JMS Compliance in the Narada Event Brokering System

Geoffrey Fox Shrideep Pallickara
gcf@indiana.edu spallick@indiana.edu
Community Grid Labs, Department of Computer Science
Indiana University, Bloomington IN-47405

Abstract

Narada is a distributed event brokering system based on the publish/subscribe paradigm and is designed to run on a very large network of broker nodes. This paper describes the process of achieving Java Message Service (JMS) compliance in the Narada system. The paper also describes our strategy for bringing Narada's benefits – scaling, availability, routing efficiencies and failure resiliency – to existing JMS compliant systems. Finally, we also include performance data from our JMS compliant system.

Messaging systems based on queuing include products such as IBM's MQSeries [1] and Microsoft's MSMQ [2]. The queuing model with their store-and-forward mechanisms come into play when the sender of the message expects someone to handle the message while imposing asynchronous communication and guaranteed delivery constraints. A widely used standard in messaging is the Message Passing Interface Standard (MPI) [3]. MPI is designed for high performance on both massively parallel machines and workstation clusters. Messaging frameworks based on the classical remote procedure calls include CORBA [4] from OMG, DCOM [5] from Microsoft and Java RMI [6] from Sun Microsystems. In publish/subscribe systems the routing of messages from the publisher to the subscriber is within the purview of the message oriented middleware (MOM), which is responsible for routing the right content from the producer to the right consumers. Industrial strength solutions in the publish/subscribe domain include products like *TIB/Rendezvous* [20] from TIBCO and *SmartSockets* [19] from Talarian. Other related efforts in the research community include *Gryphon* [21], *Elvin* [22] and *Sienna* [23]. The push by Java to include publish subscribe features into its messaging middleware include efforts like JINI [13] and JMS [14]. One of the goals of JMS is to offer a unified API across publish subscribe implementations. Various JMS implementations include solutions like SonicMQ [15] from Progress, JMQ [16] from iPlanet and FioranoMQ [17] from Fiorano.

Narada [8,9,10,11,12] is a distributed event brokering system based on the publish/subscribe paradigm and is designed to run on a very large network of broker nodes. The distributed cluster architecture, which results in the creation of "small world networks" [7], allows Narada to support large heterogeneous client configurations that scale to arbitrary size. Protocols controlling the addition of broker nodes in Narada ensure that brokers being added do not result in bandwidth degradation because of their interconnectivity. Changes to the Narada broker network include addition/removal of broker nodes and communication channels between brokers. Narada incorporates a scheme for the controlled propagation of information pertaining to changes to the broker network to relevant sections of the broker network. The propagation scheme in tandem with the broker organization makes the creation of efficient broker network maps (BNM) an efficient one. The paths computed, to reach any given destination, from these BNMs tend to be very efficient since the protocols controlling the addition of brokers and associated connections have ensured that these connections do not result in bandwidth degradations. Clients interested in using Narada can attach themselves to one of the broker nodes. Clients specify an interest in the type of events that they are interested in and the service routes events, which satisfy the constraints specified by the clients. Clients can have prolonged disconnects from the broker network and can also roam the network (in response to failure suspicions or for better response times) and attach themselves to any other node in the broker network. Events published during the intervening period, of prolonged disconnects and roams, must still be delivered to clients that originally had an interest in these events. The delivery constraints are satisfied in the presence of broker and connection failures. Narada provides for a hierarchical dissemination scheme for the delivery of events to relevant clients. The system provides for an efficient calculation of routes for disseminations, based on BNMs, while ensuring that paths computed comprise only those brokers and connections that have not failed or have not been failure suspected. Narada is designed as an event brokering system to support Community Grids [27] and needs to encompass both peer to peer (P2P) [24] and the traditional centralized middle tier style of interactions. We base support for P2P interactions through JXTA [19], which is a set of open, generalized protocols to support P2P style communications. Details pertaining to the JXTA integration can be found in [26]. This paper describes the process of achieving JMS compliance in the Narada system and strategies that bring Narada functionality to JMS clients. This paper is organized as follows. In section 1 we describe JMS compliance and what the compliance implies. Section 2 provides the rationale for achieving JMS compliance in Narada, while section 3 describes our strategy to do so. Section 4 describes the applications that were used to test Narada's JMS compliance. Section 5 presents our strategy to help JMS clients leverage Narada functionality in large distributed settings in addition to the guarantees accorded to these clients based on their conformance to the JMS specification. Section 6 provides performance data from our JMS compliant system.

