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

Joseph P. Loyall,[1] Matthew Gillen,[1] Aaron Paulos,[1] Larry Bunch,[2] Marco Carvalho,[2] James Edmondson,[3] Douglas C. Schmidt,[3] Andrew Martignoni III[4], and Asher Sinclair[5]

[1]BBN Technologies, Cambridge, MA USA
[2]Institute for Human Machine Cognition, Pensacola, FL USA
[3]Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN USA
[4]The Boeing Company, St. Louis, MO USA
[5]Air Force Research Laboratory, Rome, NY USA

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 encapsulating platform-level details and complexities. Control of some details encapsulated by SOA middleware is necessary, however, to provide managed quality-of-service (QoS) for SOA systems that require predictable performance and behavior. This paper presents a policy-driven approach for managing QoS in SOA systems called QoS Enabled Dissemination (QED). QED includes services for (1) specifying and enforcing the QoS preferences of individual clients, (2) mediating and aggregating QoS management on behalf of competing users, and (3) shaping information exchange to improve real-time performance. We describe QED's QoS services and mechanisms in the context of managing QoS for a set of Publish-Subscribe-Query information management services. These services provide a representative case study in which CPU and network bottlenecks can occur, client QoS preferences can conflict, and system-level QoS requirements are based on higher level, aggregate end-to-end goals. We also discuss the design of several key QoS services and describe how QED's policy-driven approach bridges users to the underlying middleware and enables QoS control based on rich and meaningful context descriptions, including users, data types, client preferences, and information characteristics. In addition, we present experimental results that quantify the improved control, differentiation, and client-level QoS enabled by QED.

KEY WORDS: Service Oriented Architecture, Quality of Service, Information Management

1. Introduction

Information management (IM) services have emerged as necessary concepts for information exchange in net-centric operations [25, 41, 42], including Net-Centric Enterprise Systems (NCES) [10], which provide a set of services that enable access to—and use of—the Global Information Grid (GIG) [14]. The core concept of IM is an active information management model where clients are information publishers and consumers that communicate anonymously with other clients via shared IM services, such as publication, discovery, brokering, archiving, and querying [8, 18]. Information is published in the form of typed information objects (IOs) consisting of payload and indexable metadata describing the object and its payload. Consumers make requests for future (subscription) or past

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

(query) information using predicates (e.g., via XPath [51] or XQuery [52]) over IO types and metadata values.

Common IM services include brokering (i.e., matching future published IOs to subscriptions), archiving of IOs, querying for archived objects, and dissemination of IOs to subscribing and querying clients, as shown in Figure 1. This approach is similar in concept to other publish-subscribe middleware, such as the Java Message Service (JMS) [20], the Data Distribution Service (DDS) [35], CORBA Notification Service [36], or Web Services Notification (WS-N) [34]. In contrast to these other middleware packages, however, the core functionality of the IM process (e.g., submission, brokering, archiving, query, and dissemination) is encapsulated as *Service-Oriented Architecture (SOA)* middleware, i.e., as individual services that can be distributed, orchestrated in various patterns, dynamically composed, and implemented in various ways.

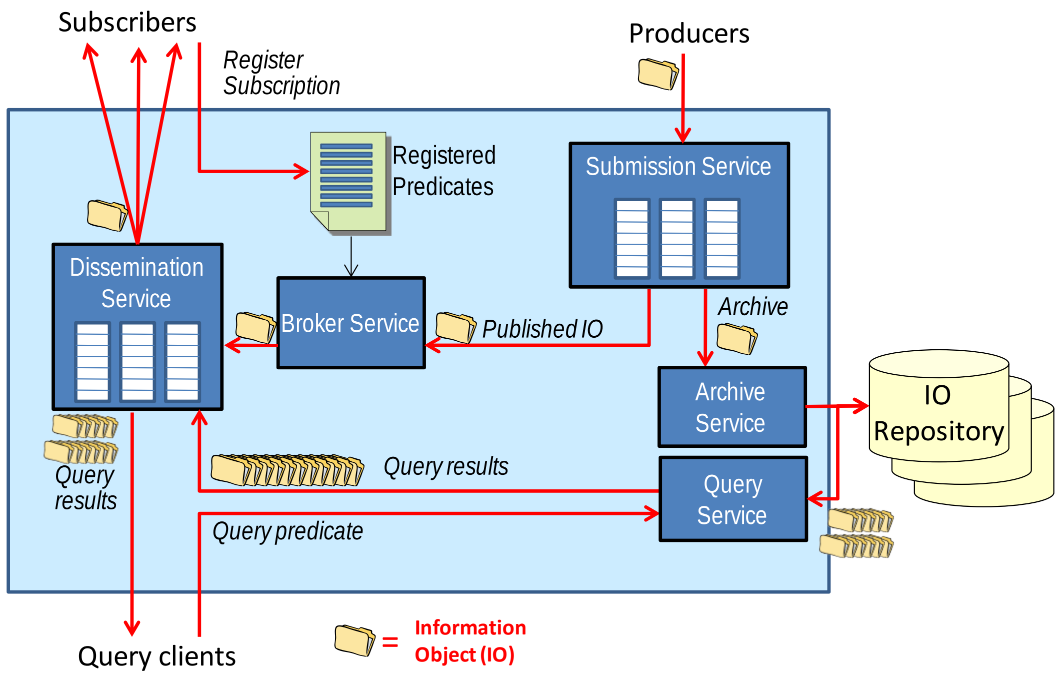


Figure 1. The Core IM services are *Submission* (of published information), *Archive*, *Brokering*, *Dissemination*, and *Query* services.

SOA middleware has emerged as a powerful environment for developing distributed applications and systems. It 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 upon which the system is run. While IM services are essential to support net-centric operations, the decoupled information management, brokering, and exchange they provide is insufficient for dynamic, mission-critical, and time-critical operations. In particular, IM services require *quality-of-service (QoS)* in their dissemination of information to meet varied quality requirements of users and the missions they undertake in a manner that is reliable, real-time, and resilient to the constrained, contended, and dynamically changing conditions in many environments.

QoS management is a key element of the design and runtime behavior of net-centric information systems, but it is often defined in terms of management of individual resources, e.g., the admission control provided by network management or CPU scheduling mechanisms or services. While individual resource management is necessary, it is not sufficient in net-centric information systems for the following reasons:

- *Effective QoS management spans individual resources.* Providing appropriate QoS to an information consumer includes managing the CPU required to process information (e.g., brokering), the network required to deliver information (e.g., submission and dissemination), and the memory required to store information (e.g., caching, queuing, and archiving).
- *QoS requirements are decoupled from QoS enforcement.* Information consumers specify their QoS requirements (e.g., the precision, rate, size, and content needed to meet their functional requirements), which may change over time. The points at which QoS enforcement occurs, however, are those at which information is created, processed, and disseminated, which can be distributed far away from consumers.
- *Management of individual resources implies a local view.* In contrast, effective end-to-end resource management must account for the aggregate requirements of the system's mission, the relative importance of resource users to the aggregate requirements, and what constitutes effective resource usage to achieve the aggregate requirements.
- *Effective QoS management must mediate the contention for resources from many simultaneously operating applications, streams of information, and systems.* The number and demand of users of resources will change dynamically and can be bursty. Mediation of this demand must frequently deny resources to some in favor of others, or no user might get sufficient resources, and should be made based upon which use benefits the overall goals of the system the most.
- *There might be multiple, simultaneous bottlenecks (i.e., the most constrained resources) and the bottlenecks might change over time.* Local enforcement to remove a single bottleneck might be ineffective if there are other bottlenecks present at the same time. Likewise, local enforcement to remove a bottleneck can simply expose or cause another bottleneck.

Management of resource consumption and system parameters are means available to help satisfy user and aggregate QoS requirements. Specifying, analyzing, monitoring, and providing QoS thus involves linking the platform and resource parameters, controls, and attributes to the broader aggregate and user-based requirements that they satisfy.

This paper describes our *QoS Enabled Dissemination (QED)* system, which provides a set of QoS services and mechanisms for SOA-based publish-subscribe middleware services that support applications and systems with predictable QoS requirements. Our previous work on QED has outlined QED's high-level architecture [28], a prototype implementation [29], and selected mechanisms and services [30]. This paper explores previously unexamined dimensions of QED, including its application to publish-subscribe IM services and experimental results that quantify its efficiency and efficacy.

This paper's main contributions include the following:

- An approach to *Service-Oriented QoS management*. This approach includes new QoS management services and QoS-enabled infrastructure services that make SOA suitable for applications with strict QoS requirements. QED exposes the controllable bottlenecks usually hidden behind SOA abstractions in a way that enables dynamic management of service execution and information dissemination for predictable QoS.

- *QoS management for decoupled information-centric systems.* A set of QoS-enabled publish-subscribe-query services that support an information-centric view of system construction. This view includes new approaches for aggregate QoS management based on policies over characteristics of decoupled clients, information, and operations; management of the fanout caused by dissemination to multiple consumers and multiple results to requests; and dynamic prioritization as the information demands and clients change.
- *Multi-layered, mission-driven QoS management.* QED provides a general-purpose QoS notation supporting mission-, user-, and client-level QoS policies based on actors, information, operations, and resources. QED's multiple QoS layers combine high-level policies, enforcement points, and enforcement mechanisms.

The remainder of the paper is organized as follows: Section 2 describes key QoS requirements in SOA-based IM services; Section 3 describes challenges associated with managing QoS for SOA-based IM services; Section 4 presents existing research and approaches; Section 5 describes how QED provides policy-driven QoS management services and mechanisms for SOA-based IM services; Section 6 analyzes the results of experiments we conducted to evaluate QED's effectiveness in providing QoS management in constrained and contended environments; and Section 7 presents concluding remarks and lessons learned.

2. Requirements for QoS in SOA-based Information Management Services

In theory, ideal QoS support for IM services would consist of information that leaves a publisher or archive and always reaches every consumer immediately, completely, and consistently. Moreover, every response to a request should convey the smallest amount of information that matches user needs exactly. In practice, this ideal QoS is unachievable since broker and retrieval processing—as well as network delivery—introduces delay into information dissemination. Likewise, resource failures and overload situations can induce information loss. Moreover, variations in the time to process requests, burstiness of client traffic, and competition with other processes can introduce variation into the system performance (jitter). Client requests for information do not always capture exactly the best qualities that they can use, and the attempt to capture these qualities frequently reduces the probability that they will match any published objects at all. Clients' demands on the IM services can also come into conflict, making it hard/infeasible to provide high QoS to one without reducing QoS to others.

QoS management services, such as QED, must manage the tradeoffs involved in providing higher aggregate levels of QoS across the users of IM services. Higher criticality operations should be provided preferential service. Loss and delay should be introduced where each can best be tolerated and the choice of which to introduce (when both are unavoidable) should depend on which is better tolerated.

Ideally, the pursuit of higher QoS should not introduce thrashing, i.e., higher but unsustainable QoS levels are not necessarily better than slightly degraded—but consistent—quality. The best information matches should be preferred, sometimes even over more information matches. Moreover, in those situations where not all information can be delivered, the information delivered should be the most important information to the most important users with respect to the goals of the overall system requirements.

The following are key dimensions of perceived quality and some ways that IM services can affect them:

- *Timeliness* – The speed at which brokering happens or the latency through the IM system, e.g., how fast a published IO reaches subscribers and how fast query responses reach the querying client. In general, the greater the timeliness (and conversely, the lower the latency), the better the perceived QoS.
- *Completeness* – The number of IOs that reach requesters from those available (i.e., published into the IM services) that are relevant (i.e., that match requests).
- *Fidelity* – A measure of the amount of information in each individual IO. Fidelity concerns whether a requester receives the entire amount (metadata and payload) of a published IO.
- *Accuracy* – A measure of the correctness of information delivered, i.e., whether IOs delivered to a requesting client have any errors introduced into them (e.g., during transformation, brokering, or other operations).
- *Smoothness* – The predictability of performance and consistent latency. In many cases, perceived QoS is higher if a user receives a consistent and expected quality than if high quality is interspersed with low quality, to the point where there is a wide variation and a user cannot know what to expect.
- *Suitability* – The better a response matches a user's needs, the higher the perceived QoS. This means that higher resolution and higher precision IOs are generally recognized as higher QoS than lower resolution or precision. Other characteristics of IOs, such as source, currency, content, trust, format, and so forth, can make them more or less suitable for a given request and therefore affect their perceived QoS.

