PrismTech Data Distribution Service Java API Evaluation

Cortney R. Riggs*
Broadus, MT 59718

My internship duties with Launch Control Systems required me to start performance testing of an Object Management Group's (OMG) Data Distribution Service (DDS) specification implementation by PrismTech Limited through the Java programming language application programming interface (API). DDS is a networking middleware for Real-Time Data Distribution. The performance testing involves latency, redundant publishers, extended duration, redundant failover, and read performance. Time constraints allowed only for a data throughput test. I have designed the testing applications to perform all performance tests when time is allowed. Performance evaluation data such as megabits per second and central processing unit (CPU) time consumption were not easily attainable through the Java programming language; they required new methods and classes created in the test applications. Evaluation of this product showed the rate that data can be sent across the network. Performance rates are better on Linux platforms than AIX and Sun platforms. Compared to previous C++ programming language API, the performance evaluation also shows the language differences for the implementation. The Java API of the DDS has a lower throughput performance than the C++ API.

I. Introduction

As an intern with Launch Control Systems, I have spent most of my time working on a software evaluation project. This opportunity has brought me from Montana State University to the Kennedy Space Center in Florida. I am working in the Engineering Directorate under the System Software Engineering Branch of the Control and Data Systems Division, or NE-C3. The software evaluation that I am performing is to aide Common Services in the development and implementation of the command and message networking for the Constellation Program's Aries launches. Also, the code development done for the Java programming language API test applications will provide a starting point for actual implementation of the product. The performance evaluation was a large enough project that it required most of my time during the internship.

The majority of this paper evaluates the Java programming language API of a Commercial off the Shelf (COTS) implementation of a Data Distribution Service (DDS) specification by the Object Management Group (OMG). The goal of the evaluation was to determine the performance of the PrismTech OpenSplice™ DDS product in the Java API compared to the C++ API. The results provide a benchmark point for the integration team of Launch Control System Software to take into consideration as the implementation of DDS develops.

A. DDS Overview

Data Distribution Service is a specification by OMG for Real-Time Networking[3]. The service is meant to be a middle-ware product. Topics are associated with data to be sent. Several instances of a topic can be created with different keys and many samples of these instances can be written or read by the middle-ware publishers and subscribers respectively. Data writers and readers are attached to applications to do the desired publishing and subscribing workload. The service uses shared memory access of nodes on the network within a created domain. Configuration of the service such as, domain database size and partitioning, is done through an extensible markup language (XML) file. Different domains for separate purposes can be created within the same network. Thus, allowing multiple applications at a node. DDS is meant to be a Real-Time information networking specification.

B. Evaluation Scope

PrismTech Ltd's OpenSplice™ COTS DDS implementation was tested. The version 3.4 release was used. A custom Java application developed during October and November of 2008 did the desired testing. C++ testing had already been performed in January 2008 and July 2008.

* Undergraduate Intern, Launch Control Systems, Kennedy Space Center from Montana State University

Performance testing was compared on several platforms. Due to issues encountered in C++ API testing[1], IBM AIX servers and Linux server platforms were used to do relevant runs of the application. However, some tests were conducted with the Sun Solaris server platform for confirmation purposes.

At the time of this document, implementations had only been evaluated for point-to-point throughput performance. However, the testing applications had already been developed to evaluate the following:

1) Many-to-Many throughput performance
2) Latency performance
3) Redundant publisher throughput performance
4) Extended test duration

While, application add-ons are in various levels of development to perform the following:

1) Redundant Publisher failover
2) Read performance

Also, other aspects of the tests that had been used in evaluation of the C++ API had been ignored due to troubles acquiring platform and process details with the Java application. Thus, memory and central processing unit (CPU) usage of the application and service is left out for IBM AIX tests. Also, implementation of strength aware data writers was not complete. As with the C++ tests[1], data persistency and DDS partitions had not been evaluated.

II. Test Set-Up

Several test variables were controlled or utilized to perform the evaluation. Operating system platform, DDS topics, working applications, and DDS Quality of Service (QoS) policies associated with publisher redundancy were all considered for each type of test. With different combinations of these variables a full evaluation of the OpenSplice™ COTS product could be performed.