1.0 JMS Compliance

The JMS specification [14] results in JMS clients being able to interoperate with any service provider, this process generally requires clients to incorporate a few changes in the initialization sequences that are specific to the vendor being used, after which interactions, as specified in the JMS API, continue. JMS clients are provider agnostic, and with a change in initialization sequences a client should be able to function just as well with any other provider. JMS does not provide for interoperability between JMS providers, though interactions between clients of different providers can be achieved through a client that is connected to the different JMS providers.

Clients need to be able to invoke operations as specified in the specification; expect and partake from the logic and guarantees that would go along with these invocations. These guarantees range from receiving only those events that *match* the specified subscription to receiving events that were published to a given topic irrespective of the failures that took place or the duration of client disconnect. Clients are built around these calls and the guarantees (implicit and explicit) that are associated with them. Failure to conform to the specification would result in clients expecting certain sequences/types of events and not receiving those sequences, which in turn lead to deviations that could result in run time exceptions. JMS applications need to be built entirely on JMS compliant calls. Some providers tend to provide specialized calls that are either a sequence of JMS calls or some specific features provided by the provider. In either case such applications tend to result in systems that are *not* JMS compliant.

2.0 Rationale for JMS compliance in Narada

There are two objectives that we seek to meet while providing JMS compliance within Narada:

1. *Providing support for JMS clients within the system.* This objective provides for JMS based systems to be replaced transparently by Narada and also for Narada clients (including other messaging styles supported by Narada such as JXTA) to interact with JMS clients. Support for clients conforming to a mature messaging product within the research prototype opens up Narada to a plethora of applications developed around JMS. Furthermore, Narada could then use these applications to further optimize certain most commonly used features exploited by these applications. The requisite changes for optimizations would need to be made at the Narada core. JMS clients could receive messages from non-JMS based clients and the interaction could proceed seamlessly. Narada also routes JXTA interactions efficiently; it is thus possible for JMS clients and JXTA peers and Narada clients to interact via the Narada brokering system.
2. *To bring Narada functionality to JMS clients/systems developed around it.* This approach (discussed in section 4.0) will transparently replace single server or limited server JMS systems with a very large scale distributed solution, with failure resiliency, dynamic real time load balancing and scaling benefits that accompany highly available systems.

3.0 JMS compliance in Narada

Narada provides clients with connections that are then used for communications, interactions and any associated guarantees that would be associated with these interactions. Clients specify their interest, accept events, retrieve lost events and publish events over this connection. JMS includes a similar notion of connections. To provide JMS compliance we write a bridge that performs all the operations that are required by Narada connections in addition to supporting operations that would be performed by JMS clients. Some of the JMS interactions and invocations are either supported locally or are mapped to corresponding Narada interactions initiated by the connections. Each connection leads to a separate instance of the bridge. In the distributed JMS strategy, section 4.0, it is conceivable that a client, with multiple connections and associated sessions, would not have all of its connections initiated to the same broker. The bridge instance per connection helps every connection to be treated independently of the others, despite each one being registered to different brokers.