Table 1 describes how QoS can be affected by each IM service shown in Figure 1. These IM services can adversely affect each of the QoS requirements (and therefore need QoS management) with the exception of fidelity; there are no IM operations in the core IM services that specifically affect the fidelity of individual IOs. Likewise, there is nothing in the IM services that specifically affects the accuracy or suitability of IOS, although certainly errors in IOs could be introduced accidentally as the result of errors in implementation. Moreover, suitability can be affected by the default semantics of the IM service implementation which return IOs in the order they are published into the system and returned by the database, even if they are less suitable than something else currently queued for brokering or dissemination.

3. **Challenges in Providing QoS in SOA-Based Information Management Services**

This section describes the QoS management challenges arising from the use of high-level abstractions in SOA infrastructure and IM services.

3.1 QoS Management Challenges Due to SOA Infrastructure

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. Service interfaces in SOA-

Table 1. Resources Used and QoS Affected by IM Services

Control Point	Description	Resources Used	QoS affected
Submission Service	Receiving IOs entering the system as the result of publishing.	Network bandwidth; memory to store objects until they are processed.	*Timeliness, completeness:* If the rate, number, and size of published objects exceed the bandwidth capacity it will introduce delay or loss. If the rate of published objects exceeds the rate at which they can be processed, delay will be introduced as objects are enqueued awaiting processing.
Brokering	Predicate evaluation to match published IOs with registered subscriptions.	CPU	*Timeliness, smoothness:* Introduces latency; calls to the predicate evaluator can take differing amounts of time, depending on the size and complexity of the metadata and predicates, thereby introducing jitter.
Archiving	Insertion of a published IO into the IO Repository	CPU to process archive. Disk space to store IO.	*Timeliness:* Introduces latency.
Query processing	Evaluation of a query operation and subsequent retrieval of results.	CPU to process the query operation. Memory to store the results (potentially many).	*Timeliness, smoothness:* Introduces latency. Queries can take differing amounts of time and result in different size result sets, thereby introducing jitter.
Dissemination	Delivery of the results of brokering (a single IO) to matched clients (potentially many). Delivery of the results of a query (potentially many IOs) to the requestor (a single querying client).	Memory to store the IOs being delivered. Bandwidth to deliver the IOs.	*Timeliness, completeness, smoothness:* If the rate, number, and size of IOs exceed the amount of bandwidth capacity available to send them, delay or loss will be introduced. Since IOs will vary in size, they will take different amounts of time and bandwidth to send, introducing jitter.

based systems are specified in standard interface description languages (IDL), such as the Web Services Description Language (WSDL), which represents an evolution of the IDLs used for components and distributed objects that preceded the SOA paradigm.

SOA-based systems also typically include an execution container model and support for inter-service communication, e.g., provided by an Enterprise Service Bus (ESB). Like the component middleware that preceded it, SOA middleware 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.

Although middleware abstractions, such as those in SOA, simplify developing, composing, and executing applications, they incur challenges for managing application QoS.

We describe these challenges in the following paragraphs in terms of SOA, but they are common to infrastructure middleware with similar aims as SOA.

SOA Infrastructure Challenge 1 – SOA infrastructure abstracts away the visibility and control necessary to manage QoS. For example, JBoss, which is an open-source implementation of Java 2 Enterprise Edition (J2EE), includes an *Application Server (AS)* that provides a container model in Java for executing services. The JBoss AS container hides runtime- and platform-level details useful for QoS management, including 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 over-utilization) 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.

As another example, 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, but does not expose any queue or flow control, 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.

SOA Infrastructure Challenge 2 – Many QoS parameters and configuration options are low-level whereas QoS requirements and the SOA concepts they work within are higher level. Many SOA implementations provide certain QoS parameters and configuration choices that are useful, but not sufficient, to support higher-level QoS provisioning. For example, JMS provides hints that allow JMS implementations to support QoS via three parameters: delivery mode (persistent or non-persistent), priority, and time-to-live. There is little/no 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. Likewise, 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, where 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 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 5 provide this support.

3.2 QoS Management Challenges Due to IM Service Middleware Abstractions

While SOA environments present challenges for QoS management, IM services provide another layer that incurs additional challenges by encapsulating low-level details and presenting an information-centric abstraction (based on a publish-subscribe-query communication model) that enables the discovery and dissemination of information based on topic and content. This abstraction hides the details of the processing needed for brokering, the bandwidth needed for transmitting and disseminating information, the memory needed for storing and caching information, and the contention for these resources between decoupled clients (i.e., with no fixed connections), varying data rates and sizes, and varying numbers of users.

Providing QoS management in IM services is similar in spirit to traditional operating system scheduling problems (e.g., the scheduling of tasks that use CPU, such as brokering and query processing, or tasks that use bandwidth, such as dissemination), and network flow control problems, (e.g., optimizing the flow of IOs through the information broker). The following significant differences, however, make QoS management in IM services and the publish-subscribe-query operations they support even more challenging:

- ***IM Service Challenge 1 – Data size is not uniform.*** Network routers can assume a maximum transmission unit, which is a relatively small (and sometimes fixed) packet size. In contrast, IOs can differ significantly in size and complexity, both in metadata and payload. This diversity makes it hard to estimate the resource usage (e.g., bandwidth to send and memory and disk space to store) and time for processing *a priori*.
- ***IM Service Challenge 2 – Processing can vary significantly in time.*** Predicate evaluation and query processing are affected by the number of predicates and subscribers; predicate complexity; and the size and complexity of metadata.
- ***IM Service Challenge 3 – The destination of information is unknown when information enters the system.*** There is no opportunity for end-to-end QoS management or prioritized treatment based on the importance of information destination or knowledge of its use.
- ***IM Service Challenge 4 – There are two occurrences of fanout.*** A single published IO can match multiple subscribers and a single query can result in many IOs to deliver. This fanout means that the downstream resource usage (e.g., during dissemination) of a published IO or a query is hard to predict and take into consideration in upstream management (e.g., during submission, brokering, and query processing).

4. Related Work

Previous research in QoS management strategies for distributed systems has primarily involved dynamic or pro-active allocation of resources for service provisioning, as well as the application of QoS capabilities at different levels of the infrastructure. This section presents prior work in several related areas.

QoS management in middleware and SOA. Prior work focused on adding various QoS capabilities to middleware. For example, [24] describes J2EE container resource management mechanisms that provide CPU availability assurances to applications. Likewise, 2K [49] provides QoS to applications from varied domains using a component-based runtime middleware. In addition, [11] extends EJB containers to integrate QoS features by providing negotiation interfaces which the application developers need to implement to receive desired QoS support. Synergy [37] describes a distributed stream processing middleware that provides QoS to data streams in real time by efficient reuse of data streams and processing components. [31] presents an algorithm for composing services to achieve global QoS requirements. In [27], Lodi et al use clustering (load balancing services across application servers on distributed nodes) to meet QoS requirements for availability, timeliness, and throughput.

The DCBL Middleware [47] uses a reinforcement learning approach combined with a control mechanism to improve and adapt QoS for a set of heterogeneous applications in a dynamic environment. G-QoSM [1] describes a mechanism for QoS management in a grid computing architecture by reserving some of the system capacity for utilization for certain classes of operations if there is resource failure or congestion. In [5] and [54], the authors show genetic algorithm based approaches for QoS-aware selection and composition of web-services and discrete operating modes for pre-defined QoS allocations.

Publish-subscribe middleware. Eugster et al provide an overview of the pub-sub interaction pattern, highlighting the decoupled nature of publishers and subscribers in time, space, and synchronization [4]. A few researchers have investigated QoS management in pub-sub middleware. Mahambre et al present a taxonomy of pub-sub middleware with QoS features, acknowledging significant gaps in the provision of QoS features to the extent that some environments need [32]. Likewise, a comparison of two common pub-sub systems, the Java Message Service (JMS) and the Data Distribution Service (DDS), focusing on their QoS features, is provided in [9]. For those pub-sub systems with QoS parameters, Hoffert et al provide machine learning techniques for configuring the QoS parameters [22]. Behnel et al provide a set of QoS metrics described in the context of publish-subscribe systems [3].

The research described in this paper improves on these pub-sub systems in the following ways. First, while the systems are similar in their loose coupling of information producers and information consumers, many of the existing pub-sub systems simply provide an interface for disseminating information, rather than the rich, active information management of the IM services. In some cases, this limits the matching to “topics”, which is a coarse division of all information into logical groups based on a shared topic name. In contrast, the IM services that we describe support pluggable brokering that can match over rich sets of metadata using predicate languages like XQuery and XPath. Many existing pub-sub middleware implementations also only support future information in the form of subscriptions. That is, consumers register interest in a topic and get information for that topic that is published *from that point onward.* There is little or no support for archival of information and matching and retrieval of past information without programming the support outside the pub-sub middleware.

Second, while several of the existing pub-sub middleware systems provide QoS parameters, they do not, in general, support the aggregate, dynamic, and policy-driven QoS that the QED system provides. For example, DDS’s QoS parameters apply to a particular connection, even though multiple connections can contend for shared bandwidth. The other pub-sub systems expose the QoS parameters to the application layer, but do not provide a language for expressing the QoS requirements based on application-level concerns, i.e., users, operations, information, and resources. They also do not provide the means for deconflicting inconsistent or competing QoS requirements.

Network QoS management in middleware. Prior work focused on integrating network QoS mechanisms with middleware. Schantz et al [39] show how priority- and reservationbased OS and network QoS management can be coupled with standards-based middleware to better support distributed systems with stringent end-to-end requirements. El-Gendy et al [15, 16] 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. In [38], the authors proposed a statistical traffic signature method to identify different classes of service for QoS management.

Schantz et al [40] intercept remote communications using middleware proxies and provide network QoS for remote communications by using both DiffServ and IntServ network QoS mechanisms. Wang et al [48] 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. Other prior work has focused on deploying applications at appropriate nodes so that their QoS requirements can be met. For example, [26, 45] analyzed application communication and access patterns to determine collocated placements of heavily communicating components. In [6], the authors focus on the service pre-allocation problem in the context of multi-media applications, for distributed indexing management. The approach relies on pre-allocation of resources for index management and load distribution. Likewise, [13, 17] have focused on intelligent component placement algorithms that map components to nodes while satisfying their CPU requirements.

Model-based design tools. Prior work has also been done on model-based design tools for specifying and enforcing QoS in middleware and applications. For example, PICML [2] enables distributed real-time and embedded system developers to define component interfaces, their implementations, and assemblies, facilitating deployment of Lightweight CORBA Component Model (CCM)-based applications. VEST [44] and AIRES [19] analyze domain-specific models of embedded real-time systems to perform schedulability analysis and provides automated allocation of components to processors. SysWeaver [12] supports design-time timing behavior verification of real-time systems and automatic code generation and weaving for multiple target platforms. In contrast, NetQoPE provides model-driven capabilities to specify network QoS requirements on DRE system application flows, and subsequently allocate network resources automatically using network QoS mechanisms. NetQoPE thus helps assure that application network QoS requirements are met at deployment-time, rather than design-time or runtime.

QoS policy languages. XACML [33] is an OASIS standard designed to address a subset of QoS—access control policy—which is a subset of security (itself one property of QoS). XACML specifies several roles for policy authoring, decision making, and enforcement, including a Policy Decision Point (PDP) and Policy Enforcement Point (PEP). XACML specifically states that the PEP initiates decision requests, i.e., an entity tries to gain access to a resource through a PEP and the PEP then asks the PDP to evaluate current policies and provide a response by which the PEP can either grant or deny access. This approach fits the local scope of access control, i.e., control is granted to a local resource, but not that of broader QoS properties. Aggregate QoS management requires higher-level (i.e., incorporating aggregate- and application-level priorities and context) and broader (i.e., encompassing many PEPs) scope. Managing access to, and use of, a single resource may or may not contribute to overall application, aggregate, or mission-level QoS, and is inadequate by itself to provide application, aggregate, or mission-level QoS. The PEP-

PDP dataflow in the XACML specification language is thus not sufficient for the general context aware QoS case. Providing QoS requires the coordination of many PEPs, based on higher-level, aggregate and mission-based policies, and therefore cannot be initiated solely by PEPs. Instead policies must be evaluated at PDPs when needed and the results of the policy evaluation should be pushed to the PEPs, which then enforce it. This approach provides part of the coordination needed since a PDP that makes an aggregate QoS decision will push consistent results to the PEPs, which then ensures consistent, coordinated QoS management if the PEPs enforce the policy decision as they are directed.

