Dynamic Policy-Driven Quality of Service in Service-Oriented Systems

Joseph P. Loyall, Matthew Gillen, Aaron Paulos
BBN Technologies, Cambridge, MA USA
e-mail: {jloyall, mgillen, apaulos}@bbn.com

Larry Bunch, Marco Carvalho
Institute for Human Machine Cognition
Ocala, FL USA
e-mail: {lbunch, mcarvalho}@ihmc.us

James Edmondson, Pooja Varshneya,
Douglas C. Schmidt
Vanderbilt University, Nashville, TN USA
e-mail: {james.r.edmondson, pooja.varshneya,
d.schmidt}@vanderbilt.edu

Andrew Martignoni III
The Boeing Company, St. Louis, MO USA
e-mail: andrew.j.martignoni-iii@boeing.com

ABSTRACT

Service-oriented architecture (SOA) middleware has emerged as a powerful and popular distributed computing paradigm due to its high-level abstractions for composing systems and hiding platform-level details. Control of some details hidden by SOA middleware is necessary, however, to provide managed quality of service (QoS) for SOA systems that need predictable performance and behavior. This paper presents a policy-driven approach for managing QoS in SOA systems. We discuss the design of several key QoS services and empirically evaluate their ability to provide QoS under CPU overload and bandwidth-constrained situations.

Keywords: service oriented architecture, quality of service

I. INTRODUCTION

Service-oriented architecture (SOA) middleware has emerged as a powerful software engineering environment for developing distributed applications and systems. SOA offers benefits in system construction including dynamic discovery of system components (i.e., services), high-level abstractions for encapsulation and system composition, and lifecycle management and virtual execution environments that hide details about the platforms in which the system is run.

These characteristics have increased adoption of SOA in enterprise environments, but, have limited its adoption in other domains, such as mission-critical distributed, real-time, and embedded (DRE) systems, that have stringent performance, resource contention, and predictability requirements. Conventional SOA middleware, such as J2EE and .NET, lacks key QoS capabilities that are needed by systems in DRE domains, including the visibility and control of shared and constrained resources, and the mediation of competing demands for resources. Ironically, the limitations of SOA are at least partially due to a fundamental tension between the higher-level abstractions it attempts to provide and the detailed visibility and control needed to provide QoS effectively in dynamic and heterogeneous DRE environments.

This paper describes key capabilities needed in SOA middleware to make it suitable to support applications and systems that require predictable QoS, including (1) task management, (2) bandwidth management, (3) aggregation of competing resource demands, and (4) QoS policy-driven prioritization and scheduling strategies. As part of a SOA-based QoS management system named *QoS-Enabled Dissemination (QED)* [11], we prototyped these capabilities for the JBoss and Java Message Service (JMS) SOA middleware and evaluated their performance in the context of CPU overload, constrained bandwidth, and dynamic policy changes.

The remainder of the paper is organized as follows: Section II briefly introduces SOA and the challenges it presents for managed QoS; Section III describes the QoS management capabilities we designed and their prototype instantiations; Section IV analyzes the results of experiments we conducted to evaluate the improved QoS predictability, control, and performance over a baseline JBoss and JMS system in the face of CPU overload and constrained bandwidth, and experiments to evaluate the speed of dynamic QoS policy updates in QED; Section V compares QED with related work; and Section VI presents some concluding remarks.

II. SERVICE ORIENTED ARCHITECTURE

A. Overview of SOA

SOA is a progression in the evolution of middleware and distributed software engineering technologies, building upon the basis of distributed objects and components. It encapsulates business functionality as services, such as Enterprise JavaBeans (EJB) or Managed Beans (MBeans), much in the way that component models, such as the CORBA Component Model (CCM), encapsulated functionality as components. SOA service interfaces are specified in standard interface description languages (IDL), such as the Web Services Description Language (WSDL), an evolution of the IDLs used for components and distributed objects preceding them.

SOA typically also includes an execution container model and support for inter-service communication, e.g., provided by an Enterprise Service Bus (ESB). SOA middleware, like the component middleware that preceded it, provides an abstract development, deployment, and execution layer that encapsulates the details of distribution, inter-service and client-to-service communication, threading, and runtime execution. SOA middleware also extends the assembly and deployment languages (often based on XML) of distributed components to include dynamic service discovery (e.g., the Universal Description Discovery and Integration, UDDI) and *orchestration* of services, which combines assembly of services, workflow description, and runtime control of workflow and service execution.

This research has been sponsored by the U.S. Air Force Research Laboratory under contract FA8750-08-C-0022.

JBoss is an open-source implementation of Java 2 Enterprise Edition (J2EE) SOA middleware that supports the SOA lifecycle goals of service orchestration, deployment, and execution (e.g., JBPM for orchestration). For the purpose of this paper we concentrate on two parts of JBoss: (1) JBoss application server, which provides a container model in Java for executing services, and (2) JMS for topic-based client-to-service communication and data transfer.