In addition to connections, JMS also provides the notion of sessions that are registered to specific connections. There can be multiple sessions on a given connection, but any given session can be registered to only one connection. Publishers and subscribers are registered to individual sessions. Support for sessions is provided locally by the bridge instance associated with the connection. For each connection the bridge maintains the list of registered sessions, the sessions in turn maintain a list of subscribers. Upon receipt of an event over the connection the corresponding bridge instance is responsible for forwarding the event to the appropriate sessions, which then proceed to deliver the event to the listeners associated with subscribers having subscriptions matching the event. In Narada each connection has a unique ID and guarantees are associated with individual connections. This ID is contained within the bridge instance and is used to deal with recovery and retrieval of events after prolonged disconnects or induced roam due to failures.

We also need to provide support for the creation of different message types and assorted operations on these messages as dictated by the JMS specification, along with serialization and de-serialization routines to facilitate transmission and reconstruction. In Narada, events are routed as streams of bytes, so as long as we provide marshalling un-marshalling operations associated with these types there are no issues with support for these message types.

In JMS the topics are generally "/" separated (e.g. Course/CPS616/Session/HPJava) while Narada supports topics which are created as <*tag, value*> pairs (e.g. Courses=CPS616, Session=HPJava), with a provision for wild card operators in the values associated with tags (e.g. Courses=*, Session=HPJava) We implemented a wrapper which efficiently maps "/" separated topics into those that are <*tag, value*> separated, if the topics are already specified as <*tag, value*> pairs no further processing would be done. The destination topic contained within the JMS message is of course not touched. In addition to this the matching algorithm [21] used in Narada is augmented with the JMS selector mechanism implemented in openJMS [18].

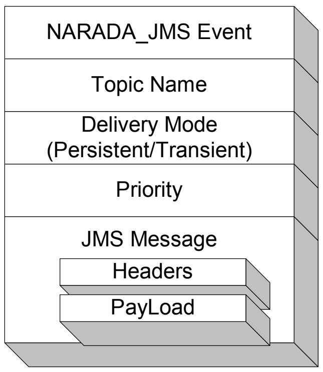

Figure 1: Narada/JMS Event

The JMS subscription request is mapped to the corresponding Narada Profile propagation request and propagated through the system. The bridge maps persistent/transient subscriptions to the corresponding Narada subscription types. JMS messages that are published are routed through the Narada broker as a Narada event. The anatomy of a Narada/JMS event, encapsulating the JMS messages, is shown in figure 1. Events are routed based on the mapped JMS Topic name contained in the event. Storage to databases is done based on delivery mode indicator in the event.

JMS provides a call that ensures that subscribers do not receive messages issued by publishers registered to the same connection. The bridge is responsible for suppressing these messages from being delivered to sessions registered to the connection that created the message. When a message is published by a publisher, the Narada/JMS event also contains information regarding the connection that event was published over. When the event is received at the connection, this information is used to suppress delivery of the message, retrieved from the event, to those subscribers that should not receive messages that were published over that connection. This information is also used by Narada to ensure that the event is not routed to the connection in the first place.

Existing JMS applications where we successfully replaced the JMS provider with Narada include the multimedia intensive distance education audio/video/text/ application conferencing system [28] by Anabas Inc and the Online Knowledge Center (OKC) [29] developed at IU Grid Labs. Both these applications were based on SonicMQ.

4.0 The Distributed JMS Solution

By having individual brokers interact with JMS clients, we have made it possible to replace the JMS provider's broker instance with a Narada broker instance. The features in Narada are best exploited in distributed settings. However, the distributed network should be transparent to the JMS clients. What we seek is that the traditional initializations involving the specification of a single hostname and port number should still be left intact. Each Narada broker should still be able to function as a standalone broker. Existing systems built around JMS should be easily replaced with the distributed model with minimal changes to the client. In fact, the initialization changes should be identical to those that are required when a JMS provider is changed. JMS clients using a standalone Narada broker as the JMS provider should not have to make any changes with any associated initializations. In general, setups on the client side are to be performed in a transparent manner. Another important constraint in the proposed distributed JMS solution is that no changes are to be made to the Narada core and the associated routing, propagation and destination calculation algorithms. The solution to the transparent distributed JMS solution would allow for any JMS based system to benefit from the distributed solution. Applications would be based on source codes conforming to the JMS specification while the scaling benefits, routing efficiencies, failure resiliency accompanying the distributed solution are all automatically are inherited by the integrated solution.