As described in the remainder of this paper, our work on QED builds upon and enhances prior work on QoS-enabled middleware and model-based tools by providing QoS for SOA-based IM services 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. In addition, QED decouples policy selection, parsing, and dissemination, which constitute the aggregate decision making and occur on discrete epochs, such as when users come and go or resource availability changes, from policy enforcement, which can be done in-line with processing and information dissemination, where resources are actually consumed, but which can happen much more frequently and rapidly.

5. **QoS Management Capabilities for SOA-Based IM Services**

This section describes how we developed the *QoS-Enabled Dissemination* (QED) middleware to meet the requirements for QoS in SOA-based IM systems described in Section 2 and address the challenges presented in Section 3. The QED middleware contains the layers shown in Figure 2 and described below:

- An *aggregate QoS management* layer that manages overall QoS policies, mediates conflicting demands for QoS across clients of the IM services, and monitors delivered QoS for the overall system;
- A *local QoS management* layer that manages QoS policies at individual policy enforcement points; and
- A *QoS enforcement mechanism* layer with specific QoS enforcement mechanisms.

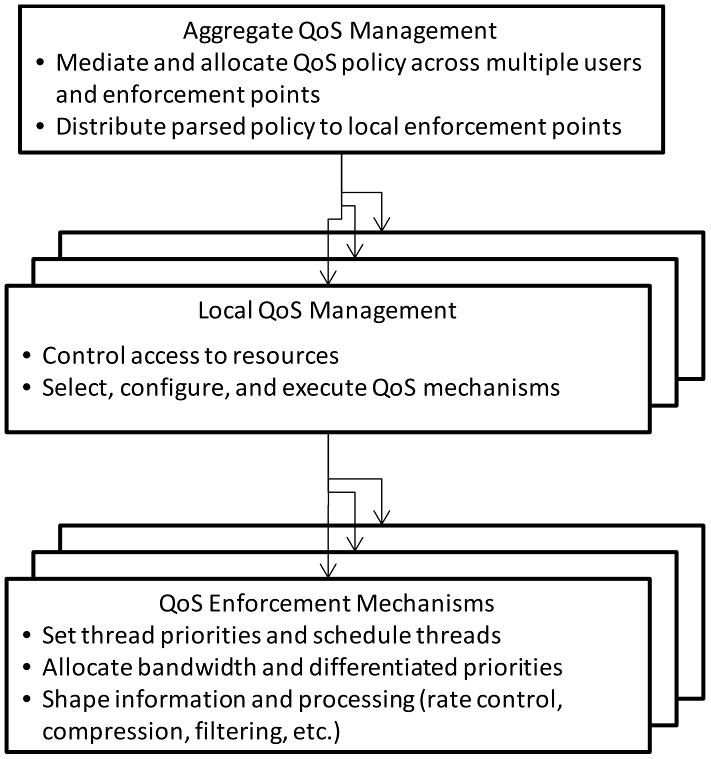

Aggregate QoS Management
• Mediate and allocate QoS policy across multiple users and enforcement points
• Distribute parsed policy to local enforcement points
Local QoS Management
• Control access to resources
• Select, configure, and execute QoS mechanisms
QoS Enforcement Mechanisms
• Set thread priorities and schedule threads
• Allocate bandwidth and differentiated priorities
• Shape information and processing (rate control, compression, filtering, etc.)

Figure 2. QED's SOA-based Layered Architecture.

QED's policy-driven QoS mechanisms ensure the *timely* brokering and dissemination of important information through the IM services, ensure *smoothness* of information brokering and dissemination, make tradeoffs to favor *completeness* or *fidelity* when systems are overloaded, and enforce client preferences and priorities to increase *suitability*.

We have implemented prototype versions of the QED architecture based on the JBoss Application Server [23] and the Spring application framework [43]. These prototype implementations of QED include two Local QoS Managers (LQMs): (1) a *Task Management LQM* that manages access to the CPU by scheduling CPU intensive tasks, such as brokering, by managing the size of the thread pool, assigning threads to processing, and setting thread priorities, and (2) a *Dissemination Service LQM* that manages access to shared network bandwidth by prioritizing outgoing IOs based on importance, size, and policy. Likewise, the QED prototype implementations include the following QoS mechanisms: (1) *differentiated queues* that prioritize operations and dissemination tasks, (2) *IO shaping* that reduces the amount of bandwidth consumed and matches client preferences, utilizing compression, cropping, scaling, filtering, and transformations (using XSLT [53]), and (3) *partitioned predicates* that group and prioritize registered predicates to enable finer grained control over the time to broker IOs.

With these capabilities, QED addresses the challenges of providing QoS in SOA and IM systems identified in Section 3 in the following ways:

- It addresses SOA Infrastructure Challenge 1 by managing the scheduling of threads and bandwidth typically hidden behind SOA interfaces.
- It addresses SOA Infrastructure Challenge 2 by providing and enforcing QoS policy that is specified at a high level, based on user, information, and system concepts.
- It addresses IM Service Challenges 1 and 2 through the use of binned queues, bandwidth estimation, and task management to handle information that varies in size and processing that varies in time.
- It addresses IM Service Challenge 3 by providing policy-driven aggregate QoS management across control points and across users, despite the publishing and consuming users being decoupled from one another.
- It addresses IM Service Challenge 4 by separately scheduling the brokering of requests and dissemination of information, so that fanout is managed. Important requests are brokered before less important requests, and results are delivered to important consumers in preference to delivery to less important consumers.

The QED prototype also includes a Web Services-based *QoS administration interface* that supports entry/selection of QoS policies and monitoring of delivered QoS, as well as a *QoS monitoring service* that collects statistics on resource usage and delivered QoS for feedback into QoS managers and for visualization through QoS administration interface.

The remainder of this section describes the structure and functionality of the QED capabilities shown in Figure 3 (i.e., aggregate QoS management, QoS policy, task management local QoS management, and bandwidth management) and explains how these capabilities address the challenges described in Section 3.

5.1 Aggregate QoS Management

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.

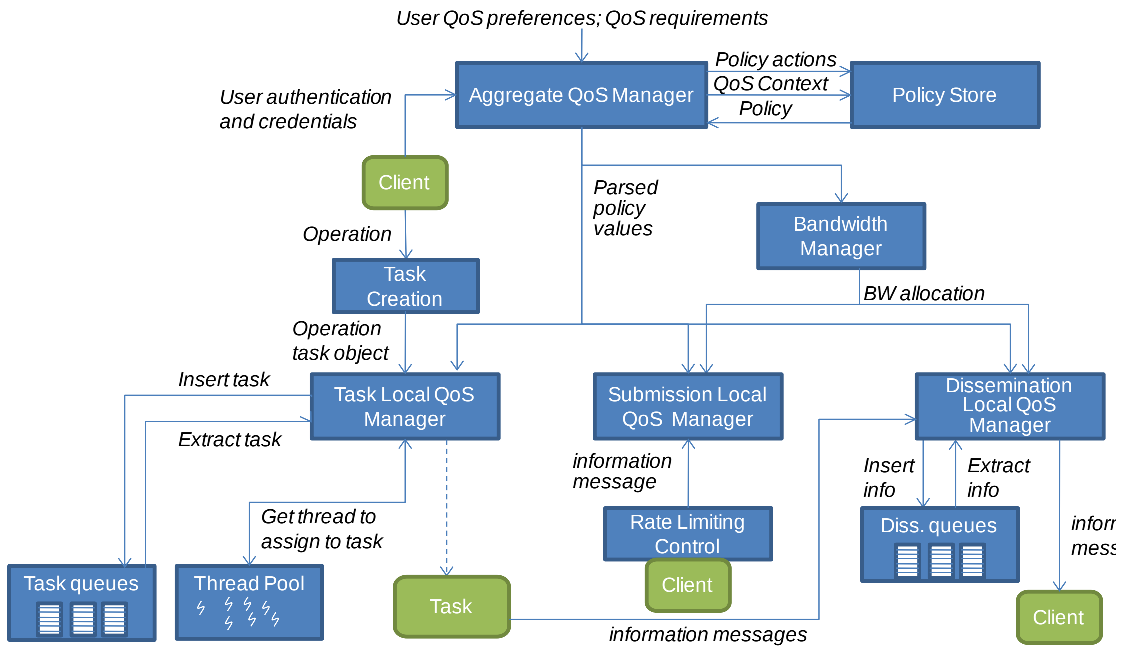


Figure 3. QED Aggregate QoS Management Capabilities for SOA-based IM Services.

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* [21]). 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. This design addresses the IM Service Challenge 3 to support consistent management of QoS from publication through brokering and dissemination.

When a client is authenticated (using an authentication service) to gain access to services, 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 messages), and resources, which addresses the SOA Infrastructure Challenge 2 by mapping the high-level QoS policies to lower-level QoS controls. The actionable policies distributed to local QoS managers can be checked quickly at local enforcement points via attribute lookups and relational comparisons so they can be applied in the control and data flow path. 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 overall distributed system, such as the entry of new clients, resource failure, and changes in overall QoS goals or policies.

5.2 QED QoS Policies

QED's QoS Policy specification language enables users to describe system contexts via meaningful, domain-specific concepts (such as subscriptions to imagery by ground rescue crews) and map these contexts to QoS concepts (such as the relative importance of fulfilling a request and the types of information filtering and shaping that are desirable for that context). Through its high-level specifications and its mapping to QoS concepts, QED QoS policy language helps address SOA Infrastructure Challenge 2. QoS policies in QED are independent of the underlying IM service implementation. These policies are also formal and readily accessible by software for reasoning and enforcement, as described below.

QoS policy definition. Each QoS policy in QED defines a mapping from the conditions in which the policy applies to the effects of the policy, i.e., *QoS Policy = System Context (O,M,E) → QoS Settings (v,i,P)*. QoS policy conditions describe a *System Context* in terms of the properties that can be observed about the system behavior and state, represented by a Boolean function over any or all of the properties of the operations (*O*), information (*M*), and entities (*E)* involved. Zero or more attributes from each category may be used in a policy's *System Context* definition.

QED's Policy specification language enables system contexts to range from the most general 'default' context (e.g., *any operation by any user on image types*), to specific contexts such as *analyst publishing rescue images.* The system context is extended to include *Resources*, such as queue lengths, CPU, and bandwidth, by introducing a resource monitoring component, *g(R)*, that adds and removes sets of policies based on monitored resource states such that *g(R) → (f(O,M,E) → (v,i,P)).*

The effects of the policy describe the desired *QoS Settings* for the given system context. In the *QoS Settings* the precedence level, *v*, is required and aids in selecting between conflicting policies; higher precedence policies are enforced in favor of lower precedence ones. By convention, policy precedence differs based on the policy source, such as administrator-level policies that can override policies from a less privileged user. Among policies from sources with equal precedence, higher precedence is assigned to those with more specific system context descriptions.

The importance, *i*, is an optional domain-specific measure of the relative value of an operation, type, or client to an overall system's goals. This value is used, along with other factors such as cost, to prioritize processing and dissemination of information.

QoS preferences, *P*, define a named set of limits and tradeoffs among aspects of QoS, such as deadlines for delivering IOs through the IM services and the ranges of information filtering and shaping allowed. Zero or one *QoS Preference Sets* may be included in a policy and each QoS preference set includes one or more name-value pairs from a predefined list of preference names. It is here that we capture aspects such as GPS track information being replaceable in most contexts where only the most recent position is useful.

The QED QoS policy specification language is implemented in the KAoS policy framework [46] using the extensible OWL semantic web language [50]. KAoS provides a generic construct for *obligation policies* that maps a context description to a desired action or state. Within this core policy construct, we extended the OWL ontology of policy concepts available to KAoS to include new hierarchies of domain-specific classes of IM operations, information types, and users including roles and groups.

Figure 4 graphically depicts the OWL classes used by the KAoS-based implementation, the properties defined for these classes, and the range of each property. The *CreateServiceOrchestration* shown in this figure is an abstract action that represents the start of any new session for a client such as creating a subscription. This action class defines properties for each of the observable attributes of this type of action including the orchestration type and the information type. Policies are defined in terms of restrictions over the properties of the action. When a component performs a policy check, it creates an instance of the action that is then checked against the restrictions in the policy to determine whether the policy applies to the given instance.

