Performance testing open source products for the TMT event service

K. Gillies*,a, Yogesh Bhateb

*aThirty Meter Telescope Observatory Corp., 1111 Arroyo Parkway, Pasadena, CA, USA 95555-0345; bPersistent Systems, Ltd., Bhageerath, 402 Senapati Bapat Road, Pune 411 016, India

ABSTRACT

The software system for TMT is a distributed system with many components on many computers. Each component integrates with the overall system using a set of software services. The Event Service is a publish-subscribe message system that allows the distribution of demands and other events. The performance requirements for the Event Service are demanding with a goal of over 60 thousand events/second. This service is critical to the success of the TMT software architecture; therefore, a project was started to survey the open source and commercial market for viable software products. A trade study led to the selection of five products for thorough testing using a specially constructed computer/network configuration and test suite. The best performing product was chosen as the basis of a prototype Event Service implementation. This paper describes the process and performance tests conducted by Persistent Systems that led to the selection of the product for the prototype Event Service.

Keywords: middleware, software design, performance testing

1. INTRODUCTION

From a software communications and integration viewpoint, the TMT software system consists of a set of software components interacting with each other through a software communications backbone and software infrastructure (middleware). The individual software components have a wide range of purposes within the software system. Some will be standalone, software-only applications; others will be software components within complex software/hardware subsystems. Figure 1 shows the TMT subsystems that rely upon and the TMT communications backbone at the telescope site.

Figure 1: The green bar represents TMT Common Software. All TMT components use the services that are part of Common Software.

*kgillies@tmt.org; phone: 1-626-395-1655; http://tmt.org
The integration of all these software components requires software infrastructure that is outside the scope of the individual components. The design of the TMT software system has resulted in a small set of services and associated software called TMT Common Software that is focused on the task of integrating software components. The connections shown between TMT subsystems in Figure 1 correspond to service connections. The idea of project-provided shared software based on a set of services and associated software is a successful strategy in large observatory software architecture[1].

The functionality of the integration services is not unique to astronomy. Several examples of each service can be found as open source or commercial products. The challenge for TMT is to integrate these services into a cohesive layer and to isolate them in order to allow replacement over the telescope’s 50 year lifecycle while minimizing the impact on the software system built upon them.

1.1 Event Service

The Event Service is undoubtedly the most important service provided by TMT Common Software. It allows one component to send a piece of information (i.e., an event) to one or more other components. The service is built using a high-performance, publish-subscribe message system. With the publish-subscribe pattern, components publish their information whenever necessary. Other components express interest by subscribing to the information and are then notified when new information is published. Subscribers only receive updates to information for which they have subscribed and are updated only when a value is published.

The advantage of this type of message system is that publishers and subscribers are decoupled. Publishers can publish regardless of whether there are subscribers, and subscribers can subscribe even if there are no publishers. The relationship between publishers and subscribers can be one-to-one, one-to-many, many to one, or even many-to-many. Components and systems can startup and stop without requiring the notification of other systems.

Subscribing can be topic-based or content-based. With a topic-based system, a component subscribes to events by a name or topic. For example, the telescope control system might publish a tcs.elevation topic. Instruments could subscribe to that topic to be notified of changes to the telescope elevation. With a content-based system a subscriber is notified only when conditions on event attributes are matched. For example, an alarm GUI might alert the observer if mobie.detector.temperature > 20. Both subscription approaches can be useful for building the system.

High performance is essential for the Event Service implementation. TMT requirements specify 20,000, 60 byte events/second, but realistically a rate of 50,000-70,000 events/second is a more reasonable requirement. Because of the importance of the Event Service and the concern for performance, a project was started to identify potential implementation products.

2. PROJECT PLAN

The project plan was created with three phases. The first phase consisted of updating a set of product requirements and creating a set of criteria for comparing potential candidate products. The set of potential products was examined and the set was reduced based on a set of general criteria. There are at least 30-50 products that could be tested, but due to time and money it was necessary to trim the list to the best candidates. A trade study was performed based on the set of comparison criteria. The result was a list of products for testing during Phase 2.

During phase 2 each of the products was subjected to a set of performance benchmarks designed to measure the overall product performance in scenarios that are representative of TMT Event Service scenarios. The final result of Phase 2 was a report and a recommendation for a prototype implementation of the Event Service during phase 3. The focus of this report is the performance benchmarking completed during phase 2.

The TMT offices in India and the United States collaborated to define a 6 month project based on these three phases. The India Institute of Astronomy conducted a competitive process to identify an India software vendor for the project. The contract was won by Persistent Systems, Ltd of Pune, India, which completed all three phases within the time scheduled.