B. *Challenges for providing QoS in SOA environments*

Although the SOA abstractions in JBoss and JMS simplify developing, composing, and executing SOA applications, they incur challenges for managing the QoS of these applications. For example, the JBoss container hides runtime- and platform-level details, e.g., the number of threads, invocation of services, assignment of service invocations to threads, and CPU thread scheduling. JBoss can thus create more threads than can be run efficiently by the hardware (leading to CPU overload) or fewer threads than needed by application and system services (leading to CPU under-utilization). Likewise, without QoS management, important services in JBoss can block waiting for threads, while less important services run (leading to priority inversion). Moreover, since service execution times vary, service invocations can tie up threads for potentially unbounded amounts of time.

In a similar manner, the JMS communication middleware hides details, such as the transport protocol, the amount of bandwidth available and used, contention for bandwidth, and communication tradeoffs (e.g., loss and delay characteristics). JMS provides point-to-point and publish-subscribe communication, reliable asynchronous communication, guaranteed message delivery, receipt notification, and transaction control. JMS does not expose any queue or flow control, however, so that large rates of messages, constrained bandwidth, or varying message sizes can end up with more important messages being delayed (even indefinitely) while less important messages are sent. In extreme cases, queues can fill up or grow unbounded, leading to resource exhaustion, information loss, or unbounded delay.

JBoss and JMS do each provide certain QoS parameters and configuration choices in their specifications. For example, JMS specifies three QoS parameters: delivery mode (persistent or non-persistent), priority, and time-to-live, that provide hints to JMS implementations to support QoS. There is little support, however, for visibility into bandwidth availability and use, matching flow of information to the bandwidth available, and managing contention for bandwidth across multiple JMS connections.

JBoss includes a message bridge for sending messages reliably across clusters, WANs, or unreliable connections that specifies the following three levels of QoS:

- QOS_AT_MOST_ONCE specifies unreliable delivery where messages may be lost, but will not reach their destinations more than once.
- QOS_DUPLICATES_OKAY specifies reliable delivery. Messages might be delivered more than once if a message arrives but its acknowledgement is lost.
- QOS_ONCE_AND_ONCE_ONLY specifies reliable delivery of both a message and its acknowledgement.

Although these QoS levels specify message delivery reliability, they do not specify the performance, resource usage, or prioritization of messages or information flows. Moreover, these QoS features of JMS and JBoss lack support for aggregation and mediation of competing QoS requirements for users and connections that are sharing bandwidth, for coordinating CPU and bandwidth usage, and for dynamic bottleneck management. In contrast, the QED QoS capabilities described in Section III provide this support.

III. QoS Management Capabilities for SOA

Our work on QoS management for DRE systems in SOA environments has yielded QED, which is SOA-based middleware whose QoS capabilities address the challenges described in Section II.B. This section describes the following QoS services and mechanisms we have developed for QED shown in Figure 1: (1) An aggregate QoS management service; (2) A QoS policy service; (3) A task management local QoS manager; and (4) A bandwidth manager.

A. *Aggregate QoS management service*

The QED aggregate QoS management service creates a set of policies guiding the behaviors of the local QoS managers that enforce CPU scheduling and bandwidth utilization. The purpose of the aggregate QoS manager is to maintain predictable behavior throughout the orchestrated system of clients and services. Since the load of client and user demands will vary, it is likely that there may not be enough bandwidth or CPU resources to provide the QoS requested by everyone. If these resources are not managed properly, no user will get a reasonable level of QoS (i.e., leading to the *tragedy of the commons* [7]). Aggregate QoS management mediates conflicting demands for QoS management, providing available resources to the most critical services or clients.

Each *local* QoS manager (task, submission, and dissemination) has only a local view. The aggregate QoS manager thus provides policies that are consistent to related control points. For example, if a policy indicates that a service invocation should have a high priority for CPU thread scheduling, then information produced by the service invocation should also have high priority for dissemination to clients or other services.

When a client is authenticated to gain access to services (using an authentication service), the authentication credentials and other information about the user, client, and orchestration are forwarded to the aggregate QoS manager. The aggregate QoS manager accesses the policy store to get the list of policies that can apply to the user, client, and operations that the client can invoke. The aggregate QoS manager resolves the list to remove overlapping and contradictory policies, using a configurable policy precedence scheme described below. The equivalent of a *session* is created for the client's operations on services in its orchestration and the relevant policies are distributed to the local QoS managers using properties on the session.

In this way, the aggregate QoS manager translates high level, goal- and user-specified QoS policies into actionable QoS policies that apply to observable properties of a client, operations it can invoke, information (e.g., parameters or

