siena [12]) and commercial products and standards (such as JMS [10], WS_NOTIFICATION [13], and the CORBA Event and Notification services [17]). This section describes several projects that are related to our work presented in this paper.

Open Architecture Benchmark. Open Architecture Benchmark (OAB) [8] is a DDS benchmark effort along with Open Architecture Computing Environment, an open architecture initiated by the US Navy. Joint efforts have been conducted in OAB to evaluate DDS products, in particular RTI's NDDS and PrismTech's OpenSplice, to understand the ability of these DDS products to support the bounded latencies required by Naval systems. Their results indicate that both products perform quite well and meet the requirements of typical Naval systems. Our DDS work extends their effort by (1) comparing DDS with either other pub/sub middleware and (2) examining DDS throughput performance.

S-ToPSS. There has been an increasing demand for content-based pub/sub applications, where subscribers can use a query language to filter the available information and receive only a subset of the data that is of interest. Most solutions support only syntactic filtering, *i.e.*, matching the information based on the syntactic information, which greatly limits the selectivity of the information. In [7] the authors investigated how current pub/sub systems can be extended with semantic capabilities, and proposed a prototype of such middleware called the *Semantic - Toronto Publish/Subscribe System* (S-ToPSS). For a highly intelligent semantic-aware system, simple synonym transformation is not sufficient. S-ToPSS extends this model by adding another two layers to the semantic matching process, *concept hierarchy* and *matching functions*. Concept hierarchy makes sure that events (data messages, in the context of this paper) that contain generalized filtering information do not match the subscriptions with specialized filtering information, and similarly that events containing more specialized filtering than the subscriptions will match. Matching functions provide a many-to-many structure to specify more detailed matching relations, and can be extended to heterogeneous systems. DDS also provides QoS policies that support content-based filters for selective information subscription, but they are currently limited to syntactic match. Our future work will explore the possibility of introducing semantic architectures into DDS and evaluate their performance.

PADRES. The Publish/subscribe Applied to Distributed Resource Scheduling (PADRES) [1] is a distributed, content-based publish/subscribe messaging system. A PADRES system consists of a set of brokers connected by an overlay network. Each broker in the system employs a rule-based engine to route and match publish/subscribe messages, and is used for composite event detection. PADRES is intended for business process execution, and business activity monitoring, rather than for DRE systems. It does not follow DDS API but its publish/subscribe model is close to that of DDS, so we plan to explore how a DDS implementation might be based on PADRES.

7 Concluding Remarks

This paper first evaluated the architectures of three implementations of the OMG Data Distribution Service (DDS). We then presented the DDS Benchmarking Environment (DBE) and showed how we use the DBE to compare the performance of these DDS implementations, as well as non-DDS pub/sub platforms. Our results indicate that DDS performs significantly better than other pub/sub implementations for the following reasons: (1) DDS's communication model provides a range of QoS parameters that allow applications to control many aspects of data delivery in a network, (2) implementations can be optimized heavily, (3) DDS can be configured to leverage fast transports, *e.g.*, using shared memory to minimize data copies within a single node, and to improve scalability, *e.g.*, by using multicast to communicate between nodes.

As part of the ongoing Pollux project, we will continue to evaluate other interesting features of DDS needed by large-scale DRE information management systems. Our future work will include (1) tailoring our DBE benchmarks to explore key classes of applications in DRE information management systems, (2) devising generators that can emulate various workloads and use cases, (3) empirically evaluating a wider range of QoS configurations, *e.g.* durability, reliable vs. best-effort, and integration of durability, reliability and history depth, (4) designing mechanisms for migrating processing toward data sources, (5) measuring participant discovery time for various entities, (6) identifying scenarios that distinguish performance of QoS policies and features (*e.g.,* collocation applications), and (7) evaluating the suitability of DDS in heterogeneous dynamic environments, *e.g.,* mobile ad hoc networks, where system resources are limited and dynamic topology and domain participant changes are common.