3. PRODUCT ANALYSIS

The product analysis process is shown in Figure 2. Based on requirements, the project team merged a provided preliminary list of products with a list of products discovered from market research and internal experience. The initial
unfiltered list included 33 products, which included a wide range of technology solutions. The goal of phase 1 was to recommend a list of 5 products for comprehensive benchmarking so it was necessary to reduce the product list.

A small set of filter criteria consisting of 5 necessary requirements were used to do the initial list reduction. If a product did not satisfy all the 5 criteria, the product was eliminated. Following the filtering, the list of 33 was reduced to 12. The 12 products were evaluated in a more comprehensive way in a trade study based on 31 criteria in 2 categories. The core category included criteria that were deemed essential. The additional category included criteria that were desirable, but not essential. The core criteria were more heavily weighted than the criteria in the additional category. For instance, products already claiming to support 50 thousand messages/second was a core criteria, but a product’s current use in astronomy was additional.

The products selected for comprehensive benchmarking in phase 2 are shown in Table 1. The set of products includes Data Distribution Service (DDS) products that are currently in use at telescope facilities. A wide range of technologies are covered including peer to peer implementations and implementations using a centralized broker.

Table 1: Products identified for comprehensive benchmarking during phase 2 of the project.

<table>
<thead>
<tr>
<th>Product</th>
<th>Description</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>HornetQ 2.3.0</td>
<td>Open source high performance, asynchronous message system</td>
<td><a href="http://hornetq.jboss.org">http://hornetq.jboss.org</a></td>
</tr>
<tr>
<td>RTI Connext DDS 5.0.0</td>
<td>Commercial Data Distribution Service</td>
<td><a href="http://www.rti.com">http://www.rti.com</a></td>
</tr>
<tr>
<td>Redis 2.6.16</td>
<td>Advanced key-value store with publish/subscribe</td>
<td><a href="http://redis.io">http://redis.io</a></td>
</tr>
<tr>
<td>Open Splice DDS 6.3.1</td>
<td>Open source Data Distribution Service</td>
<td><a href="http://www.prismtech.com">http://www.prismtech.com</a></td>
</tr>
</tbody>
</table>

HornetQ, Redis and Redhat MRG are broker-based products. This means that all clients send their messages to a centralized server program (1 or more if replicated). The broker server keeps track of which clients are subscribing to which topics. The broker server manages the delivery of messages to subscribing clients. RTI Connext DDS and Open Splice DDS are peer-to-peer products. This means there is no central broker, the infrastructure that is part of each participant keeps track of subscriptions and delivery.

4. BENCHMARKING TEST SUITE

The benchmarking was planned and executed by the performance testing group at Persistent Systems, Ltd in Pune, India. Together, a list of tests was devised to evaluate the throughput, latency and jitter of each product using a few scenarios that simulate TMT situations. During each test, system parameters were measured including: memory usage, disk I/O, CPU usage and network utilization. Any message loss was measured during each test along with a check that messages retained ordering. All tests were performed on 1 GbE and 10 GbE networks. The 10 GbE tests were particularly
interesting and informative since this is the baseline network speed for TMT. Latency tests were also run on the Redhat MRG real-time kernel to understand the effect of the kernel on latency.

Each product was configured similarly to ensure the results could be compared as much as possible with an overall goal of maximizing performance. All message persistence was disabled. Message delivery was allowed to be best effort rather than semantics that require delivery. This disables acknowledgements and other traffic that impacts performance. All batching was turned off if it kept events from being delivered immediately. All testing was done on distributed machines so network serialization and buffering in the product and OS was necessary.

A testing environment was constructed to execute the tests and to gather and correlate related statistics. The environment ensured that tests were started and stopped properly, and that time stamps were measured uniformly across all products and tests. The environment ensured that software products were “warmed-up” before beginning to collect data. The majority of the 6 month project schedule was devoted to benchmarking, but the testing of each product was only allocated 2-3 weeks. Success would have been difficult without the use of the testing environment.

Table 2: Most important benchmarking tests used to measure the performance of candidate event service products.