A. Platforms

Three platforms were available for the evaluation. In previous C++ API testing, issues with two platforms had been encountered. These issues influenced the way evaluation tests were performed for the Java API.

According to the COTS DDS Evaluation report[1], IBM AIX confirmed a kernel issue as the cause of occasional lock up when COTS products neared their performance thresholds in C++ API evaluation. I am not sure what the resolution was at the time of Java API evaluation. Also, effects of the resolution were not considered since there was no previous testing of the Java API. Tests performed in this evaluation on the AIX partition were performed without occurrences of the issue. AIX platform tests include application and DDS CPU and memory utilization from the AIX command “ps”.

As stated previously, the Sun Solaris platform showed CPU utilization inconsistency in the C++ API evaluation. In my initial test application debugging and running on this platform, inconsistent results were observed in publishing rate and duplicate packets read. At the time, CPU utilization was not measured efficiently through the Java applications, but the duplicate packet trend indicated similar utilization problems. Point to point throughput on the same node was the only testing done on this platform.

Linux platform servers were also available for evaluation. Due to a license file issue, only one Linux server was used. No prior issues had been found in utilization of this platform by Scott Cummins previous testing[1]. Also, all tests performed on Linux partitions include test duration CPU and memory utilization of the applications and services because measurement could be obtained by the Java application via /proc/<pid>/stat file.

Significant evaluation testing was performed or planned to be performed on the AIX and Linux platforms.

B. Topics

Topics for the Java API evaluation were the same as topics used in the C++ API evaluation. Four were utilized for the various performance tests. The throughput topic was used to publish Measurement messages to all subscribers as proof of concept for the LCS. The command topic, implemented only for latency performance tests, simulated low rate commanding from an application server to a gateway. Three control commands of the control topic were sent from publishers to subscribers: START informed that the publisher will begin publishing messages, STOP informed that the publisher has stopped publishing messages, and TERMINATE informed all publishers and subscribers to terminate the test. The last topic, ready topic, sent a status from subscribers to publishers: READY informed that the subscriber is ready to receive messages and LATENCY_CHECK sent time statistics back to publishers for latency check messages received.

Related to the throughput topic, an informative value was the megabits per second of throughput data. However, measurement of an object size in the Java programming language is not a provided concept. A rough estimate was

achieved for the Measurement message object by averaging heap size change after multiple creations of new Measurement message objects. Using these values, the rates on different platforms could be compared relative to the actual amount of data being delivered.

C. Application Wrapper

To simulate the end design goal, a wrapper package was designed to contain the test application Java API implementation. Several classes and objects were used in the wrapper package. The main purpose of the package was to create applications to use the service with desired properties. For the purposes of the evaluation, the package also included a couple class objects to obtain test information—which were also troublesome to implement and described in section E. The package was not fully implemented to some concepts used for earlier testing, but the general concept was achieved.

Quality of service properties for DDS define the behavior of the service. With arguments provided from the implementing applications and an extensible markup language (XML) file—different purpose from the configuration XML file—the wrapper sets the needed QoS policies to achieve the desired behavior. Any service entities created in the application will have the desired QoS settings.

D. Applications

Two applications were created to do a majority of the testing, DDSPub and DDSSub. The DDSPub application was the main publisher application. DDSSub was the main subscriber application. I implemented various publishing styles to publish at a targeted rate in the DDSPub. An XML file was parsed by the applications to specify QoS settings. Also, latency statistics and synchronization of multiple subscribers and publishers was supported by the applications.

Initially, the DDSPub application had several algorithms to attempt to maintain a desired rate of publishing. Also, an option for constant publishing over the duration was implemented. This was used to compare to the performance threshold to the maximum publishing rates. In comparisons of the actual performance of the algorithms, the most efficient was kept. Thus, as with the publisher in the C++ application, an interval timer performed the tasks necessary.