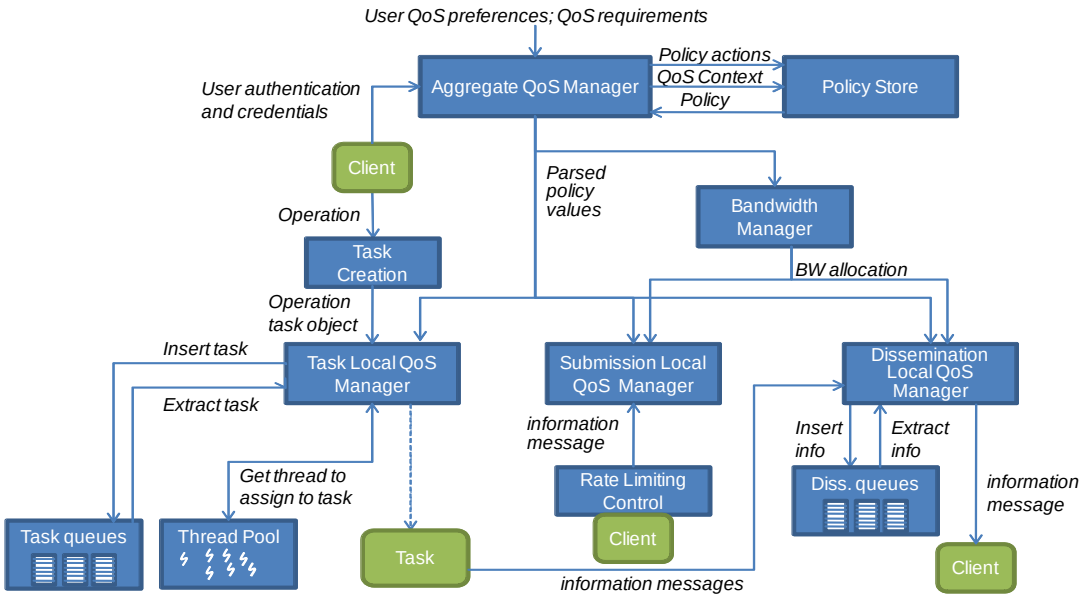

Figure 1. QED Capabilities Providing Aggregate QoS Management for SOA-based DRE Systems

messages), and resources. The actionable policies distributed to the local QoS managers can be checked quickly at the local enforcement points via attribute lookups and relational comparisons so they can be applied in the path of the control and data flow. In contrast, policy lookup, parsing, and distribution of policies by the aggregate QoS manager is out-of-band with the control and data flow and is relatively infrequent compared to local policy enforcement. They occur only on discrete events that affect the makeup of the overall distributed system, such as the entry of new clients, resource failure, and changes in overall QoS goals or policies.

B. *QoS policies*

Each QoS policy includes a condition over observable properties of the system, which is defined as follows:

QoS Policy: f(E, M, O, R) ⇒ QoS settings

The current QED prototype supports conditions over observable properties of *Entities (E)* such as clients or user credentials, *Information (M)* such as message types or information metadata, *Operations (O)* such as service invocations, and *Resources (R)* such as queue lengths, threads, or bandwidth. *QoS settings* provide guidance to the local QoS management components and are defined as the combination of an *importance (i)*, a set of *QoS preferences (P)*, and a *precedence level (v)*, as follows:

QoS settings: (i, P, v)

The importance is a high-level measure of the relative value of an operation, type, or client to a system's overall goals. This value is used, along with other factors such as cost, to prioritize processing and dissemination of information. QoS preferences define limits and tradeoffs among aspects of QoS, such as deadlines and acceptable ranges of quality.

The precedence level aids in selecting between conflicting policies; higher precedence policies are enforced in favor of lower precedence ones. In general, more specific policies should override less specific ones. Policies are maintained in the Policy Store service.

C. *Task management*

Achieving predictable performance requires managing the execution of all CPU intensive operations, such as service invocations, for each CPU (or equivalent virtual machine, VM) onto which clients and services are distributed, including the following capabilities:

- Prioritized scheduling of operations based on importance and cost (e.g., time to execute).
- Limiting the size of the thread pool to a number of threads that can be executed on the CPU (or a portion allocated to the VM) without overloading it.
- Scheduling according to an appropriate policy, such as *strict* or *weighted fair*.

To manage these tasks, QED provides *Local Task* Managers, whose design is shown in Figure 2. Each Local Task Manager manages the CPU intensive operations for a given CPU or VM using priority scheduling. The goal is to avoid

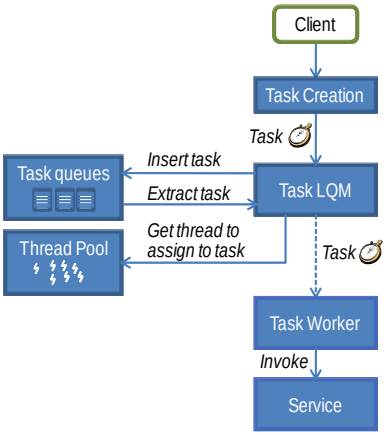

Figure 2. QED Local Task Manager Design

CPU overload in the form of too many threads or service invocations and to avoid priority inversion, in the form of lower priority service invocations getting CPU when higher priority service invocations are awaiting execution.

When CPU-intensive operations (e.g., service invocation) are performed, tasks are created and submitted to the Task Manager, where they are inserted into a binned priority queue using a configurable number of bins (queues), each representing a priority.[1] Task creators calculate an importance (derived from a policy applied to the operation, information type, and/or client) and cost for the task. The Task Manager takes importance and cost as inputs and generates a priority (bin assignment). Binned queues also allow QED to support a weighted-fair policy, which is hard to implement in a heap-based implementation. The tradeoff is that we have a fixed granularity with which to distinguish tasks.

The Task Manager assigns threads from the thread pool to tasks according to a queue management strategy under control of the aggregate QoS manager. QED currently has two queue management policies implemented: strict and weighted fair. In both the strict and weighted fair policies, there is FIFO behavior within individual bins. In *Strict*, the Task Manager always pops off the highest-priority bin that is not empty. The weighted-fair queue management policy provides an opportunity to service all bins with a built in weighting to service higher priority bins more often.