The *orchestrationType* is an OWL object property with a range over the OWL *OrchestrationType* class, with instances *Publish*, *Subscribe*, and *Query* for the IM service context in which the QED QoS policies are defined. In QED, an *orchestration* represents an aggregate composition of services over which policies apply. For example, the IM *Publish* orchestration instance refers to a composition of the *Submission*, *Brokering*, *Archive*, and

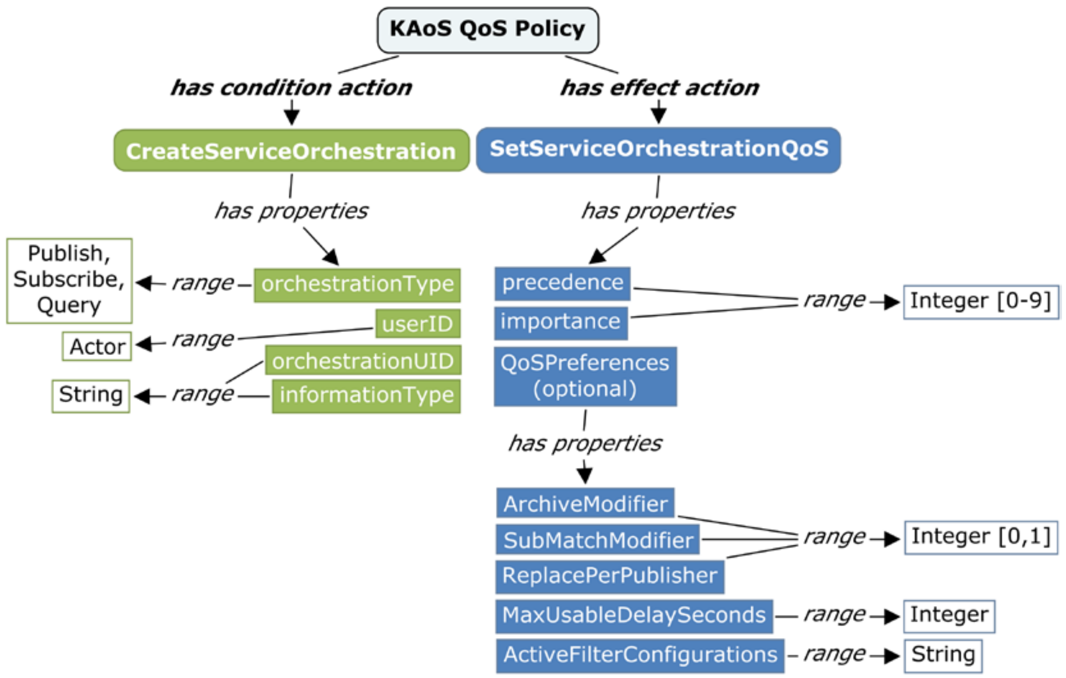


Figure 4. Representation of OWL Ontology Describing Service Orchestration Creation Context, QoS Settings, and Attributes of QED QoS Preferences.

Dissemination services needed to process a published IO, as shown in Figure 5.

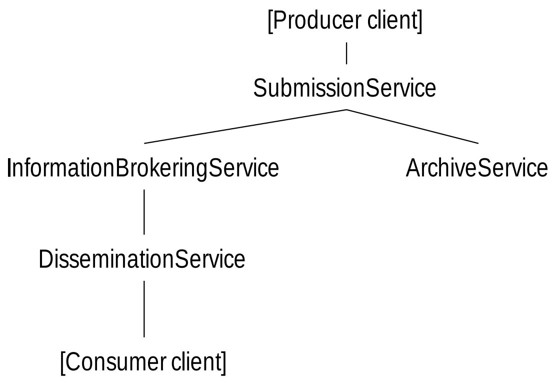


Figure 5. The IM Publish Orchestration

Each IM service includes an object, called a *Context*, that holds information about the service's state. The properties of a service orchestration therefore hold all the observable information about the entity (i.e., the *actor*), operation, and information that makes up the condition of a QoS policy rule. QED uses orchestration instances to store parsed policies and to disseminate policies to the local enforcement points, as described below. The *orchestrationUID* is a string property containing the unique ID of an orchestration instance associated with a client's connection to the IM services.

The *informationType* is a domain-specific identifier for the type of information, such as *MapImage* or *ATODocument*. Regular expressions can be used in policies to define the applicable range of values. The *userID* is an OWL object property ranging over the class of Actors which includes any Roles or Groups defined in the ontology.

Matching the condition of a QoS policy rule *obligates* the setting of QoS attributes, i.e., the *effect* of the QoS policy. The QoS Settings are represented by an operation *SetServiceOrchestrationQoS* which specifies setting the following attributes on the context objects in the appropriate Service Orchestration instance:

- *Precedence*, which defines an integer ordering used to determine how to resolve ambiguities in overlapping policies. Higher precedence policies can override policies with lower precedence values so administrators can define wide-ranging policies that cover most cases and then make specific exceptions to override the base case. For example, allowing compression before disseminating any images could be the default case, overridden by a higher precedence policy for specific image types and operations for which the fidelity must be maintained. QED currently assigns a precedence automatically based on the source of the policy (e.g., Administrator vs. User) followed by the specificity of the rule criteria. More specific policies are, by default, given preference in comparison with more general policies.
- *Importance*, which is an integer value representing the relative importance of information in a given context to the overall success of the mission or operation.
- *qosPreferences*, which is a set of constraints on QoS behaviors that can be used to determine how to best adjust the performance of a client's information flow in the face of resource bottlenecks. This OWL object property defines a range over the class of *QoSPreferences*, which in turn contains the following properties: (1) *MaxUsableDelaySeconds*, which is the integer number of seconds of delay in the IM services after which the information is no longer useful in the given context (a value of 0 indicates that there is no maximum delay, i.e., indefinite delay), (2) *ReplacePerPublisher* (range: 0,1), which is a Boolean indication of whether an IO queued in the IM should be replaced (dropped) if a new IO from the same publisher arrives, (3) *ArchiveMo-*

difier/*SubMatchModifier*, which are integers between -1 and 1 weighting the tradeoff between subscription matching and archival where a higher value indicates higher desired QoS for one over the other, and (4) *ActiveFilterConfiguration*s, which is a list of string values that identifies the shaping operations that QED can perform.

QoS Policy Management and Decision Making. QED includes a *Policy Store* component, as shown in Figure 3, that maintains the set of policies available and provides transactional management of policy storage, retrieval, parsing, deconfliction, and dissemination. The primary role of the Aggregate QoS Manager is to manage the overall QoS policies for the information services and their users and disseminate the appropriate policies to the points at which they can be enforced. It reasons about the applicable set of policies associated with users, their information type and operations, and the resources in the system. It also manages the distribution of the appropriate policies to the local enforcement points, as described below. The QED Aggregate QoS Manager performs policy checking and conflict resolution when Submission and Dissemination Services create a service orchestration and the resulting QoS Preferences are set as attributes on the appropriate context object, as shown in Figure 6.

QED includes two implementations of the Policy Store: one based on the KAoS policy services and another implemented with *Plain Old Java Objects* (POJOs). The KAoS version includes a step to translate policies into OWL, uses the features of the KAoS Directory Service to store and retrieve policies, and includes a step to parse OWL-specified policies into Java classes. This design enables the KAoS version to use ontology classes to create policies concerning classes such as user groups and roles and abstract information types, such as all images. The POJO version stores the policies in the Java classes directly and provides Java methods to store and retrieve policies. The POJO version is limited to

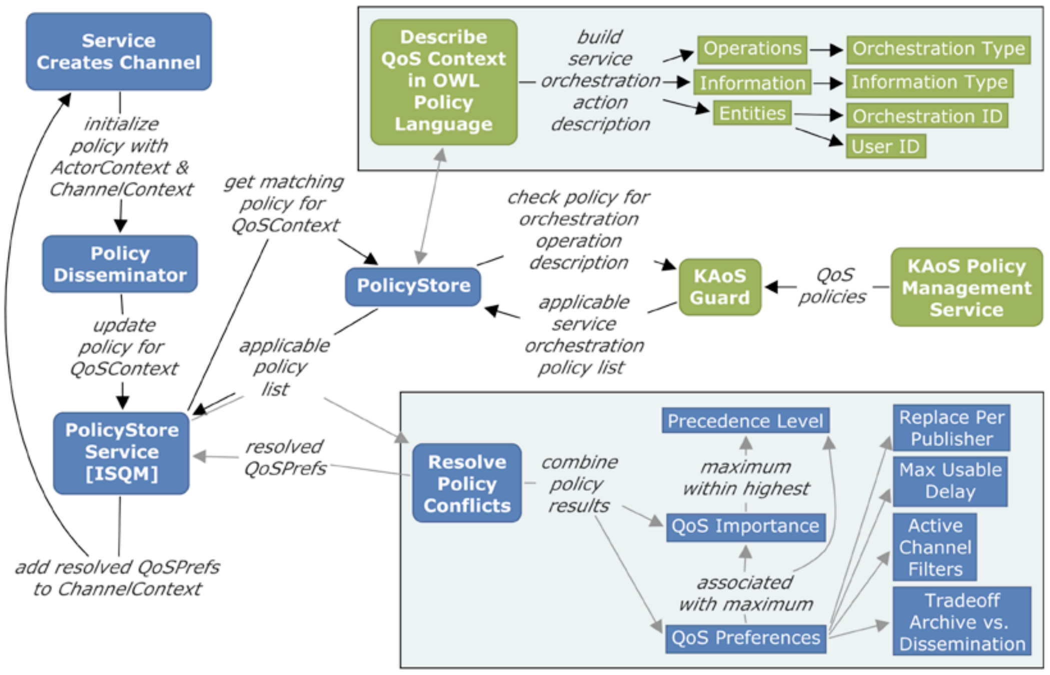

Service Creates Channel
initialize policy with ActorContext & ChannelContext
Policy Disseminator
update policy for QoSContext
PolicyStore Service [ISQM]
add resolved QoSPrefs to ChannelContext
get matching policy for QoSContext
PolicyStore
applicable policy list
resolved QoSPrefs
Describe QoS Context in OWL Policy Language
build service orchestration action description
Operations
Information
Entities
Orchestration Type
Information Type
Orchestration ID
User ID
check policy for orchestration operation description
applicable service orchestration policy list
KAoS Guard
QoS policies
KAoS Policy Management Service
Resolve Policy Conflicts
combine policy results
Precedence Level
maximum within highest
QoS Importance
associated with maximum
QoS Preferences
Replace Per Publisher
Max Usable Delay
Active Channel Filters
Tradeoff Archive vs. Dissemination

Figure 6. The QED Aggregate QoS Manager Workflow

lists of individual users and information types.

Since multiple policies may apply to the context of a single action, *f(O,M,E)*, and these policies may specify conflicting QoS Settings *(v,i,P)*, a mechanism to resolve these conflicts is required. The Aggregate QoS Management class contains the logic to resolve conflicts among a set of applicable QoS policies to arrive at a single Importance value (*i*) and a single value for each of the QoS Preference. This algorithm depends upon the relative precedence values (*v*) for the policies as well as the relative importance values associated with the policies to calculate a single aggregate value for the importance (*i*) and each of the QoS preferences (*p*) according to the following rules:

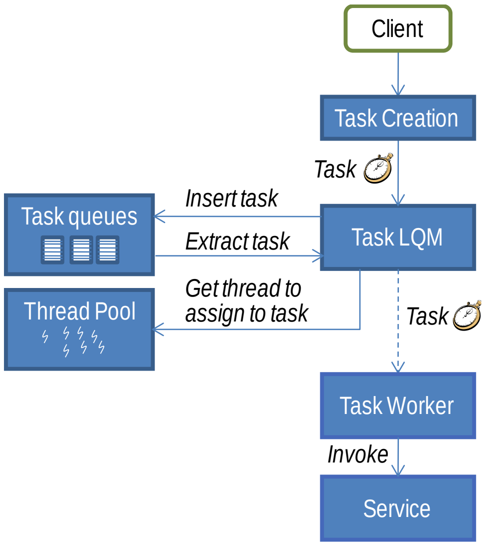


Figure 7. QED Local Task Manager Design

- *Importance (i)*: find the highest precedence level (*v*) at which (*i*) is specified, then take the highest (*i*) with that precedence; *Importance = max(i, max* (v_i)*).*
- *QoS Preference (p)*: find the highest precedence level (*v*) at which (*p*) is specified, then find the highest importance value (*i*) associated with (*p*); *QoS Preference = max(p, max*$(i_p$, *max*$(v_p)))$.