A simple solution to this problem would be to set up the Narada broker network after which individual clients choose the broker that they would connect to. Thus, individual clients still specify the broker they need to connect to, the only difference being that they now have a much larger set of brokers to choose from. The clients also need to make sure that the broker that they would be connecting to, is currently up and running. This scheme forces JMS clients to be aware of the broker interconnection scheme and also to be aware of brokers that have failed, been failure suspected, recovered and those that have been newly added. The process of moving towards a distributed JMS architecture is obviously not transparent to the JMS clients. Furthermore, it is conceivable that clients would continue to access a certain known broker over and over again while newly added brokers continue to be under utilized.

To circumvent the problem of discovering valid brokers we introduce the notion of *broker locators*. The broker locators' primary function is the discovery of brokers that a client can connect to. Clients thus do not need to keep track of the brokers and their states within the broker network. The broker locator has certain properties and constraints based on which it arrives at the decision regarding the broker that a client would connect to.

1) *Load balancing* – Broker locators keep track of the number of concurrent connections maintained by each broker. It also maintains the published limit on concurrent connections at a broker node. Connection requests are always forked off to the best available broker. This enables us to achieve dynamic real time load balancing.
2) *Incorporation of new brokers* – When a new broker is available that broker would be the best available broker to handle new connection requests. Clients thus incorporate these brokers faster into the routing fabric.
3) *Availability* – The broker locator itself should not constitute a single point of failure neither should it be a bottleneck for clients trying to utilize network services. The Narada topology allows brokers to be part of domains. There could be more than one broker locators for a given administrative domain.
4) *Minimal logic* – The broker locator is not supposed to maintain active concurrent connections to any element within the Narada system. The loss of the broker locator should not affect processing pertaining to any other node within the system.

4.1 Metrics for Decision Making

To determine the best available broker to handle the connection request, the metrics that play a role in the broker locator's decision include the IP-address of the requesting client, the number of connections that are still available at the brokers that are best suited to handle the connection, the number of connections that currently exist, the computing capabilities and finally the availability of the broker (a simple ping test). We now discuss the sequence of operations that take place once a decision has been made regarding the broker that is best suited to handle connection request.

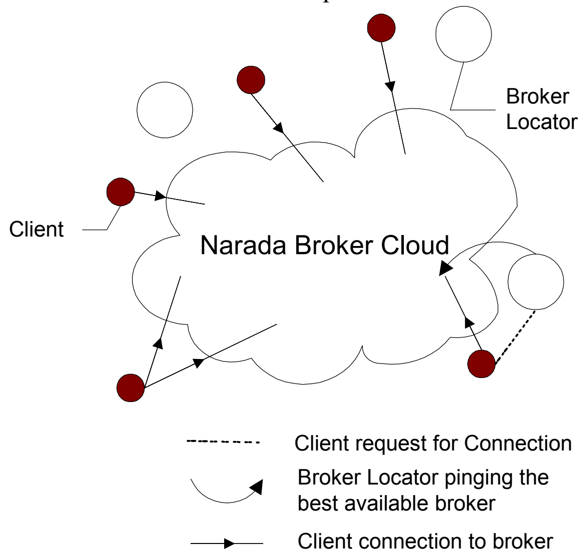
Broker
Locator
Client
Narada Broker Cloud
Client request for Connection
Broker Locator pinging the best available broker
Client connection to broker

Figure 2: Distributed JMS approach

Once a valid broker has been identified, the broker locator also verifies if the broker process is currently up and running. Once this is confirmed the broker locator proceeds to route broker information to the client. If the broker process is not active, the computed broker is removed the list of available brokers the broker locator computes the next best broker and the sequence of actions listed above is repeated. The broker information propagated to the client includes the hostname/IP-address of the machine hosting the broker, the port number on which it listens for connections/communications and the transport protocol that is used for communication. The client then uses this information to establish a communication channel with the broker. Figure 2 depicts this sequence of operations and also the scenario where a client has connections to two different brokers. Once it is connected to a Narada broker, the JMS client can proceed with interactions identical to the single broker case.