Estimating the cost of operations for use in the scheduling decision requires an accurate model of service execution time. Constructing such models is hard in the dynamic DRE systems we target since service execution time can vary significantly depending on the power of the platform on which it is deployed and characteristics of inputs to the service. We combine two approaches to solve this problem. First, we use heuristics—based in part on experimental and testing runs—to identify the conditions under which a service is more or less costly to execute. Second, a QoS monitoring service [11][12] that is part of our overall solution monitors service execution and reports the measured time (stored as a time series) to the local task manager so that its model of service execution time improves as the system executes.

D. Bandwidth management

The Bandwidth Manager is a host-level entity that assigns bandwidth slices for inbound and outbound communications based on policy provided by the aggregate QoS manager. For SOA architectures, bandwidth is managed at the level of information objects, not packets, as is done by network-level QoS, since the loss or delay of an individual packet could invalidate an entire (and potentially much larger) object of information. Here the inbound and outbound managers are referred to as the Submission Local QoS Manager (LQM) and Dissemination LQM, respectively. The current version of the Bandwidth Manager provides a static bandwidth allocation per interface and to each of the LQMs.

The Submission LQM, shown in Figure 3, manages the consumption of inbound bandwidth by throttling external

[1] Bins ensure that insertion time of newly created tasks is constant vs. the *log n* insertion time needed for heap-based priority queues.

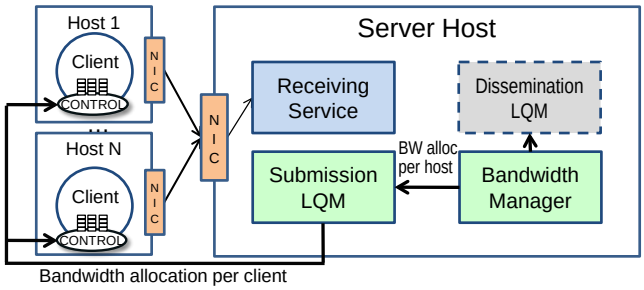

Figure 3. QED Submission LQM Design

clients and providing bandwidth slices to cooperative SOA clients, which in turn enforce the restriction on the client's outbound connection. When coupled with information prioritization (enforced by priority-driven differential queuing on the client's outbound side), this form of incoming message rate control serves two purposes: (1) the rate throttling reduces the potential for resource overload on the service hosts and (2) the utility of information that ultimately reaches the invoked service is enhanced through outbound prioritization.

The Submission LQM provides a per-process service registration interface for inbound bandwidth management. This results in an equal sharing of inbound bandwidth resources per-process. The Submission LQM invokes an out-of-band RMI call to external SOA-clients to reallocate their bandwidth as needed. As with the aggregate policy distribution, we expect these reallocation calls will be infrequent compared to the service invocation and messages to services. Factors such as the duration of the connection lifecycle, frequency of connection failures and client request model for a particular SOA-deployment should be considered when determining an appropriate reallocation scheme.

The Dissemination LQM shown in Figure 4 provides managed dissemination by scheduling over differential queues. Queue counts coincide with the same number of bins used by the Task Manager. This modular design for managed differential dissemination can be used to schedule and send prioritized messages across outbound connections while meeting strict bandwidth requirements. QED uses differential queuing for outbound messages from services to clients, but the dissemination approach may also be applied to service-to-service communications in deployments where service-to-service messages span host boundaries.

As shown in Figure 4, the resulting "write-to-client" call from a service invocation is treated as a managed task. When outgoing messages are to be sent to a client, the Dissemina-

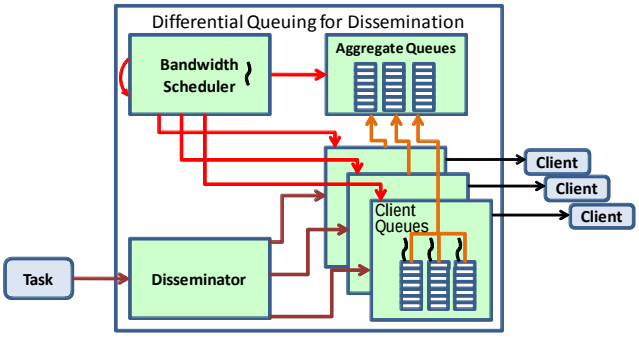

Figure 4. QED Dissemination LQM Design

tion LQM calculates the importance of the information for each receiving client, by checking the parsed policy held in attributes on state information for each connection.

After calculating the information importance the Disseminator component of the LQM will distribute the information to the appropriate ClientQueue. The ClientQueue calculates the priority from a combination of the provided importance and a *cost* measure based on the size of the information being disseminated (representing the amount of bandwidth sending the information will consume). At this point, the priority is used to determine which client bin should be used to enqueue the data.

The head of each ClientQueue bin is managed by a threaded class called the *ClientBinManager*, (shown here as a thread-line on the top of each ClientQueue bin). The ClientBinManager manages two operations for the head item of the queue. The first operation is an aggregate level enqueue and block. This ensures that each ClientQueue has only one piece of information allotted per bin that can be in contention for a chunk of the aggregate bandwidth. The second operation is unblock-and-send on signal which is triggered by the bandwidth scheduler upon selecting a particular client's priority bin. Through this mechanism the differential queuing allows for the fair scheduling across multiple client connections and priority bins.