Publisher and subscriber synchronization was needed to ensure messages were sent or reads were taken when all applications were ready. Here also, the Java applications followed the same pattern as the previous C++ applications. First, application command line options were used to specify to the subscribers how many publishers and vice versa. Second, listeners were attached to the Control topic by each subscriber to wait for liveliness assertion by the publishers on this topic. The Throughput topic was not used, because publishers needed to register the throughput message instances and attach a listener to the Ready topic before asserting liveliness. Once it determines all publishers have asserted liveliness, a subscriber loops through the following until a TERMINATE command is received:

1) Publish a READY status on Ready topic
2) Wait for a START command on the Control topic
3) Save the start time
4) Read data until a STOP command is received
5) Save the stop time
6) Delay until no new data samples have been received over a 3 second period
7) Calculate and report statistics

A publisher also loops. The publisher will continue to do the following until a set number of loops have been run or it detects a previously obtained maximum throughput rate at the subscribers has not been met on 3 consecutive loops:

1) Wait for the specified number of subscribers to send a READY status on the Ready topic
2) Save the start time
3) Send a START command on the Control topic
4) Publish samples at the current test rate for the specified duration
5) Save the stop time
6) Send a STOP command on the Control topic
7) Calculate and report statistics
8) Increment Test Rate

Two other applications used in C++ testing had not been developed in Java yet. The purposes of these applications were to assert liveliness change on a publisher after a shutdown or kill and provide statistics on DDS instance lookups and queries. Many of the performance tests could be performed without these applications.

E. Application Programming Challenges

When creating the applications for the evaluation testing, there were several challenging problems to try to overcome for evaluation data. One I believe is significant accomplishment is the measuring of object size in Java. This was a task that I tried to solve in between preliminary runs of the applications.

Measuring object size is not native to the Java language like it is in C++. With some understanding of the Java Virtual Machine and its heap stack, I attempted to construct a class that could calculate the approximate byte size of any object. While I attempted to measure the average increase in heap size from multiple creations of new objects passed to a function, I did not seem to get feasible results. With the help of an aggressive garbage collection found in an open source forum about the same problem, I was able to improve my concept to have a possibly large estimate of object size.

Also, retrieving process statistics from through the Java language proved challenging. The InputStream classes of the Java framework did not seem to be able to parse the byte information on UNIX platforms, while they were able to read the Linux byte information just fine. Due to this, I had to find other platform specific methods for AIX and Sun process statistics. However, the methods I found do not give exact desired results—i.e. overall CPU% utilization rather than since last read CPU% utilization.

III. Test Descriptions

Most test designs from the C++ API evaluations can be performed with the applications created. At the time of this document only one DDS performance test had been completed. This was the point-to-point throughput test. Since I have not had much experience with testing software, another test was done first in order to find the most efficient method to publish at regular intervals.

A. Publishing Efficiency Test

Several same node point-to-point throughput tests were run with different algorithms to publish the message at regular intervals. The tests involved a single subscriber and a single publisher. The tests were performed with 1 topic instance, QoS setting of Reliable delivery, and a listener subscriber read method implementing the DDS Listener interface (actually this is test case ‘a’ in the point-to-point throughput tests). The five algorithms for comparison did the following:

1) Timed Task publish – this called the publishing function of the DDSPub application on the set interval.
2) Timed Task Counting Semaphore – this released permits of a semaphore on the set interval. The DDSPub application had to acquire a permit before each publish.
3) Timed Task Limited Semaphore – this performed the same as the Counting Semaphore algorithm, but only one permit could be available at a time and tardiness of publishes from the interval were counted.
4) While Loop publish – each loop would check the current time to the interval time and publish if required.
5) Max publish – this just looped on a publish method for the duration specified at the incremented messages per update rate.

The Max publish case calculated an initial packets per update from arbitrary setting of publication rate. This was then incremented by 10 packets per update every 60 seconds. For the first four cases, a low publication rate was used by the publisher, then incremented by 1000 samples per second every 60 seconds until a peak rate was reached. These peak rates were compared to each other and the Max publish case. The Max publish and the method from algorithms 1-4 that produced the best publication rate performance were the only two kept implemented in the DDSPub application.

The Publishing Efficiency testing was done during initial runs of the test applications. These were performed on an available Sun Solaris platform server. Tests were run several times and only deemed relevant if duplication of message packets was not substantial to account for the noticed inconsistency in the platform.