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>Number of Publishers</th>
<th>Number of Subscribers</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload 1</td>
<td>Varying Message Size</td>
<td>1</td>
<td>1</td>
<td>Message varied from 100 bytes to 8 KB</td>
</tr>
<tr>
<td>Workload 4/4.1</td>
<td>Varying subscribers with 2 message sizes</td>
<td>1</td>
<td>1-20</td>
<td>Fixed 60 byte size and 1 KB size with varied number of subscribers</td>
</tr>
<tr>
<td>Workload 7</td>
<td>Aggregate throughput increasing sets of publisher-topic-subscriber</td>
<td>1 to 200</td>
<td>matching 1 to 200</td>
<td>Varying message size from 100 bytes to 3 KB</td>
</tr>
<tr>
<td>Workload 8</td>
<td>Throttled throughput to 1000 messages/second per publisher with increasing publishers and subscribers</td>
<td>1 until max with failure</td>
<td>matching number of subscribers</td>
<td>Message sizes of 256 and 1024 bytes</td>
</tr>
<tr>
<td>Latency and Jitter</td>
<td>Time taken for a message to travel from source to destination</td>
<td>1</td>
<td>1</td>
<td>½ round-trip time was used for latency</td>
</tr>
<tr>
<td>Reliability</td>
<td>Workload 7 was run for 8 hours</td>
<td>200</td>
<td>200</td>
<td>Statistics gathered every 5 minutes for 8 hours</td>
</tr>
</tbody>
</table>

Each performance test is called a workload. Each workload was designed to push the performance of the product by testing aspects of publish/subscribe and varying inputs as described in Table 2. For instance, Workload 1 uses one publisher publishing as quickly as possible to one subscriber to see how many messages can pass through the system. Workload 2 uses 1 publisher publishing as quickly as possible to see how the number of subscribers impacts throughput with two different message sizes. Workload 7 increased publisher/topic/subscriber sets until failure with each publisher publishing as quickly as possible. Workload 8 is similar, but the publisher is limited to 500 messages/second.

5. HARDWARE AND SOFTWARE DESCRIPTON FOR TESTING

A dedicated computer and network environment was created for executing the performance tests. All tests were executed on 3 identical Dell R720 servers. Each computer used 2 Intel Xeon processors with 6 cores and 32 GB of memory. Each server included a 1 Gb and 10 Gb network interface. The private testing environment was connected through a dedicated 10 Gb capable network switch. Scala/Java are the chosen primary languages for the TMT services and all products were tested using 64-bit Java 1.6. The operating system was 64-bit CentOS Linux, the distribution chosen for TMT.
The three machines are used as shown in Figure 3. With a broker-based message product, one server was used for publishers, one for subscribers, and one was dedicated to executing the broker. With peer to peer products one server was used for publishers and one for subscribers.

Figure 3: Configuration of test machines for broker-based products and peer-to-peer products.

6. TEST RESULTS

At the conclusion of the testing period, a final report was written that compared all the tested products. A recommendation was made for the prototype implementation based on the test results. Many tests were run for each product, and all results cannot be shown in this paper. This section shows and explains some key benchmarking results. In all the following tests, the messages consisted of a few counter values and binary padding that was not processed.

Figure 4: Single publisher throughput in msgs/sec for all tested products on 10G network as a function of message size.

Workload 1 results in Figure 4 show the throughput possible when a single publisher publishes messages as fast as possible as a function of message size without system failure. HornetQ was able to publish 111,566 600 byte messages. It is expected that the throughput in msgs/sec will decrease with message size. In a perfect system, the decrease would be linear. As shown, this is mostly true but begins to fail for larger message sizes.
The workload 4.1 results in Figure 5 show how the publisher throughput drops off as a function of the number of subscribers. In a broker system, the broker must copy the message to each of the subscribers. As expected, each product curve is roughly linear. RTI DDS shows the least variation but also the lowest overall throughput.

The next 3 figures show results from Workload 7, which tests how many publisher/topic/subscriber sets each product can support. In this test, a set consists of one publisher and one subscriber on different machines publishing and listening to a unique topic. The maximum throughput is shown for message sizes between 600 to 3100 bytes. The value plotted is the maximum throughput measured at each message size for each product. The number of publisher/topic/subscriber sets that generate the maximum throughput is different for each product. For example, with HornetQ and a 600 byte message size, the maximum occurs with 20 publisher/topic/subscriber sets.

This test exercises more of the product internals. For broker-based systems, publisher throughput is fairly linear falling off as message sizes increase. Note that although the total number of msgs/sec for the DDS products is lower, their throughput is very linear in this message size range.
Figure 7 shows the Workload 7 throughput measured on the publisher. Figure 7 shows the values measured on the subscriber for the very same tests. In a perfect system, the shapes and values measured on the publisher and subscriber machine should be the same, but in this test they are not. The biggest difference is for Redhat MRG. At 600 bytes, the publisher throughput is around 400,000 msgs/sec, but on the subscriber it is measuring 200,000 msgs/sec. It took some time to understand the difference between these two plots. The test environment ran the publisher for 30 seconds, but it ran the subscriber until all messages arrived to ensure there was no message loss. However, messages sent by the publisher often took longer than the test window to arrive. This is clearly saying something about the quality of the product. In the telescope scenario, we want the messages to arrive as soon as possible, not some number of seconds later.