The Bandwidth Scheduler, with operations shown with red lines, has a scheduling thread that alternates in a sleep/wake cycle based on the availability and use of bandwidth. When awakened, the scheduling thread selects the next dissemination task that should be processed. The scheduling algorithm provides identical support for the strict and weighted fair algorithms as described in Section III.C. The Bandwidth Scheduler calculates the amount of time to send the information by dividing the information size by the amount of available bandwidth. It then calls the callback of the selected task's ClientBinManager to notify its availability to send the information message. The send is immediately followed by a sleep for the amount of time calculated to send the information. At this point, the notified ClientBinManager removes the actual task from the appropriate bin and sends a message with the information to the receiving client.

E. Deployment and distribution of QoS managers

There should be a local task manager for each shared CPU resource, virtual or actual. This means that a host could have one local task manager that schedules operations running on that CPU, or it could have several local task managers, one each for the VMs running on the host with each VM having a specific "partition" of the CPU (e.g., controlled by the size of their available thread pool).

There should be a bandwidth manager for each occurrence of shared bandwidth, which could be associated with the NIC card on a host, a virtual private network, or dedicated network. True bandwidth management is only possible in those situations where the network is controlled by the QoS management services. Deployment of services across an unmanaged network (such as the Internet) will result in approximate and reactive QoS management only, since the amount of available bandwidth at any given time, the ability to control competition for the bandwidth, and honoring of network priorities (e.g., DiffServ Code Points) is beyond the QoS management service purview. Increased performance can be achieved in even these environments, however, through active monitoring of the bandwidth achieved between two points (e.g., by monitoring the latency and throughput of messages or using a tool such as TTCP [1]) and shaping and prioritizing traffic as if that is all the bandwidth available (leaving a reserve of unallocated bandwidth increases the delivered QoS predictability).

Likewise, there should be a submission and dissemination LQM for each occurrence of shared bandwidth used for incoming and outgoing messages, respectively. The aggregate QoS manager can be either centralized or distributed. If it is distributed extra care should be taken to synchronize the policy stores and policy distribution.

IV. Experimental Results

This section evaluates the QED capabilities in the context of a set of publication-subscription information management (IM) services shown in Figure 5. These services include (1) a Submission Service that receives incoming information, (2) a Broker Service that matches incoming information to registered subscriptions, (3) an Archive Service that inserts information into a persistent database, (4) a Query Service that handles queries for archived information, and (5) a Dissemination Service that delivers brokered information to subscribers and query results to querying clients. The baseline pub-sub IM services run on top of JBoss application server and JMS SOA middleware.

Below we present the results of three experiments conducted to evaluate the efficacy and performance of the pub-sub IM services with the QED QoS management services.

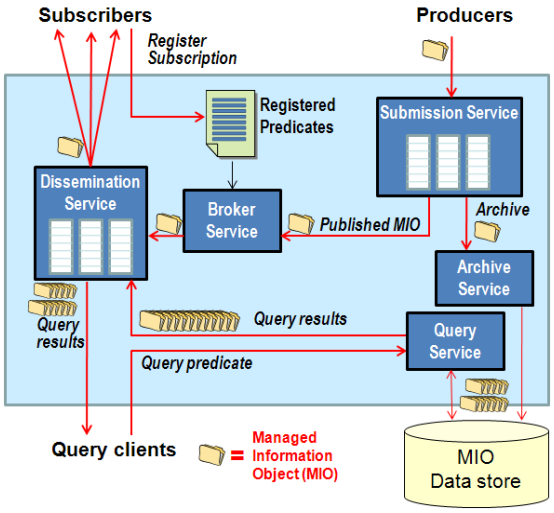

Figure 5. QED Information Management Services

We first measure the effect of CPU overload conditions on the servicing of information to demonstrate QED's differentiated services and then measure the effects of a shared bandwidth resource with high service contention and show how QED provides predictable service despite the contention (both experiments are contrasted with a baseline of the pub-

sub IM services without QED QoS management). Finally, we measure the performance of applying new policies to QED's QoS management infrastructure to evaluate QED's dynamism and scalability characteristics. The experiments indicate that QED can change policies dynamically and efficiently to handle many users and policy rules.

All experiments ran on ISISLab (www.isislab.vanderbilt.edu) using the Red Hat FC6 operating system over dual core 2.8 Ghz Xeon processors with 1 GB RAM and gigabit Ethernet (the Bandwidth Bound experiments required custom bandwidth limitation via the Linux kernel). Each experiment was conducted on three nodes: one for subscribers, one for publishers, and one for JMS and QED services.

A. *CPU Overload*

The first experiment evaluates QED's ability to provide differentiated service to important clients and information during CPU overload. The information brokering and query services are the most CPU intensive IM services. Each subscription or query has a predicate (specified in XPath or XQuery) that is evaluated and matched against the metadata of newly published (for information brokering) or archived (for query) information objects.