B. Point-to-Point Throughput

A series of point-to-point throughput tests were run utilizing a single publisher and subscriber with different combinations of QoS settings, number of topic instances, and read methods at the subscriber. The combinations

were exactly like the C++ API performance point-to-point throughput testing. The three QoS setting/topic instance combinations were:

1) 1 topic instance with Reliable delivery – this combination best reflects a topic used to publish commands and events.
2) 70,000 topic instances with Reliable delivery – this combination best reflects a topic used to publish measurement data.
3) 70,000 topic instances with Best Effort delivery – this combination was simply used to obtain a data point relative to throughput performance for measurement data using best effort delivery.

The three method of reading data were:

1) Listener – this method implements a DDS Listener interface. A listener method is invoked within a COTS/DDS thread when a new instance sample is received.
2) Polling – this method repeatedly sleeps for the specified time then reads all new instance samples that have been received. A sleep time of 10-15 ms was used for this evaluation.
3) WaitSet – this method implements the DDS WaitSet construct. This construct allows an application thread to block until the attached ReadCondition is satisfied. In this case a ReadCondition specifying any new instance sample was used.

The point-to-point throughput test case combinations used are outlined in Table 1 below and are referred to as test cases a-i throughout this document.

Test Case	Topic Instances	Reliability	Read method
a	1	Reliable	Listener
b	70,000	Reliable	Listener
c	70,000	Best Effort	Listener
d	1	Reliable	Polling
e	70,000	Reliable	Polling
f	70,000	Best Effort	Polling
g	1	Reliable	WaitSet
h	70,000	Reliable	WaitSet
i	70,000	Best Effort	WaitSet

Table 1. Point-to-Point Throughput Test Cases. *The combinations of number of Topic Instances, delivery Reliability, and Subscriber Read Method for the nine test cases a-i.*

Similar to the publishing efficiency testing, the publisher in each test case initially used a publication rate known to be below the peak, and then incrementally increased the rate by 1000 samples/second every 60 seconds until the peak rate was determined.

At completion of the point-to-point throughput tests, the 70,000 topic instance/reliable subscriber read method that provided the best throughput performance was the ideal scenario that would be used for the continued testing.

C. Test Configurations

Table 2[†] shows the different test configurations utilized for the two test implementations on each platform. P1 refers to the publisher and S1 refers to the subscriber[‡].

	Linux Nodes		AIX Nodes		Solaris Nodes	
Test	A	B	A	B	A	B
Code Efficiency					P1/S1	
Same node Point-to-Point a-i	P1/S1				P1/S1	
Two nodes Point-to-Point a-i			S1	P1		

Table 2. Node Configurations of Tests. *Set up of the test applications on the available nodes.*

[†] Node representations in Table 2 are arbitrary letters signifying separate physical servers on a network.

[‡] Numbered notation of publisher and subscriber are for ease of future updates with multiple publishers and subscribers in other tests.

IV. Test Results

A. Publishing Efficiency

Table 3 shows the top rates obtained for each of the efficiency tests. There were significant differences in the top rates. However, the two semaphore using algorithms have approximately the same performance.

Publish Style	Goal Rate at Max	Samples/s	Mbits/s	Average ms/update
Sem. Count	35,000	28,599	13.5	12
Sem. Limit	41,000	28,592	13.5	14
Interval Pub.	27,000	26,442	12.5	10
Looped Pub.	22,000	20,259	9.6	8
Max Pub.		28,310	13.4	

Table 3. Publishing Rates Achieved. *Top rates for each style of publishing.*

The two semaphore algorithms reached their top rates when trying to send data at significantly higher rates. The Timed Task Counting Semaphore algorithm achieved throughput at the goal rate for each increment through 28,000 samples/s, while the Timed Task Limited Semaphore algorithm's throughput strayed from the goal at 24,000. Also, the Counting Semaphore's rate stayed relatively close to the peak for the remainder of the test.

The While Loop publishing method achieved the worst performance. A top rate was achieved before the average ms/update (8 ms/update) had reached the update interval of 10 milliseconds. During most of the test, the rates for each 60 second duration were hundreds of packets per second away from the goal throughput rate. This would be an unsatisfactory performance.

The Max publish algorithm test was only performed for a duration of 7 minutes, or 7 test durations, because the samples per second were consistently above 28,000 with small variation.