Dissemination of QoS Policy. The aggregate QoS manager uses orchestration instances to disseminate policy to the local QoS managers. When a client creates a connection to the IM services to publish, subscribe, or query, the appropriate orchestration instance is instantiated and assigned a unique identifier as its *orchestrationUID.* The aggregate QoS manager selects the policies that apply to the client, information types it registers, and operations, deconflicts the policies, parses them, and sets attributes on the orchestration instance's context objects. The parsed policy is then available through efficient attribute lookups at each of the local QoS enforcement points, e.g., where CPU or bandwidth is consumed by invocations of the Brokering or Dissemination services, respectively.

5.3 QED Task Management

Predictable performance requires managing the execution of all CPU intensive operations, such as IM Broker 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 7. Each Local Task Manager manages the CPU intensive operations for a given CPU or VM using priority scheduling, which addresses IM Service Challenge 2. The goal is to avoid 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 such as invocations of the Broker Service 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 policy applied to the operation, information type, and/or client as described in Section 5.2) 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 QED has a fixed granularity with which to distinguish tasks.

The QED Task Manager assigns threads from the thread pool to tasks according to a queue management strategy under control of the aggregate QoS manager. In this manner, the Task Manager gracefully handles CPU overload by scheduling the highest priority tasks with the available threads and thus addresses SOA Infrastructure Challenge 1 by managing the Application Server's thread pool. 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 IM 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 (the basis of IM Service Challenge 2). For example, the time to execute the Broker Service depends heavily on the complexity of registered predicates for subscriptions (expressed in XPath or XQuery) and the complexity of the metadata of IOs. We combine two approaches to solve this problem. First, we use heuristics—based 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 [28, 29] 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.

5.4 QED Bandwidth Management

The QED 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. By managing bandwidth at the granularity of an information object, the QED bandwidth manager helps address SOA Infrastructure Challenge 2. The inbound and outbound managers are referred to as the Submission LQM and Dissemination LQM, respectively. The current

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

version of the Bandwidth Manager provides a static bandwidth allocation per interface and to each of the LQMs.

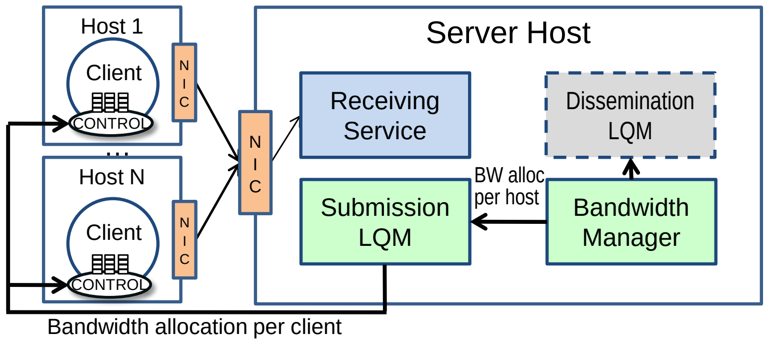


Figure 8. QED Submission LQM Architecture

The Submission LQM architecture, shown in Figure 8, manages the consumption of inbound bandwidth by throttling external 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 architecture, shown in Figure 9, 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 9, the resulting "write-to-client" call from a service invocation is treated as a managed task. When outgoing messages

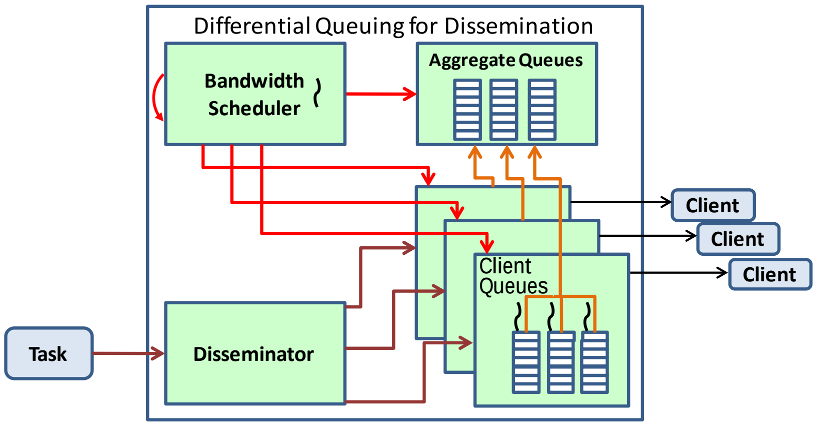


Figure 9. QED Dissemination LQM Architecture

are to be sent to a client, the Dissemination 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. Treating each information-receiving client pair as a distinct schedulable event controllable by policy addresses the fanout identified in IM Service Challenge 4.

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 the information will consume when sent). 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 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 5.3. 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. By taking information size into account when scheduling an object and when determining how long it will take to send, QED addresses IM Service Challenge 1.

5.5 Deployment and Distribution of QoS Managers

QED can be configured to allocate a local task manager for each shared CPU resource, virtual or actual. A host could therefore have one local task manager that schedules operations running on each CPU. Conversely, 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).

QED can also be configured to allocate a bandwidth manager for each occurrence of shared bandwidth, which could be associated with the Network Interface Card (NIC) 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 [7]) and shaping and prioritizing traffic as if that is all the bandwidth available (leaving a reserve of unallocated bandwidth increases the delivered QoS predictability). QED can also be configured to employ a bandwidth management technique in environments in which the amount of available bandwidth is unknown by using TCP acknowledgements to indicate when the next IO should be sent. In this configuration, TCP's congestion control provides an approximation of the amount of bandwidth available to a connection. This approach comes with some risk of slight under-utilization of the bandwidth that can be alleviated using heuristics such as keeping one IO "in transit."

Likewise, QED can be configured to use a submission and dissemination LQM for each occurrence of shared bandwidth for incoming and outgoing messages, respectively. The aggregate QoS manager can be either centralized or distributed. If it is distributed, the policy stores and policy distribution need to be synchronized.

6. **Experimental Results**

This section emperically evaluates the QED capabilities described in Section 5 in the context of the publication-subscription (pub-sub) IM services shown in Figure 1 [8, 18, 25], including (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.

Below we present the results of the following five sets of experiments conducted to evaluate the efficacy and performance of the pub-sub IM services implemented using the QED QoS management services:

1. Evaluate the effect of CPU overload conditions on the servicing of information to demonstrate QED's differentiated services.
2. Evaluate 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).
3. Evaluate the performance of applying new policies to QED's QoS management infrastructure and QED's ability to change policies dynamically and efficiently to handle many users and policy rules.
4. Evaluate QED's ability to shape information characteristics (e.g., size and rate) to match the available bandwidth and improve overall QoS.
5. Evaluate QED's ability to deconflict policies when multiple policies apply in a dynamic situation.

All experiments were run 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.

5.1 Experiment 1: Evaluating QED's Differentiated Service During CPU Overload

This experiment evaluated QED's ability to differentiate service to important clients and information during CPU overload situations. 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 metadata of newly published (for information brokering) or archived (for query) information objects.

The experiment used 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, 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 IOs, no additional messages are disseminated to subscribing clients). We then executed two scenarios: one in which all 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 publish one information object each second (1 Hz), while the low importance publisher publishes 300 information objects per second (300 Hz). 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 10 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. As a result, therefore, only slightly more than half of the high and medium importance information is delivered (.58 Hz for both high and medium publishers).

In contrast, The QED services used JBoss threads to broker the more important information. As a result, they achieved a rate of .99 Hz for both the high and medium information publishers, nearly the full 1 Hz publication rate, prioritizing both over the low importance information that is

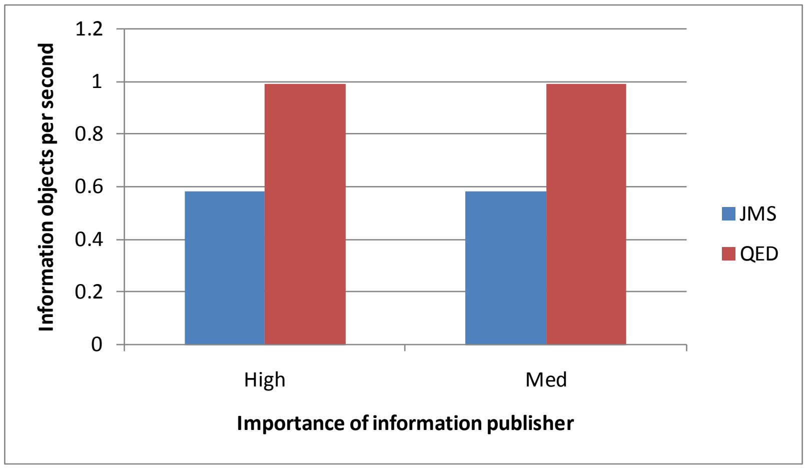


Figure 10. Differentiation among High and Medium Importance Clients in a CPU Overload Scenario.

overloading the system. The baseline JBoss 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 11 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).

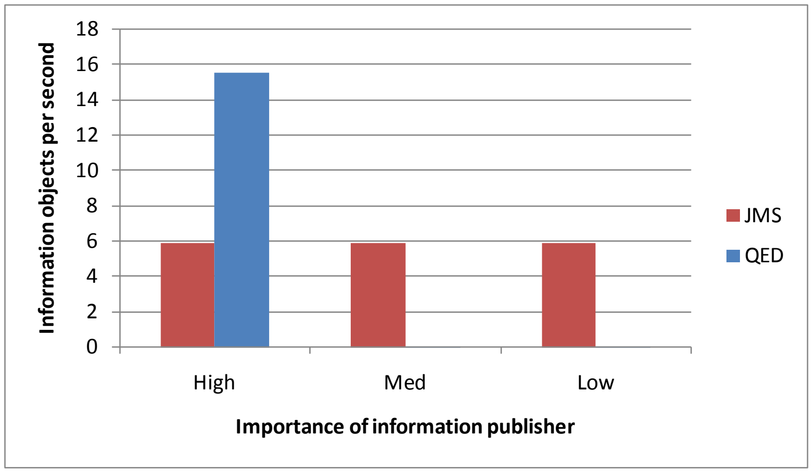


Figure 11. Differentiation in Second CPU Overload Scenario with Each Publisher Type Overloading the System with Information.

5.2 Experiment 2: Evaluating QED's QoS on Bandwidth Constrained Links

Outgoing messages from the Dissemination Service to requesting clients and incoming messages to the Submission Service from publishing clients are the most bandwidth intensive 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 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 12 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 subscribers that were specified as the most important.

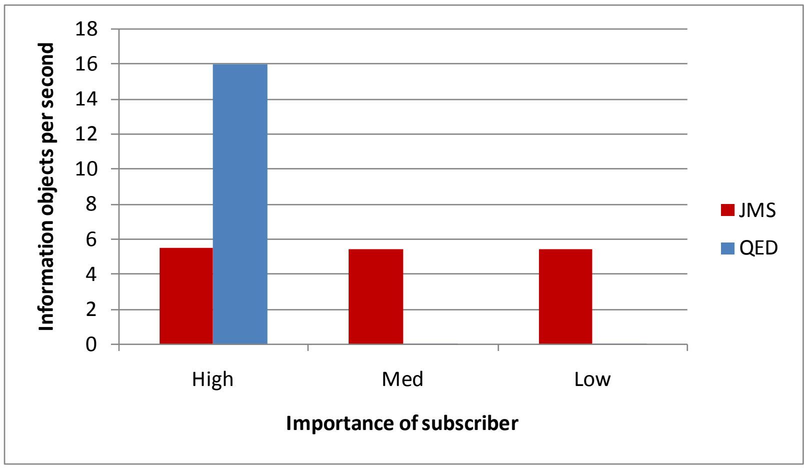


Figure 12. Subscriber Differentiation during Bandwidth-constrained Operation.

5.3 Experiment 3: Evaluating QED's Policy Change Dynamism and Scalability

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 measured the time to add and distribute a policy when the number of existing policies is 2, 10, 100, and 300. Figure 13 shows that 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 measured 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 the time that elapsed before the policy took effect. The results in Figure 14 show that the time needed to effect a policy change scales well, with subsecond times to effect a policy change even with several hundred connections.

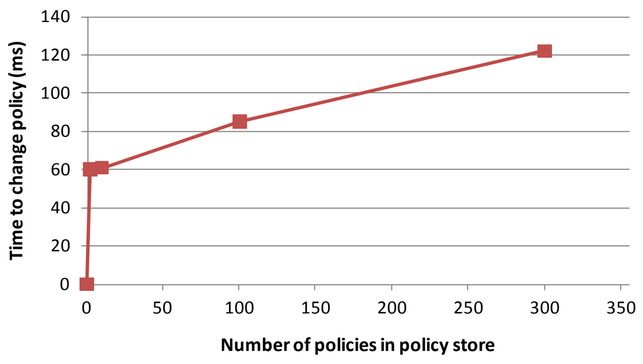


Figure 13. Time to Add a Policy Compared to the Number of Policies.