This experiment uses three subscribing and three publishing clients (one each with high, medium, and low importance), with each subscriber matching the information objects from exactly one publisher. To introduce CPU overload in this experiment, we created an additional 150 subscribing clients with unique predicates that do not match any published objects. These subscriptions create CPU load (in the form of processing many unique predicates) without additional bandwidth usage (since the predicates do not match any information objects, no additional messages are disseminated to subscribing clients). We then executed two scenarios: one in which all of the CPU load is caused by low priority information and the other in which CPU load is caused by all information (high, medium, and low importance).

In the first scenario, the high and medium importance publishers are publishing one information object each second (1 Hz), while the low importance publisher is publishing 300 information objects per second. The evaluation of the 153 registered predicates against the metadata of the two high and medium importance information objects is well within the capacity of the CPU, while the evaluation of the 153 registered predicates against the 300 low importance information objects (a total of 45,900 XPath/XQuery searches per second) is more than the CPU can handle.

Figure 6 shows a comparison of the number of high and medium importance information objects in the baseline IM services running over JBoss and the IM services running over JBoss with QED QoS management. The JBoss baseline does not differentiate the operations competing for the overloaded CPU and, as a result, only slightly more than half of the high and medium importance information gets through (.58 Hz for both high and medium publishers). The QED services, in contrast, used JBoss threads for brokering the more important information and, as a result, achieved a rate of .99 Hz for both the high and medium information publishers, nearly the full 1 Hz publication rate. The baseline JBoss

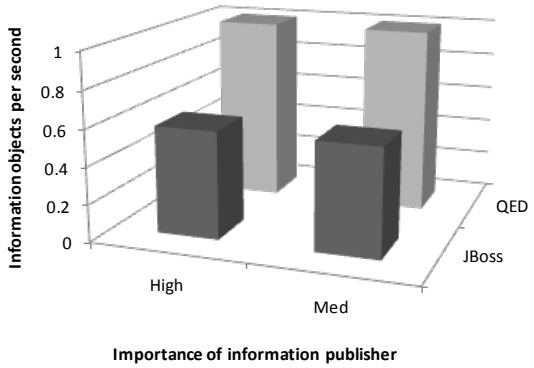

Figure 6. Differentiation Among High and Medium Importance Clients in CPU Overload Scenario

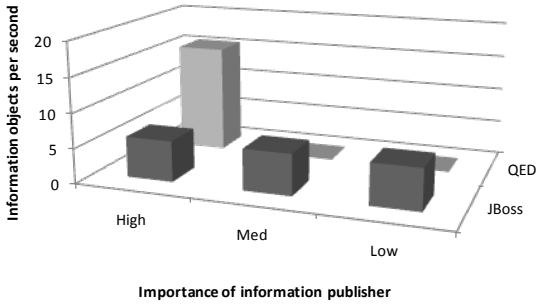

Figure 7. Differentiation in Second CPU Overload Scenario with Each Publisher Type Overloading the System with Information

system processes the low importance information at 16.28 Hz, while the JBoss system with QED services processes them at a rate of 13.59 Hz, which indicates there is significant priority inversion in the IM services running over the baseline JBoss, i.e., lower priority information is processed when there is higher priority information to process.

In the second scenario, all three publishers publish at a rate of 20 information objects per second (i.e., 20 Hz). This experiment overloads the CPU with predicate matching of information from high, medium, and low importance publishers, each of which is sufficient by itself to overload the CPU of our experiment host. Figure 7 shows how the IM services running on the baseline JBoss system exhibit no differentiation, processing almost equal rates of high, medium, and low importance information (5.9 information objects per second). In contrast, the QED services cause the IM services and JBoss to provide full differentiated service, with the high importance information being processed at the much higher average rate of 15.52 information objects per second. Meanwhile, medium and low importance information are not starved, and medium importance information is processed twice as often (0.2 Hz) as low importance (0.1 Hz).

B. *Bandwidth constrained*

Outgoing messages from the Dissemination Service to requesting clients and incoming messages to the Submission Service from publishing clients are the most bandwidth intensive of the IM services. This experiment forced a bandwidth bottleneck by constraining the shared bandwidth available from the Dissemination Service to all requesting clients to 320 Kbps. We then evaluated the ability of the IM services

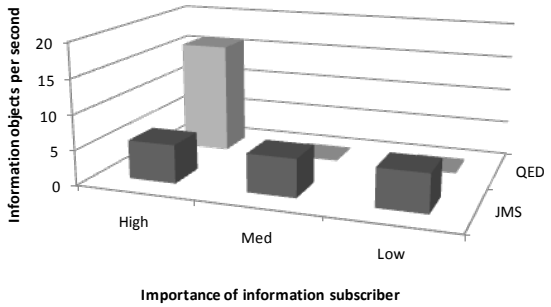

Figure 8. Subscriber Differentiation by QED during Bandwidth-constrained Operation

to use this constrained bandwidth for important outgoing traffic when utilizing the baseline JMS communication middleware and JMS with QED QoS management.

After constraining the outgoing bandwidth, we ran three publishers, publishing two information objects with a 1KB payload each second, and twelve subscribers, each with identical predicates that match all published information (i.e., all subscribers are interested in the data being published by all three publishers). This configuration ensured that the predicate processing (i.e., the CPU) is no longer a bottleneck. Each information object was delivered to 12 subscribing clients, resulting in over 576 Kbps of information trying to get through the 320 Kbps of available bandwidth.