Interestingly, the samples per second for the subscriber application did not always match the publisher application. Moreover, the subscriber seemed to report higher rates than the publisher as often as reports of lower rates. The differences in reported rates were small, less than 10, and usually within one or two samples/s.

B. Point-to-Point Throughput

Table 4 shows the test cases that obtained the highest throughput on each platform and Table 5 shows the comparison two the top rates from previous C++ API testing[1]. The Sun and Linux tests were run with both applications on the same node, while the AIX tests used two nodes. Tests on the AIX platform ran without any interruptions seen on the other two platforms. The three platforms had varying results. However, the top performance for a 70,000 instance/Reliable delivery method test case was 'e' on the Linux and AIX platforms.

Platform	Test Case	Samples/s	Message Bytes	Mbits/s
Sun	f	47,564	59	22.4
Linux	d	67,503	60	32.4
AIX	d	38,156	48	14.7

Table 4. Top Rates. *Top rates for each platform.*

Platform	API	Samples/s
Linux	Java	67,503
	C++	84,000
AIX	Java	38,156
	C++	50,800

Table 5. Language Top Rates. *Top rates for each programming language API.*

CPU% comparisons across platforms were not relevant. The methods for each platform varied and were platform specific. Details of the ways that the methods reported CPU% and other process statistics were not the same.

Rates on the Sun platform had inconsistent results. Throughput would be poor with many duplicate messages for extended periods and very slow average milliseconds per update—15 ms and greater for a 10 ms interval. Most of the time, the test would not be able to continue. Occasionally, this trend would stop during the test and the test would continue on with efficient throughputs for much higher test rates. Also, the subscriber samples per second rate would have a unique value in these cases—usually around 27,000 samples/s—regardless of the current test rate goal.

Generally, the Linux platform had higher maximum performance rates. However, the subscriber would terminate with no signal sent by the publisher on occasion. This would occur while the throughput rate was matching the desired test rate. Early terminations on happened with the Listener read method. The cause of the early terminations was not discovered.

WaitSet read methods on the Linux platform started at poor average update times then improved as the rate incremented to higher values. The service seemed to need to warm-up for these tests. I am unsure if this is a

platform issue. The same tests on the other platforms started too close to their threshold to allow time for this to happen or started above their threshold. Time did not allow for further insight into this event.

The full test results for point-to-point throughput are included in the appendix. The Java implementation of the DDS has lower peak performance rates. Also, the each test case had lower top rates for the Java API than for the C++ API.

C. Other Findings

Other results that were unexpected came from running the tests. First, the OpenSplice™ DDS take method had to pass a max number of sample to take argument of unlimited. The method takes a created Topic type sequence holder and a DDS SampleInfoSeqHolder along with some read masks and a max number of samples to take. According to the OpenSplice™ DDS Java Reference Manual, the max samples parameter has to be less than or equal to the length of the Topic type or be set to unlimited. When the special unlimited case is not used, the method will fail this condition even when it is met.

Second, on the Sun platform, manual command line killing of the application sometimes did not shut down the applications. The poor publishing and duplicate packets received by the subscriber problem always accompanied the observation. This could be due to the signal handling conflict of the OpenSplice™ DDS and the Java Virtual Machine. However, the conflict would seem to cause the service to shutdown instead of the application, which was not the case in these observations.

V. Test Summary

The performance test of the Java API for PrismTech Ltd's OpenSplice™ Data Distribution Service provided useful results. The relative speed of the networking service on platforms was discovered. When compared to the C++ API for the service, the Java API implementation is less efficient.

Across platforms, speeds of the DDS varied. The Linux platform provided the fastest throughput rate. The top rate for Linux was twenty to thirty thousand samples per second faster than the Sun and AIX top rates. This difference was the same on C++ API testing.

While the patterns of the top throughput rate were the same in both programming language API's, the speed of the languages differed. Earlier tests[1] returned higher throughput rates than the Java API implementation. The C++ API implementation of the OpenSplice™ Data Distribution Service carries data across a network more efficiently in point-to-point scenarios. Since development and integration will include the Java API, as well as the C++ API, the lower performance rate will need to be addressed if it is below an expected threshold for the end networking product.