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 IM policy infrastructure.

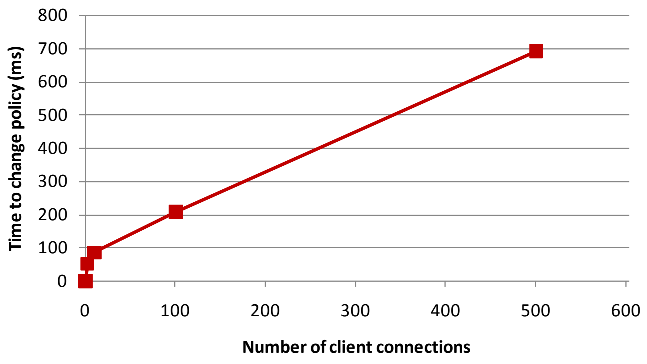


Figure 14. Time to Add a Policy Compared to the Number of Client Connections.

5.4 Experiment 4: Evaluating QED's Data Shaping Capabilities

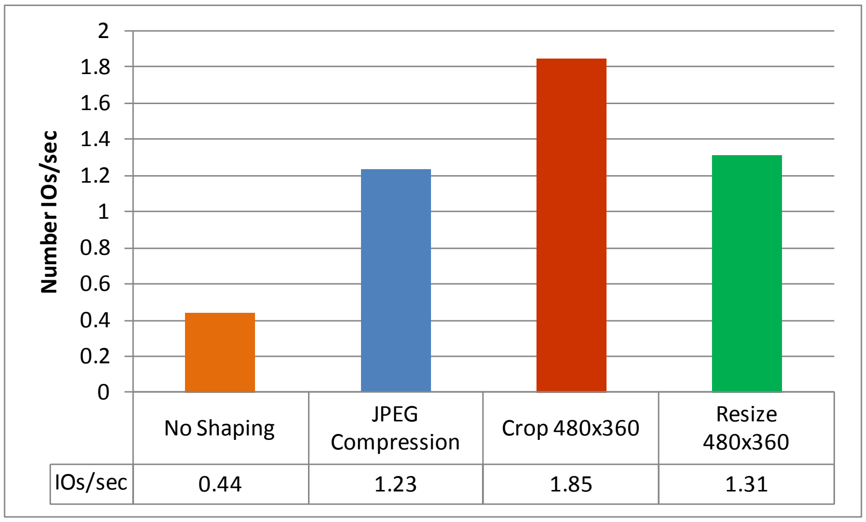
This set of experiments evaluated QED's ability to dynamically change image and XML payloads to reduce latency and jitter experienced by end to end users. We conducted four experiments: three that shaped IOs containing imagery payload by applying image specific transformations and one that shaped more general document-oriented IOs by performing XML transformations.

Image shaping. QED supports a variety of image shaping operations, including *cropping*, *resizing*, and *compression*. Cropping reduces image size by removing less important or unneeded parts of the image. Resizing involves reducing the image scale by shrinking the image dimensions (e.g., from 1200x800 to 300x200). Image compression uses compression algorithms (e.g., JPEG) to retain the original image dimensions but accomplish a smaller size in bytes.

The image shaping experiments all involved a client publishing IOs with payloads containing 1,200x800 images (225 KB size), publishing 3 IOs each second (3 Hz), with a single subscriber subscribing to all IOs, and bandwidth limited to 100 KBps. We then configured the QED services to have a policy that enables each of three specific shaping operations for the payload of the disseminated IOs. The three shaping configurations were JPEG compression to a particular image quality (resulting in a payload approximately 38 KB in size), cropping the image to 480x360 (resulting in a payload of 35 KB size), and resizing the image to 480x360 (resulting in a payload of 40 KB size).

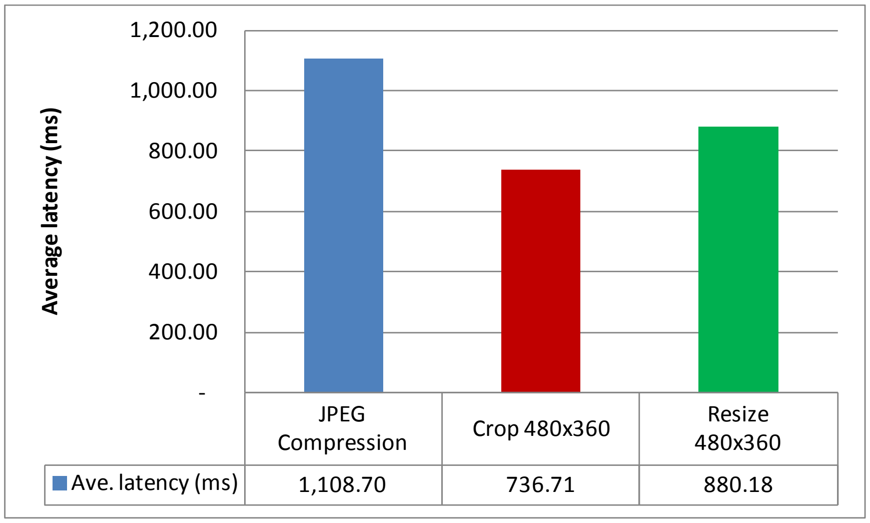
Figure 15 shows the throughput achieved with the three image shaping strategies, compared to a baseline that does no image shaping. As expected, image shaping is able to increase the throughput dramatically with the relative increases in throughput consistent with the differences in resulting size indicated above. Note that the 3 Hz rate is not achievable in any of the cases since the image shaping only affects the payload, not the metadata and transport overhead, such as TCP headers that also contribute to bandwidth usage.

Combining image shaping with replacement policy can increase the perceived quality for imagery significantly, by keeping the latency for delivered imagery low and delivering the freshest imagery available. Figure 16 and Figure 17 show the average latency and jitter (i.e., standard deviation) of each of the image shaping with replacement strategies. In the baseline, queues are growing unbounded because the IOs are simply too large to disseminate at the rate they are being produced. IO shaping transforms IOs to a size that is more closely matched to the available bandwidth, enabling the QED dissemination service to better keep up with the rate of publication and keep the average latency to around a second with low jitter, while the replacement policy flushes old information from the queues.



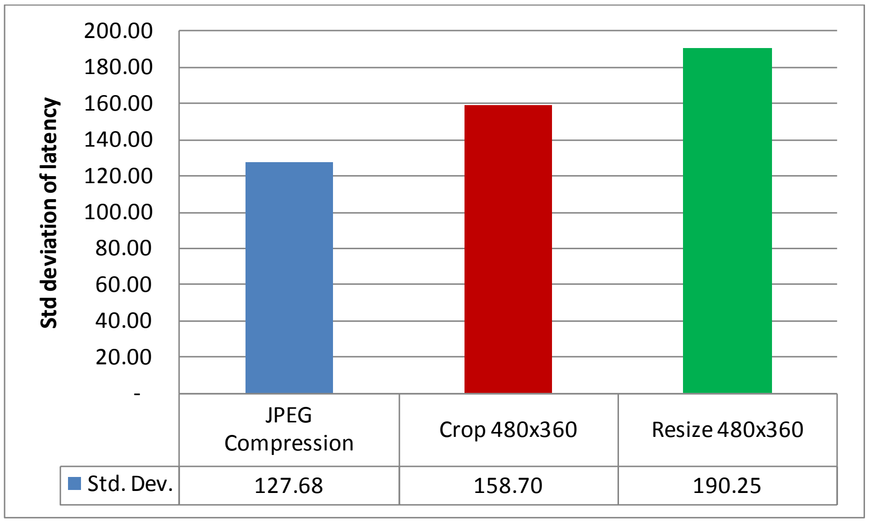
	No Shaping	JPEG Compression	Crop 480x360	Resize 480x360
IOs/sec	0.44	1.23	1.85	1.31

Figure 15. Throughput of the Image Shaping Operations Compared to a Baseline with No Shaping, in a Bandwidth Constrained Scenario.



	JPEG Compression	Crop 480x360	Resize 480x360
🟦 Ave. latency (ms)	1,108.70	736.71	880.18

Figure 16. Average Latency for QED with Image Shaping.



	JPEG Compression	Crop 480x360	Resize 480x360
🟦 Std. Dev.	127.68	158.70	190.25

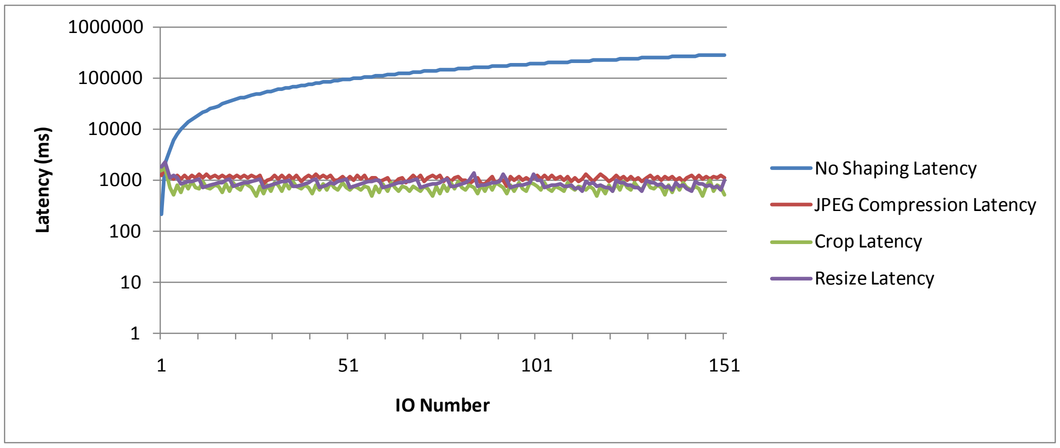
Figure 17. Standard Deviation (Jitter) of Latency for QED with Image Shaping.

With IO shaping and replacement, QED exhibits orders of magnitude improvements in average latency over the baseline. Figure 18a shows the latency of the IO processing and dissemination on a logarithmic scale. The trend of ever increasing latency for the unshaped IOs is clear, while the shaping and replacement of IOs is able to better match the IO size to available bandwidth and control the queue growth, resulting in more controllable latency. Figure 18 also shows the relative lack of jitter and the ability of QED to maintain the latency not just on average, but consistently, within about a second. Figure 18b shows a closeup of the IO shaping strategies on a linear scale.