References

1. A. Cheung, H. Jacobsen, "Dynamic Load Balancing in Distributed Content-based Publish/Subscribe," ACM/IFIP/USENIX Middleware 2006, December, Melborne, Australia.
2. D. C. Schmidt and C. O'Ryan, "Patterns and Performance of Distributed Real-time and Embedded Publisher/Subscriber Architectures," Journal of Systems and Software, Special Issue on Software Architecture -- Engineering Quality Attributes, edited by Jan Bosch and Lars Lundberg, Oct 2002.
3. C. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and d. C. Schmidt, "Integrated Adaptive QoS Management in Middleware: An Empirical Case Study," Proceedings of the 10th Real-time Technology and Application Symposium, May 25-28, 2004, Toronto, CA.
4. G. Pardo-Castellote, "DDS Spec Outfits Publish-Subscribe Technology for GIG," COTS Journal, April 2005.
5. M. Balakrishnan, K. Birman, A. Phanishayee, and Stefan Pleisch, "Ricochet: Low-Latency Multicast for Scalable Time-Critical Services," Cornell University Technical Report, www.cs.cornell.edu/ projects/quicksilver/ public_pdfs/ricochet.pdf.
6. OMG, "Data Distribution Service for Real-Time Systems Specification," www.omg.org/docs/formal/04-12-02.pdf.
7. I. Burcea, M. Petrovic, H. Jacobsen, "S-ToPSS: Semantic Toronto Publish/Subscribe System," International Conference on Very Large Databases (VLDB). p. 1101-1104. Berlin, Germany, 2003.
8. B. McCormick, L. Madden, "Open Architecture Publish-Subscribe Benchmarking," OMG Real-Time Embedded System Work Shop 2005, www.omg.org/news/meetings/workshops/RT_2005/03-3_McCormick-Madden.pdf.
9. A. S. Krishna, D. C. Schmidt, R. Klefstad, and A. Corsaro, "Real-time CORBA Middleware," in *Middleware for Communications*, edited by Qusay Mahmoud, Wiley and Sons, New York, 2003.
10. Hapner, M., Burridge, R., Sharma, R., Fialli, J., and Stout, K. 2002. Java Message Service. Sun Microsystems Inc., Santa Clara, CA.
11. Fox, G., Ho,A., Pallickara, S., Pierce, M., and Wu,W, "Grids for the GiG and Real Time Simulations," Proceedings of Ninth IEEE International Symposium DS-RT 2005 on Distributed Simulation and Real Time Applications, 2005.
12. D.S.Rosenblum, A.L.Wolf, "A Design Framework for Internet-Scale Event Observation and Notification," *6th European Software Engineering Conference*. Lecture Notes in Computer Science 1301, Springer, Berlin, 1997, pages 344-360.
13. S. Pallickara, G. Fox, "An Analysis of Notification Related Specifications for Web/Grid Applications," International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II pp. 762-763.
14. Real-Time Innovation, "High-reliability Communication Infrastructure for Transportation," www.rti.com/markets/transportation.html.
15. Real-Time Innovation, "High-Performance Distributed Control Applications over Standard IP Networks," www.rti.com/markets/industrial_automation.html.
16. Real-Time Innovation, "Unmanned Georgia Tech Helicopter files with NDDS," controls.ae.gatech.edu/gtar/2000review/rtindds.pdf.
17. P. Gore, D. C. Schmidt, C. Gill, and I. Pyarali, "The Design and Performance of a Real-time Notification Service," Proceedings of the 10th IEEE Real-time Technology and Application Symposium (RTAS '04), Toronto, CA, May 2004.
18. N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju, "Differential Serialization for Optimized SOAP Performance**,**" Proceedings of *HPDC-13: IEEE International Symposium on High Performance Distributed Computing*, Honolulu, Hawaii, pp. 55-64, June 2004.
19. Distributed Object Computing Group, "DDS Benchmark Project," www.dre.vanderbilt.edu/DDS.
20. C.Ma and J.Bacon "COBEA: A CORBA-Based Event Architecture," In Proceedings of the 4rd Conference on Object-Oriented Technologies and Systems, USENIX, Apr.1998
21. P. Gore, R. K. Cytron, D. C. Schmidt, and C. O'Ryan, "Designing and Optimizing a Scalable CORBA Notification Service," in Proceedings of the Workshop on Optimization of Middleware and Distributed Systems, (Snowbird, Utah), pp. 196–204, ACM SIGPLAN, June 2001.