Based on system requirements new brokers can be added to deal with load balancing and scaling issues. Furthermore failure of brokers will not affect clients since clients could be induced to 'roam' the broker network and attach itself to another available broker node. The system guarantees that the client will not loose events due to any failures that may takes place in the system. The other advantage is that this distributed solution would be selective in the links and the nodes that it employs to ensure dissemination of events. Narada also ensures that every broker, either targeted or en route to one will always traverse the shortest path to reach its destination. Furthermore, the only brokers that are part of these shortest paths are those that have not failed. The guaranteed delivery scheme within Narada does not require every broker to have access to a stable store or DBMS. The replication scheme is flexible and easily extensible. Stable storages can be added/removed and the replication scheme can be updated. Stable store's can fail but they do need to recover within a finite amount of time, however during these failures the clients that are affected are those that were being serviced by the failed storage.

5.0 JMS Performance Data

To gather performance data, we run an instance of the SonicMQ (version 3.0) broker and Narada broker on the same dual CPU (Pentium-3, 1 GHz, 256MB) machine. We then setup 100 subscribers over 10 different JMS TopicConnections on another dual CPU (Pentium-3, 866MHz, 256MB) machine. In addition there is a *measuring* subscriber and a publisher that are set up on a third dual CPU (Pentium 3, 866MHz, 256MB RAM) machine. Since we will be computing communication delays setting up the measuring subscriber and publisher on the same machine enables us to obviate the need for clock synchronizations and differing clock drifts. The three machines involved in the benchmarking process have Linux (version 2.2.16) as their operating system. The runtime environment for the broker, publisher and subscriber processes is Java 2 JRE (Java-1.3.1, Blackdown-FCS, mixed mode).

Subscribers subscribe to a certain topic and the publisher publishes to the same topic. Once the publisher starts issuing messages the factor that we are most interested in is the *transit delay* in the receipt of these messages at the subscribers. This delay corresponds to the response times experienced at each of the subscribers. We measure this delay at the measuring subscriber while varying the publish rates and message sizes of the messages being published. We control the publish rates by varying the time interval between the publishing of two consecutive messages. We vary the message size by changing the payload contained in the message. For a sample of messages received at the measuring subscriber we calculate the *mean transit delay* and the standard deviation within this sample. We also calculate the *system throughput* measured in terms of the number of messages received per second at the measuring subscriber.

Figures 3-5 depicts the transit delays for JMS clients under Narada and SonicMQ for varying publish rates and payload sizes. Figures 6-8 depicts the standard deviation associated with message samples under conditions depicted in figures 3-5 respectively. Figures 9 and 10 depict the system throughputs, during high publish rates and smaller payloads, for Narada and SonicMQ clients respectively. As can be seen from the results Narada compares very well with SonicMQ while also outperforming SonicMQ in several cases. Furthermore, the standard deviation associated with the message samples (for individual test cases) received at clients in Narada were, for the most part, lower than those at clients in SonicMQ for the cases that were benchmarked.