Four of the 12 subscribers were set to high importance, four to medium importance, and four to low importance. Figure 8 shows that the IM services running on the baseline JBoss do not differentiate between the important subscribers and the less important subscribers, i.e., all subscribers suffer equally in JMS. The IM services running on JBoss with QED provides similar overall throughput but with better QoS to the subscribers that were specified as the most important.

C. *Policy change*

This set of experiments evaluated QED's dynamism and scalability, measuring (1) how quickly policy changes can be made and distributed to the LQM services in QED and (2) how the time to change policies scales with the number of users and existing policies. The first experiment measures the time to add and distribute a policy when the number of existing policies is 2, 10, 100, and 300. Figure 9 shows the time required to check the new policy against existing policies and apply the policy change scales well with the number of policies existing in the store. In fact, the slope of the line decreases as the number of existing policies increases.

The next experiment measures the time needed to add and distribute a policy as the number of client connections increases. We made a policy change with 2, 10, 100, and 500 client connections and measured how long it takes for the policy to take effect. The results in Figure 10 show that the time needed to effect a policy change scales well, with only subsecond time to effect a policy change even with several hundred connections.

Further testing showed that this linear trend continues when both large numbers of clients and existing policies exist at the same time, with the existing policies in the store being the primary bottleneck. QED's ability to quickly apply policy changes during run time adds dynamic control and responsiveness to the policy infrastructure.

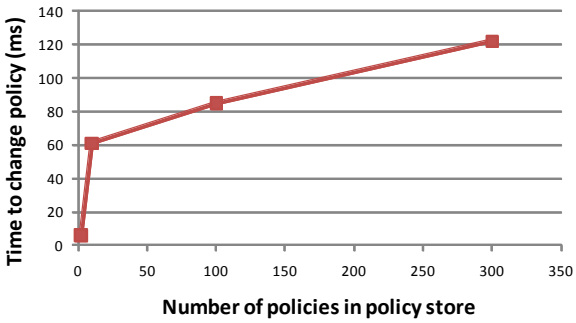

Figure 9. Time to Add a Policy Compared to # of Policies

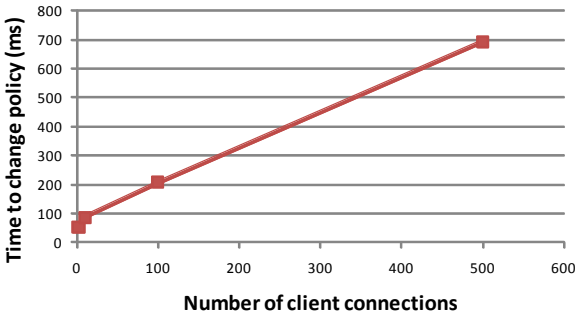

Figure 10. Time to Add a Policy Compared to # of Client Connections

V. Related Work

QoS management in middleware and SOA. Prior work focused on adding various QoS capabilities to middleware. For example, [8] describes J2EE container resource management mechanisms that provide CPU availability assurances to applications. Likewise, 2K [19] provides QoS to applications from varied domains using a component-based runtime middleware. In addition, [2] extends EJB containers to integrate QoS features by providing negotiation interfaces which the application developers need to implement to receive desired QoS support. Synergy [14] describes a distributed stream processing middleware that provides QoS to data streams in real time by efficient reuse of data streams and processing components. [13] presents an algorithm for composing services to achieve global QoS requirements. In [10], Lodi et al use clustering (load balancing services across application servers on distributed nodes) to meet QoS requirements for availability, timeliness, and throughput.

Network QoS management in middleware. Prior work focused on integrating network QoS mechanisms with middleware. Schantz et al. [15] show how priority- and reservation-based OS and network QoS management can be coupled with standards-based middleware to better support distributed systems with stringent end-to-end requirements. Gendy et al. [4][5] intercept application remote communications by adding middleware modules at the OS kernel space and dynamically reserve network resources to provide network QoS for the application remote invocations.

Schantz et al. [16] intercept remote communications using middleware proxies and provide network QoS for remote communications by using both DiffServ and IntServ network QoS mechanisms. Yemini et al. [18] provide middleware APIs to shield applications from directly interacting with complex network QoS mechanism APIs. Middleware frameworks transparently converted the specified application QoS requirements into lower-level network QoS mechanism APIs and provided network QoS assurances.

Deployment-time resource allocation. Prior work has focused on deploying applications at appropriate nodes so that their QoS requirements can be met. For example, [9][17] analyzed application communication and access patterns to determine collocated placements of heavily communicating components. Likewise, [3][6] have focused on intelligent component placement algorithms that maps components to nodes while satisfying their CPU requirements.

Our work on QED builds upon and enhances this prior work on QoS-enabled middleware by providing QoS for SOA systems that (1) works with existing standards-based SOA middleware; (2) provides aggregate, policy-driven QoS management; and (3) provides applications and operators with fine-grained control of tasks and bandwidth.