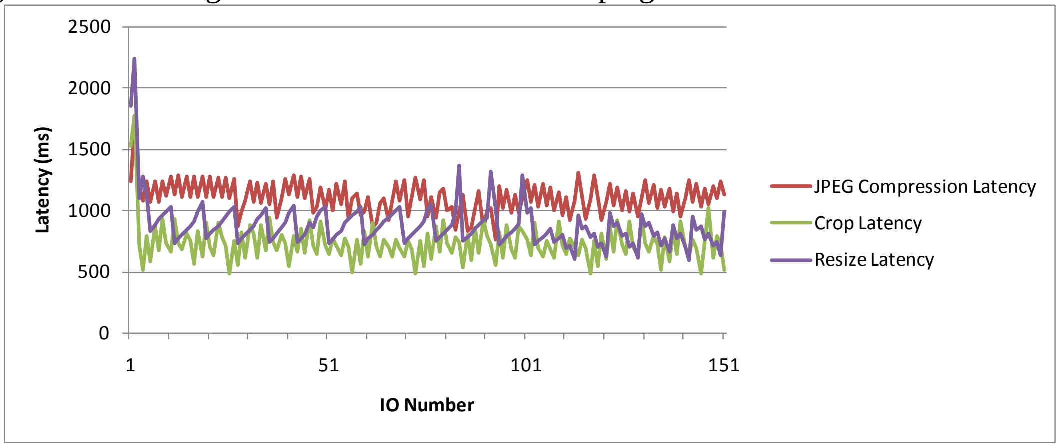
XML shaping. QED's IO shaping supports pluggable algorithms. As an alternative to image shaping, we tested QED's implementation of document transformation using XSLT. We ran an experiment using the *xsl:copy-of* operation to copy elements selectively from an XML source payload element-by-element, according to a user specified file and policy set. The specific XSLT transformation we used is shown below:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                  version="1.0">
<xsl:template match="/">
  <events>
    <xsl:for-each select="events/detail[priority='high']">
      <xsl:copy-of select="self::node()" />
    </xsl:for-each>
  </events>
</xsl:template>
</xsl:stylesheet>
```

This XSL transformation takes an IO with an XML payload and only copies those entries that have a priority element set to 'high'. To gauge the effectiveness of QED's XML shaping to reduce latency and jitter, we created three targeted experiments, each with a 50



(a) Plotted on a logarithmic scale with the no shaping case



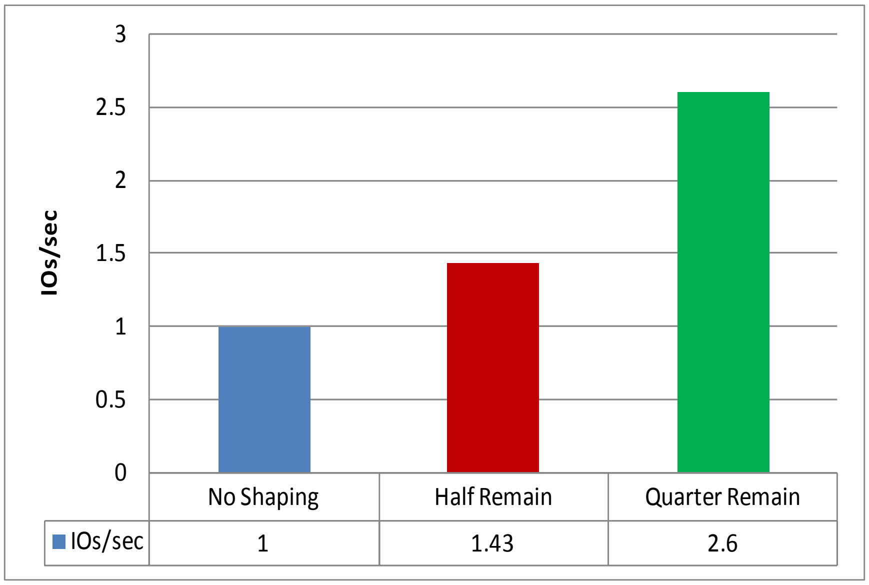
(b) Removing the no shaping case and plotted on a linear scale

Figure 18. Latency of IOs in Shaping Experiments.

Hz publisher of IOs with a 300 KB XML document payload, and a subscriber to all of the IOs. The first experiment had no XML shaping enabled, the second experiment reduced the XML payload to half size (150 KB) via the above transform shown above, and the third experiment reduced the XML payload to a quarter (75 KB) of its original size.

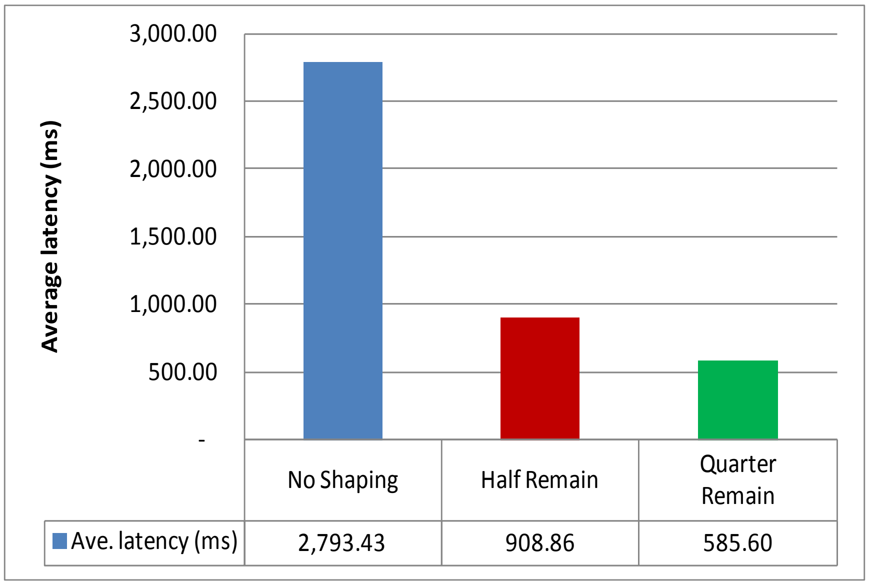
Figure 19 shows the throughput achieved with the two XML shaping strategies, compared to a baseline that does no shaping. XML shaping increases the throughput dramatically. As with the image shaping, the increase in throughput is not equal to the difference in payload size, due to the unshaped metadata and transport overhead.

Figure 20 and Figure 21 show the average latency and standard deviation of the XML shaping compared to the baseline. Similar to the results for image shaping, reducing the XML payload results in greatly improved latency. Selectively copying over half of the elements results a 67% reduction in average latency and over a 94% reduction in jitter. Reducing the XML payload to only a quarter of its original size resulted in a 79% reduction in average latency and slightly better jitter than seen in the half reduction.



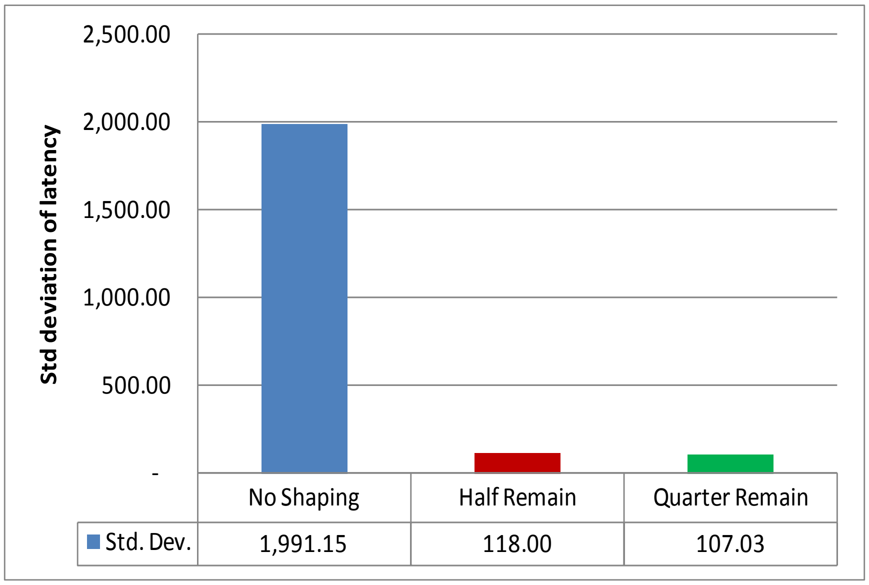
	No Shaping	Half Remain	Quarter Remain
■ IOs/sec	1	1.43	2.6

Figure 19. Throughput of the XML Shaping Operations Compared To a Baseline with No Shaping.



	No Shaping	Half Remain	Quarter Remain
■ Ave. latency (ms)	2,793.43	908.86	585.60

Figure 20. Average Latency for QED with XML Shaping Versus a Baseline with No Shaping.



	No Shaping	Half Remain	Quarter Remain
■ Std. Dev.	1,991.15	118.00	107.03

Figure 21. Standard Deviation (Jitter) of Latency for QED with XML Shaping

The image and XML data shaping operations supported by QED offer users powerful policy-driven mechanisms to match information size and rate to the available bandwidth. When combined with CPU and bandwidth bound prioritization and deadline enforcement, QED's information shaping can help keep queue growth (and therefore latency) of important IOs bounded (approximating real-time delivery), resulting in increased quality of service, improved performance, and a better overall end user experience.

5.5 Experiment 5: Evaluating QED's Policy Deconfliction Capabilities

This experiment demonstrates how QED uses policy precedence for deconfliction in situations in which two or more policies apply to the same operation, information, or actor. In the case of conflicting policy statements, QED uses the policy deconfliction algorithm described in Section 5.2 to select the policy statement that supersedes the others.

The experiment consists of two subscribers (*JTAC* and *TOC*) and two publishers (*UAV1* and *UAV2*), the four policy statements shown in Table 2, and a constrained bandwidth link set at 24,576 Bytes/sec. All IOs generated are tailored to be exactly 8,192 Bytes, and the bandwidth limit is set to allow three IOs per second to be disseminated. The experiment includes two information types, *Blueforcetrack* and *Targeting*. Initially UAV1 is publishing Blueforcetrack IOs. Halfway into the experiment, UAV2 begins to publish targeting information. Both subscribers, JTAC and TOC, maintain subscriptions to both IO types.

Table 2. Policies used in Experiment 5.

No.	Policy Name	Operation	IO Type	Actor	Importance	Precedence
1	BFT	Subscribe	Blueforcetrack	—	5	1
2	TOC#BFT	Subscribe	Blueforcetrack	TOC	7	2
3	Targeting	Subscribe	Targeting	—	3	1
4	JTAC#Targeting	Subscribe	Targeting	JTAC	9	2

The experiment shows deconfliction for both information types. Each IO type has a base policy at precedence 1 (Policy 1 for *Blueforcetrack* and Policy 3 for *Targeting*). These policies (which specify an IO type and operation, but no actor) apply to a subscription by any client for Blueforcetrack and Targeting IOs, respectively.

Policies 2 and 4 are user-specific policies at higher precedence levels (precedence 2) than the base policies. A TOC client subscribing to Blueforcetrack IOs will match policies 1 and 2, which conflict because they assign different importances (5 and 7, respectively). Similarly, a JTAC client subscribing to Targeting IOs will match policies 3 and 4, which conflict because they also assign different importances (3 and 9, respectively). Notice also that in this experiment, when the base policies (1 and 3) apply, subscribers to Blueforcetrack IOs are more important than subscribers to Targeting IOs. However, JTAC clients subscribing to Targeting IOs are more important (importance of 9) than any other users subscribing to any IO type, *if QED deconfliction performs properly*.

During the first part of the experiment, while UAV1 is publishing two Blueforcetrack IOs per second, policy 1 will apply to both the TOC and JTAC client subscriptions and policy 2 will apply to the TOC client subscription. Since there are four IOs to disseminate each second (two Blueforcetrack IOs to each of the two subscribers) and bandwidth to disseminate only three, we expect the TOC client to receive twice as many IOs as the JTAC client if deconfliction functions correctly (resulting in the higher precedence policy 2 applying to the TOC client subscription).

When the UAV2 publisher starts publishing one Targeting IO per second, there will be six IOs to fit into the 3 IO per second constrained bandwidth. We expect the JTAC to receive one Targeting IO per second (in place of the Blueforcetrack IO it was receiving). This requires policy deconfliction to work correctly and apply policy 4, since the other policies applicable to the JTAC would result in delivery of Blueforcetrack IOs. The TOC subscriber should continue to receive its two Blueforcetrack IOs (but no Targeting IOs due to the relatively lower importance assigned by policy 3).

Figure 22 shows the aggregate throughput results from a four minute run in terms of IOs received over one second intervals. Across the entire run, the TOC subscriber receives approximately two Blueforcetrack IOs per second (the TOC-BFT graph), showing that policy 2 is correctly selected during the deconfliction of policies 1 and 2 that both apply to the TOC subscription for Blueforcetracks. During the first two minutes, the JTAC receives approximately one Blueforcetrack IO per second (JTAC-BFT). At the two minute mark, UAV2 begins publishing Targeting IOs. The second case of policy deconfliction occurs, resulting in the correct application of policy 4 to the JTAC subscription to Targeting IOs, elevating that operation to the highest priority. The graph in Figure 22 shows the behavior that we expected and that the policies should enforce. Also as expected, Targeting IOs are not received by the TOC subscriber at any time during the experiment run. The base pol-

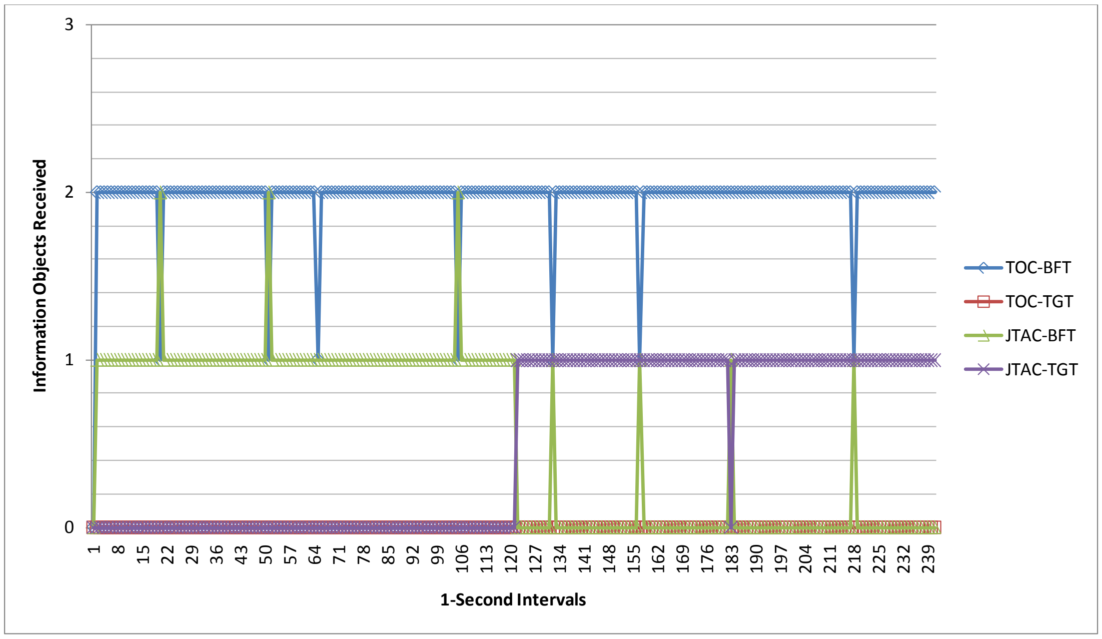


Figure 22. Number of IOs disseminated during policy deconfliction experiment.

icy (policy 3) for Targeting applies to this subscription and the importance assigned (3) is the lowest assigned by any policy, so that IOs of this type for this subscriber never get scheduled for dissemination.

7. Concluding Remarks

Service-Oriented Architecture and Information Management are emerging as two popular paradigms for modern net-centric systems. Each provide powerful abstractions for encapsulating details and supporting the engineering of complex, distributed systems, enabling systems to be developed around the concept of composable services and semantically-rich information objects, respectively. Moreover, information management services have emerged as a critical set of core services in SOA-based net-centric systems, to handle the provision, processing, and dissemination of information. QoS-enabled IM services are essential to support dependable and timely information exchange in net-centric systems. SOA environments and IM services, however, present significant challenges for providing QoS, which is a key requirement for many mission-critical domains and systems.

This paper presented the QED approach to providing QoS for IM services in SOA environments. QED addresses many of the challenges associated with providing QoS for IM services in SOA environments. Our QED prototype and experiments with representative IM systems show significant improvement in performance, predictability, and control over the baseline Phoenix [18] and Apollo [8] IM services, and over the JBoss and JMS SOA middleware. QED includes a high-level QoS policy language, mapping of policies to enforcement points and QoS mechanisms, dynamic task and bandwidth management, aggregation of competing resource demands, and QoS policy-driven prioritization and scheduling strategies.

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