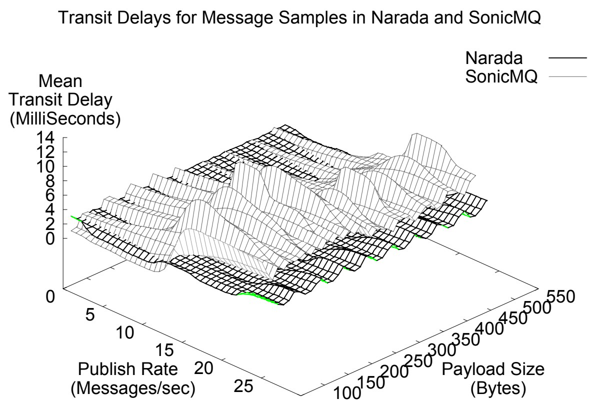

Figure 3: Transit Delays - Lower publish rates smaller Payloads

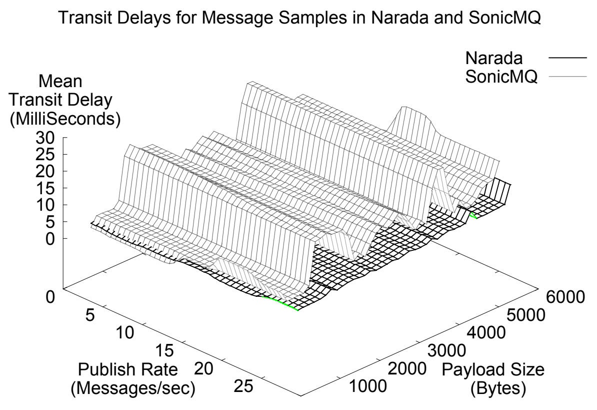

Figure 4: Transit Delays - Lower publish rates bigger payloads

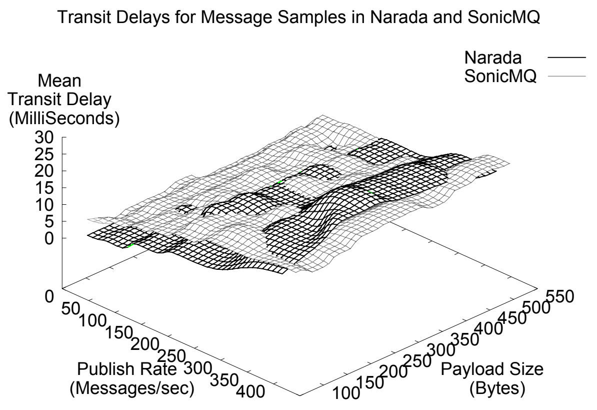

Figure 5: Transit Delays - Higher publish rates smaller payloads

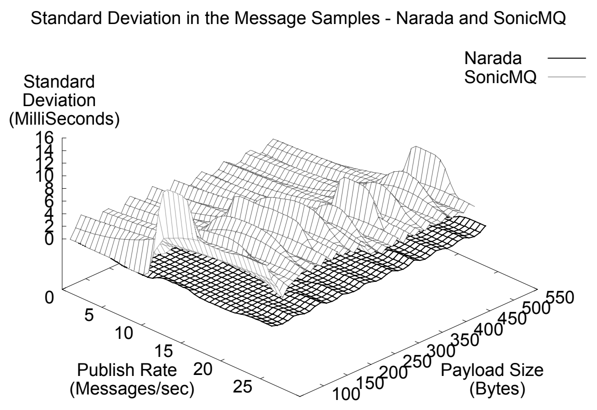

Figure 6:Standard Deviation - Lower publish rates smaller payloads

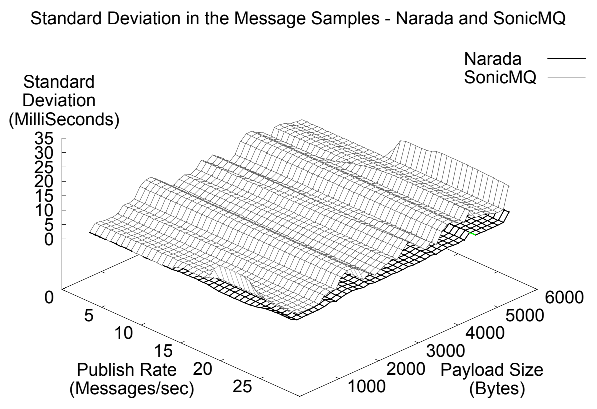

Figure 7:Standard Deviation - Lower publish rates bigger payloads

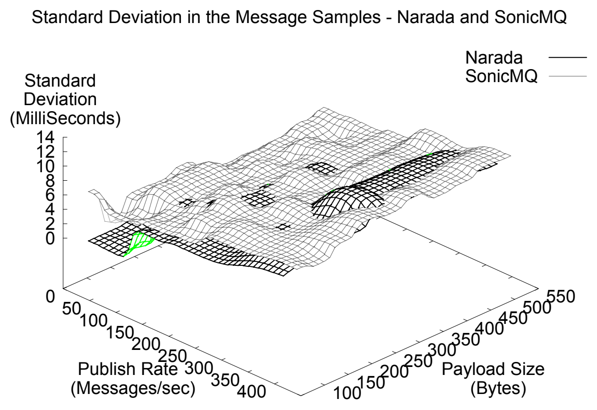

Figure 8: Standard Deviation - Higher publish rates smaller payloads

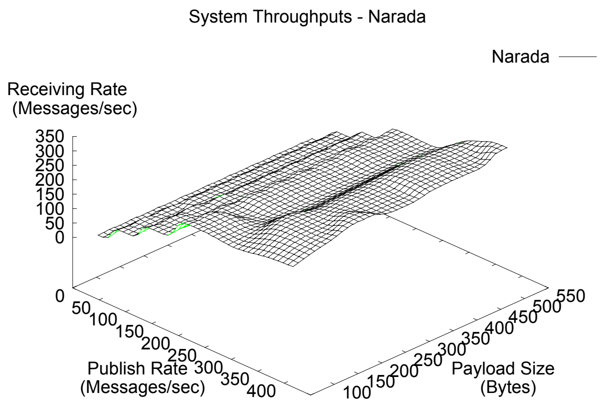

Figure 9: System Throughputs (Narada) - Higher publish rates smaller payloads

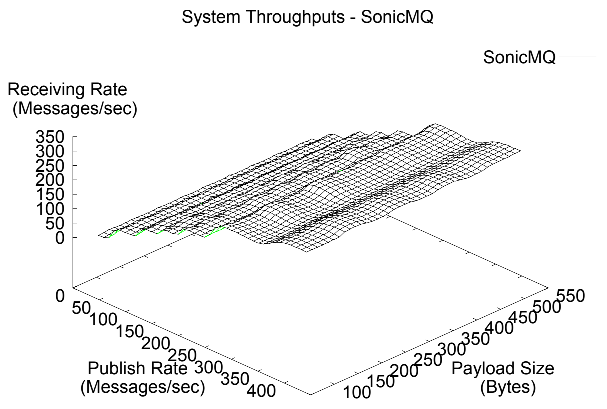

Figure 10: System Throughputs (SonicMQ) - Higher publish rates smaller payloads

6.0 JMS over UDP in Narada

We also support JMS over UDP. This feature facilitates the creation of JmsTopicConnections, which provide UDP communication support for the hosted sessions and the publishers and subscribers associated with these sessions. We however do not provide packet loss and out of order delivery detection and associated error corrections. This feature should thus be used only for transient events and applications where packet losses can be sustained. However with advancements in networking technology the errors associated with UDP communication tend to very few over an extended duration of time. Applications that can sustain such small losses can greatly benefit from this feature.

7.0 Conclusion

In this paper we outlined the process of providing JMS compliance within Narada. There are several benefits to be accrued by this compliance. We also describe a scheme that allows existing JMS based applications to inherit Narada features in distributed settings.