Figure 8 shows how well each product subscriber “keeps up” with the publisher as the ratio of the subscriber throughput divided by the publisher throughput. The best products will show a flat curve and the closer the ratio is to 1, the better the product. Again, HornetQ is clearly the best product, but surprisingly, Redis is the second best product with 80% of its messages arriving within the measurement period. The worse product is Redhat MRG with only 40%-60% arriving within the measurement window.

Workload 9 in Figure 9 measures how many publisher/topic/subscriber sets publishing 1K messages at 1 KHz each system can support before message loss or other failure occurs. HornetQ and Redhat MRG both supported about 350 sets before failure. Redis is only slightly worse at 300 sets.
Message latency, the time it takes a message to reach the subscriber, is also an important measurement of product quality. It was not possible to synchronize the clocks on the publisher and subscriber servers to the accuracy necessary to directly measure the latency, so the round trip time was measured and halved. This eliminated the need for clock synchronization. This method does introduce uncertainty to the latency measurement. For the broker-based systems, the broker was used to measure latency rather than delivering and returning from a client.

Average latencies can be misleading because the average can be skewed by an outlier. The averages in microseconds are: HornetQ:466, Redhat MRG:427, Redis: 122, RTI DDS:147, OpenSplice DDS:28. The measured latencies are shown in Figure 10 as percentiles. The plot shows what percentage of the messages fall below a latency value. For instance, for Redis, 90% of the messages have a latency of no more than about 125 microseconds. Note the DDS products have lower and more consistent latencies. OpenSplice latency is consistently around 100 microseconds. HornetQ has the lowest latency in the 50%, 80% and 90% measurements (although at the 95% it shows issues). This explains why HornetQ is flat and close to unity in Figure 8.

If the jitter is assumed to be random and Gaussian, the variance of the latency represents the jitter measurement. The statistical values for the variance of the latency measurements are: HornetQ:18859, Redhat MRG:183, Redis:141, RTI DDS: 66, OpenSplice DDS: 28. The DDS products are superior in the latency tests. Redhat MRG has by far the worst jitter measurement.

The plot of Figure 11 shows how resources are used on the broker machine during the peak of a test with 1K messages. The broker is a potential bottleneck in the system because every message must first be sent to the broker and processed (our tests used only one broker although some products support multiple brokers). This also means every message takes one extra message hop.
Note that in this test all 24 CPUs on the broker machine were at 100%. Clearly the broker is potentially limiting performance. The 10G network is not saturated. Also note that the Redis server/broker uses very little CPU. It turns out that the Redis server is single-threaded and uses only one CPU. It might be expected that Redis would be a poor performer, but you can see from previous figures that it does very well demonstrating that Redis is a well-written, efficient product.

7. CONCLUSIONS

The HornetQ product was clearly the best overall product tested and was recommended for phase 3 prototyping. The real surprise is the second place Redis product. This product shows remarkable performance with very little resource usage. Although in use at multiple telescopes, the DDS-based products did not fare well in these tests. Based on these tests, the DDS products do not meet the TMT requirements. It is possible that the team was not able to get the optimum performance from the DDS products in the time allocated. The DDS APIs are complicated, and 2-3 weeks may not be enough time to fully understand and tune the DDS products for our use case.

Benchmarking is notoriously difficult. There are a lot of variables and it is sometimes difficult to interpret the results. However, the testing system and suite of tests did a good job in comparing the chosen products. There are potential problems in these tests. For instance, the performance impact of message marshaling and un-marshaling is not part of these tests and overall performance will be impacted. The good news is that there are products that appear to meet the TMT requirements by a significant amount. Both HornetQ and Redis are being tested more fully following the completion of these performance tests.

8. ACKNOWLEDGMENTS

The TMT Project gratefully acknowledges the support of the TMT collaborating institutions. They are the Association of Canadian Universities for Research in Astronomy (ACURA), the California Institute of Technology, the University of California, the National Astronomical Observatory of Japan, the National Astronomical Observatories of China and their consortium partners, and the Department of Science and Technology of India and their supported institutes. This work was supported as well by the Gordon and Betty Moore Foundation, the Canada Foundation for Innovation, the Ontario Ministry of Research and Innovation, the National Research Council of Canada, the Natural Sciences and Engineering Research Council of Canada, the British Columbia Knowledge Development Fund, the Association of Universities for Research in Astronomy (AURA) and the U.S. National Science Foundation.

REFERENCES