VI. Concluding Remarks

This paper described the QED approach to dynamic task and bandwidth management, aggregation of competing resource demands, and QoS policy-driven prioritization and scheduling strategies. Our prototype and experiments with an information management system show significant improvement in performance, predictability, and control over the baseline JBoss and JMS SOA middleware. Future versions of QED will feed monitored statistics, including interface usage, service execution, and QED internals such as priority queue lengths, into the LQMs to supplement the existing QoS management algorithms with feedback control and learning. We are also incorporating disruption tolerance to handle temporary client to service and service-to-service communication disruptions.

We learned the following lessons from our experience developing and evaluating QED over the past several years:

- SOA's abstractions and portability are at odds with providing traditional QoS since key platform-level details are hidden from applications and operators. The QED management layers are a step toward developing effective, and largely portable, abstractions for QoS concepts.
- Overall system QoS can be improved when individual control points in SOA middleware are coordinated. QED's QoS management works with the QoS features and configuration parameters emerging for SOA infrastructure, supplemented with dynamic resource allocation, scheduling, and adaptive control mechanisms.
- As SOA middleware infrastructure evolves, so must the QoS management capabilities to ease QoS policy configuration, QoS service composition, runtime behavior evaluation, and service deployment, which is all distributed in ever increasingly pervasive and ubiquitous computing environments. QED's policy-driven approach to QoS management strikes an effective tradeoff between fine-grained control and ease of use.

References

[1] Cisco, "Using Test TCP (TTCP) to Test Throughput," Doc. 10340.

[2] M.A. de Miguel, "Integration of QoS Facilities into Component Container Architectures," ISORC, 2002, Washington, DC, USA.

[3] D. de Niz, R. Rajkumar, "Partitioning Bin-Packing Algorithms for Distributed Real-time Systems," Journal of Embedded Systems, 2005.

[4] M.A. El-Gendy, A. Bose, S. Park, K. Shin, "Paving the First Mile for QoS-dependent Applications and Appliances," 12th International Workshop on Quality of Service, June 2004. Washington, DC, USA.

[5] M.A. El-Gendy, A. Bose, K. Shin, "Evolution of the Internet QoS and Support for Soft Real-time Applications," Proceedings of the IEEE, Vol. 91, No. 7, July 2003.

[6] S. Gopalakrishnan, M. Caccamo, "Task Partitioning with Replication Upon Heterogeneous Multiprocessor Systems," Real-Time & Embedded Technology & Apps. Symp., 2006, Washington, DC.

[7] G. Hardin, "The Tragedy of the Commons", *Science*, Vol. 162, No. 3859 (December 13, 1968), pp. 1243-1248.

[8] M. Jordan, G. Czajkowski, K. Kouklinski, G. Skinner, "Extending a J2EE Server with Dynamic and Flexible Resource Management," Middleware, 2004, Toronto, Canada.

[9] D. Llambiri, A. Totok, V. Karamcheti, "Efficiently Distributing Component-Based Applications Across Wide-Area Environments," 23rd IEEE Int'l Conf. on Distributed Computing Systems, 2003.

[10] G. Lodi, F. Panzieri, D. Rossi, E. Turrini, "Experimental Evaluation of a QoS-aware Application Server," Fourth Int'l Symp. on Network Computing and Applications (NCA'05), July 27-29, 2005.

[11] J. Loyall, M. Carvalho, A. Martignoni III, D. Schmidt, A. Sinclair, M. Gillen, J. Edmondson, L. Bunch, D. Corman, "QoS Enabled Dissemination of Managed Information Objects in a Publish-Subscribe-Query Information Broker," SPIE Conference on Defense Transformation and Net-Centric Systems, April 13-17, 2009.

[12] J. Loyall, A. Sinclair, M. Gillen, M. Carvalho, L. Bunch, A. Martignoni III, M. Marcon. "Quality Of Service In Us Air Force Information Management Systems," MILCOM, October 18-21, 2009.

[13] N. Mabrouk, S. Beauche, E. Kuznetsova, N. Georgantas, V. Issarny, "QoS-Aware Service Composition in Dynamic Service Oriented Environments," Middleware, Nov. 30-Dec. 4, 2009, Champaign, IL.

[14] T. Repantis, X. Gu, V. Kalogeraki, "Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems," Middleware '06, 2006, Melbourne, Australia.

[15] R. Schantz, J. Loyall, C. Rodrigues, D. Schmidt, Y. Krishnamurthy, I. Pyarali, "Flexible and Adaptive QoS Control for Distributed Real-Time and Embedded Middleware," Middleware, 2003.

[16] R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquier, J. Loyall, "An Object-level Gateway Supporting Integrated-Property Quality of Service," ISORC 1999, Los Alamitos, CA.

[17] C. Stewart, K. Shen, "Performance Modeling and System Management for Multi-component Online Services," Symposium on Networked Systems Design & Implementation, 2005, Berkeley, CA.

[18] P. Wang, Y. Yemini, D. Florissi, J. Zinky, "A Distributed Resource Controller for QoS Applications," IEEE/IFIP Network Operations and Management Symposium, 2000, Los Alamitos, CA.

[19] D. Wichadakul, K. Nahrstedt, X. Gu, D. Xu, "2K: An Integrated Approach of QoS Compilation and Reconfigurable, Component-Based Run-Time Middleware for the Unified QoS Management Framework," Middleware, 2001, London, UK.