References:

1. IBM MQSeries. http://www.ibm.com/software/mqseries
2. Peter Houston. Building Distributed Applications with Message Queuing Middleware. Microsoft White Paper.
3. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. May 1994.
4. Object Management Group (OMG). CORBA Services. http://www.omg.org/technology/documents/ . June 2000.
5. Guy Eddon and Henry Eddon. Understanding the DCOM Wire Protocol by Analyzing Network Data Packets. *Microsoft Systems Journal*. March 1998.
6. Sun Microsystems. Java Remote Method Invocation (Java RMI) - Distributed Computing for Java. White Paper. http://java.sun.com/marketing/collateral/javarmi.html
7. D.J. Watts and S.H. Strogatz. Collective Dynamics of Small-World Networks. *Nature*. 393:440. 1998.
8. The Narada Event Brokering System http://grids.ucs.indiana.edu/ptliupages/projects/narada/
9. Geoffrey Fox and Shrideep Pallickara, An Event Service to Support Grid Computational Environments, to be published in *Concurrency and Computation: Practice and Experience*, Special Issue on Grid Computing Environments.
10. Geoffrey Fox and Shrideep Pallickara, The Narada Event Brokering System: Overview and Extensions. To appear in the Proceedings of the *2002 International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA'02), Las Vegas June 2002.
11. Geoffrey C. Fox and Shrideep Pallickara , An Approach to High Performance Distributed Web Brokering. *ACM Ubiquity* Volume2 Issue 38. November 2001.
12. Pallickara, S., "A Grid Event Service." PhD Syracuse University, 2001.
13. Ken Arnold, Bryan O'Sullivan, Robert Scheifler, Jim Waldo and Ann Wollrath. The Jini Specification. Addison-Wesley. June 1999.
14. Mark Happner, Rich Burridge and Rahul Sharma. Sun Microsystems. Java Message Service Specification. 2000. http://java.sun.com/products/jms
15. SonicMQ JMS Server http://www.sonicsoftware.com/
16. iPlanet JMQ. Java Message Queue Documentation. http://www.iplant.com
17. Fiorano Corporation. A Guide to Understanding the Pluggable, Scalable Connection Management (SCM) Architecture - White Paper. http://www.fiorano.com/products/fmq5_scm_wp.htm
18. The OpenJMS Project http://openjms.exolab.org/
19. Talarian Corporation. *SmartSockets: Everything you need to know about middleware: Mission Critical Interprocess Communication*. Technical Report: URL: http://www.talarian.com/products/smartsockets
20. TIBCO Corporation. TIB/Rendezvous White Paper. URL: http://www.rv.tibco.com/whitepaper.html, June 1999.
21. Marcos Aguilera, Rob Strom, Daniel Sturman, Mark Astley and Tushar Chandra. Matching Events in a Content-based Subscription System. *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*. May 1999.
22. Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. *In Proceedings AUUG97*, September 1997.
23. Antonio Carzaniga, David S. Rosenblum and Alexander L. Wolf. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. *Proceedings of 19th ACM Symposium on Principles of Distributed Computing,* July 2000.
24. "Peer-To-Peer: Harnessing the Power of Disruptive Technologies", edited by Andy Oram, O'Reilly Press March 2001.
25. Sun Microsystems. The JXTA Project and Peer-to-Peer Technology http://www.jxta.org
26. Geoffrey Fox, Shrideep Pallickara, Xi Rao, Pei Qinglin. Scaleable Event Infrastructure for Peer-to-Peer Grids.
27. Geoffrey Fox, Ozgur Balsoy, Shrideep Pallickara, Ahmet Uyar, Dennis Gannon, Aleksander Slominski. Community Grids. Proceedings of the *International Conference on Computational Science*. Amsterdam, April 2002.
28. The Anabas Conferencing System. http://www.anabas.com
29. The Online Knowledge Center (OKC) Web Portal http://judi.ucs.indiana.edu/okcportal/index.jsp