VI. Conclusion

As an intern with NASA through the Undergraduate Student Research Program, I was involved with an exciting project. Most of this paper has been based around the project and work I did. With the help of my mentor and others, I was able to apply my classroom lessons to a real world application. I had never done software testing before, so this was a new challenge.

I was able to write a test application for a networking service focused on Real-Time Data Distribution. The networking middleware, PrismTech Ltd's OpenSplice Data Distribution Service, was an implementation of the Object Management Group's specifications for DDS. I believe I have made a good starting assessment of the Java API for the middleware. That assessment has shown the performance of the service over various platforms and made a comparison between the two programming language API's. I have gained much from the work.

Appendix

Test Results

		Test Case							
Statistic	**Platform**	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
Top Rate (Samples/s)	Sun	28,475	27,775	*	*	22,105	47,564	*	*	*
	Linux	47,997[#]	45,397	43,863	67,504	62,429	62,307	49,623	40,907	34,997
	AIX	27,143	24,360	22,597[+]	38,377	35,032	33,334[+]	27,266	23,776	22,564[+]
Mbits/s	Sun	13.4	13.1	*	*	10.4	22.5	*	*	*
	Linux	23.0	21.8	21.1	32.4	30.0	29.9	23.8	19.6	16.8
	AIX	10.4	9.4	8.8	14.7	13.5	12.8	10.5	9.1	8.7
Sub. App. CPU%	Sun	--	--	--	--	--	--	--	--	--
	Linux	--	94.9	--	--	--	0.3	--	--	--
	AIX	--	12.5	11.8	7.0	6.7	6.6	13.6	14.8	13.5
Sub. Serv. CPU%	Sun	--	--	--	--	--	--	--	--	--
	Linux	--	0.0	--	--	--	0.0	--	--	--
	AIX	--	13.6	12.7	18.7	15.1	15.0	15.2	14.6	12.8
Pub. App. CPU%	Sun	--	--	--	--	--	--	--	--	--
	Linux	--	77.3	--	--	--	99.7	--	--	--
	AIX	--	7.0	18.2	11.0	8.6	8.6	8.5	7.2	10.0
Pub. Serv. CPU%	Sun	--	--	--	--	--	--	--	--	--
	Linux	--	0.0	--	--	--	0.0	--	--	--
	AIX	--	3.8	3.2	7.6	6.2	7.6	4.0	3.6	9.2

Point-to-Point Throughput Performance Test Results. This table shows the overall results for the Java API point-to-point throughput performance testing. CPU% measurements with dashes (--) are from testing that did not have the process statistics working properly. CPU% measurements should not be compared across platform, because methods measured the CPU usage against different time scales.

* Contradicting values were given by the subscriber and publisher applications. Usually the subscriber reported rates in the 20,000 to 30,000 samples/s range and the publisher barely above 10,000 for all cases. Note, this only occurred on the Sun platform.

Tests prematurely terminated in all attempts for this test. The max continuous publish algorithm was used to gain this value instead.

+ On AIX platforms, all three Best Effort scenarios always continued to publish on to much higher rates (up to 52,000). The subscriber only received messages at the rates shown.

Acknowledgments

C Riggs thanks the Universities Space Research Association for the opportunity and financial support to intern with NASA at the Kennedy Space Center through their Undergraduate Student Research Program. He would also like to thank his mentor, Jennifer Boelke, for guidance and help with the process. Mr. Riggs thanks Chad Chamberlin, Scott Cummins, and Chris Shuler for their insight and patience with the work.

References

[1]Cummins, S. R., "COTS Data distribution service (DDS) Evaluation, June-July 2008," NASA CxP Draft, Kennedy Space Center, FL, Aug. 12, 2008.

[2]Eckel, B. *Thinking in Java*, 3rd ed., Prentice Hall PTR, Upper Saddle River, NJ, 2003

[3]Object Management Group, *DataDistribution Service for Real-time Systems* [online document], Ver. 1.2, URL: http://www.omg.org/spec/DDS/1.2/PDF/ [cited 3 Dec. 2008].

[4]OpenSplice Data Distribution Service, Software Package, Ver. 3.4, PrismTech Ltd, Netherlands, 2008.