- **QoS management in distributed systems is about managing tradeoffs, not about providing guarantees**. The dynamic numbers of users, competing demands, and fluctuations in resource availability and usage means that providing desired QoS to every user will frequently not be possible. Users and system designers must be provided with a high-level interface (like QED's policy language) to express their desired tradeoffs, such as the relative importance of users, information, and operations; whether loss or delay is more tolerable in overload situations; and whether complete information or the newest information is most important.
- **SOA's abstractions and portability are at odds with providing traditional QoS**. Since conventional SOA abstractions shield applications and operators from key platform-level details it is hard to provide traditional QoS assurance. The QED management layers are an important step toward developing effective, and largely portable, abstractions for QoS concepts.
- **The layered approach is effective in achieving aggregate QoS across the decoupled information publishers and consumers in IM systems**. Disseminating high-level policies (in parsed form) to local enforcement points enables smooth, QoS managed flow of information from publication, through processing, to dissemination, with coordinated policies applying at each enforcement point, e.g., publisher policies apply at the entry of information into the system, consumer policies apply at the dissemination point, and a combination of policies apply when published information is being matched to subscriptions.
- **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.
- **QED is an effective and efficient platform for providing QoS as a normal feature of IM services**. We have successfully integrated QED with two generations of existing IM services and integrated it into those software systems. Moreover, the results presented in this paper validate that QED provides a solid basis for QoS management features in SOA infrastructure in general, not limited solely to IM services.
- **QED's policy-driven approach to QoS management strikes an effective tradeoff between fine-grained control and ease of use**. As SOA middleware infrastructure and IM service instantiations evolve, so must the QoS management capabilities to ease QoS policy configuration, QoS service composition, runtime behavior evaluation, and service deployment, which are distributed in ever increasingly pervasive and ubiquitous computing environments.

Our future work on QED will extend it to 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.

References

1. R. Al-Ali, A. Hafid, O. Rana, and D. Walker, “An approach for quality of service adaptation in service-oriented grids,” *Concurrency and Computation: Practice and Experience*, (15)5:401-412, 2004.
2. K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, “A platform-independent component modeling language for distributed real-time and embedded systems,” *Journal of Computer Systems Science*, 73(2):171–185, 2007.
3. S. Behnel, L. Fiege, and G. Muhl, “On quality-of-service and publish-subscribe,” Proc. 26th IEEE International Conf. on Distributed Computing Systems Workshops (ICDCSW’06), July 4-7, 2006, Lisboa, Portugal.
4. P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Computing Surveys*, 35(2):114-131, 2003.
5. G. Canfora, M. Di Penta, R. Esposito, and M.L. Villani, “An approach for QoS-aware service composition based on genetic algorithms,” Proc. 2005 Genetic and Evolutionary Computation Conference, June 25-29, 2005, Washington, D.C.
6. F. Cao, J. Smith, and K. Takahashi, “An architecture of distributed media servers for supporting guaranteed QoS and media indexing,” Proc. IEEE International Conference on Multimedia Computing and Systems, June 7-11, 1999, Florence, Italy.
7. Cisco, “Using Test TCP (TTCP) to Test Throughput,” Doc. 10340.
8. Combs, V., Hillman, R., Muccio, M., and McKeel, R., “Joint battlespace infosphere: information management within a C2 enterprise,” Proc. 10th Int. Command and Control Research and Technology Symp. (ICCRTS), June 13-16, 2005, McLean, VA.
9. A. Corsaro, L. Querzoni, S. Scipioni, S. Piergiovanni, and A. Virgillito, “Quality of service in publish/subscribe middleware,” *Global Data Management*, IOS Press, 2006.
10. Defense Information Systems Agency, “Net-centric enterprise services,” http://www.disa.mil/nces/.
11. M.A. de Miguel, “Integration of QoS facilities into component container architectures,” Proc. 5th IEEE Int. Symp. on Object-oriented Real-time distributed computing (ISORC), April 29-May 1, 2002, Crystal City, VA.
12. D. de Niz, G. Bhatia, and R. Rajkumar, “Model-based development of embedded systems: the SysWeaver approach,” Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS), April 4-7, 2006, San Jose, CA.
13. D. de Niz and R. Rajkumar, “Partitioning bin-packing algorithms for distributed real-time systems,” *International Journal of Embedded Systems*, 2(3/4):196-208, 2006.
14. DoD CIO, “Department of defense global information grid architectural vision, vision for a net-centric, service-oriented DoD enterprise, version 1.0,” 2007. http://www.defenselink.mil/cio-nii/docs/GIGArchVision.pdf.
15. M.A. El-Gendy, A. Bose, S. Park, and K. Shin, “Paving the first mile for QoS-

dependent applications and appliances," Proc. 12th Int. Workshop on Quality of Service, June 7-9, 2004, Montreal, Canada.

16. M.A. El-Gendy, A. Bose, and K. Shin, "Evolution of the Internet QoS and support for soft real-time applications," *Proceedings of the IEEE,* 91(7):1086-1104, 2003.
17. S. Gopalakrishnan and M. Caccamo, "Task Partitioning with Replication Upon Heterogeneous Multiprocessor Systems," Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS), April 4-7, 2006, San Jose, CA.
18. R. Grant, C. Combs, J. Hanna, B. Lipa, and J. Reilly, "Phoenix: SOA based information management services," Proc. SPIE Defense Transformation and Net-Centric Systems Conference, April 13-17, 2009, Orlando, FL.
19. Z. Gu, S. Kodase, S. Wang, and K. G. Shin, "A model-based approach to system-level dependency and realtime analysis of embedded software," Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS), May 27-30, 2003, Washington, DC.
20. M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. "Java message service version 1.1" Sun Microsystems, April 12, 2002, http://java.sun.com/products/jms/.
21. G. Hardin, "The Tragedy of the Commons", *Science*, 162(3859):1243-1248, 1968.
22. J. Hoffert, D. Mack, and D. Schmidt, "Using machine learning to maintain pub/sub system qos in dynamic environments," Proc. 8th Workshop on Adaptive and Reflective Middleware (ARM), December 1, 2009, Urbana-Champaign, IL.
23. JBoss Community, "JBoss application server," http://jboss.org/jbossas/.
24. M. Jordan, G. Czajkowski, K. Kouklinski, and G. Skinner, "Extending a J2EE server with dynamic and flexible resource management," Proc. 5th Int. Middleware Conference, October 18-22, 2004, Toronto, Canada.
25. M. Linderman, B. Siegel, D. Ouellet, J. Brichacek, S. Haines, G. Chase, and J. O'May, "A reference model for information management to support coalition information sharing needs," Proc. 10th Int. Command and Control Research and Technology Symp. (ICCRTS), June 13-16, 2005, McLean, VA.
26. D. Llambiri, A. Totok, and V. Karamcheti, "Efficiently distributing component-based applications across wide-area environments," 23rd IEEE Int. Conference on Distributed Computing Systems (ICDCS), May 19-22, 2003, Providence, RI.
27. G. Lodi, F. Panzieri, D. Rossi, and E. Turrini, "Experimental evaluation of a QoS-aware application server," Proc. 4th Int. Symp. on Network Computing and Applications (NCA), July 27-29, 2005, Cambridge, MA.
28. J. Loyall, M. Carvalho, A. Martignoni III, D. Schmidt, A. Sinclair, M. Gillen, J. Edmondson, L. Bunch, and D. Corman, "QoS enabled dissemination of managed information objects in a publish-subscribe-query information broker," Proc. SPIE Defense Transformation and Net-Centric Systems Conference, April 13-17, 2009, Orlando, FL.
29. J. Loyall, A. Sinclair, M. Gillen, M. Carvalho, L. Bunch, A. Martignoni III, and M.

Marcon. “Quality of service in US Air Force information management systems,” Proc. Military Communications Conference (MILCOM), October 18-21, 2009, Boston, MA.

30. J. Loyall, M. Gillen, A. Paulos, J. Edmondson, P. Varshneya, D. Schmidt, L. Bunch, M. Carvalho, and A. Martignoni III. “Dynamic policy-driven quality of service in service-oriented systems,” Proc. 13th IEEE Computer Society Symp. on Object/component/service-oriented Real-Time distributed Computing (ISORC), May 5-6, 2010, Carmona, Spain.

31. N. Mabrouk, S. Beauche, E. Kuznetsova, N. Georgantas, and V. Issarny, “QoS-aware service composition in dynamic service oriented environments,” Proc. 10th Int. Middleware Conference, Nov. 30-Dec. 4, 2009, Champaign, IL.

32. S. Mahambre, M. Kumar, and U. Bellur, “A taxonomy of QoS-aware, adaptive event-dissemination middleware,” *IEEE Internet Computing*, 11(4):35-44, 2007.

33. OASIS, “Extensible access control markup language (XACML) version 2.0,” OASIS Standard, docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf, February 1, 2005.

34. OASIS, “Web services-notification,” www.oasis-open.org/committees/wsn.

35. Object Management Group. “Data distribution service for real-time systems, version 1.2,” OMG Specification, formal/07-01-01, January 2007. http://www.omg.org/cgi-bin/doc?formal/07-01-01.

36. Object Management Group. “Notification service, version 1.1,” OMG Specification, formal/2004-10-11, October 2004. http://www.omg.org/technology/documents/formal/notification_service.htm.

37. T. Repantis, X. Gu, and V. Kalogeraki, “Synergy: Sharing-aware component composition for distributed stream processing systems,” Proc. 7th Int. Middleware Conference, November 27-December 1, 2006, Melbourne, Australia.

38. M. Roughan, S. Sen, O. Spatscheck, and N. Duffield, “Class-of-service mapping for QoS: a statistical signature-based approach to IP traffic classification,” Proc. 4th ACM SIGCOMM Conference on Internet Measurement, August 30 – September 3, 2004, Portland, OR.

39. R. Schantz, J. Loyall, C. Rodrigues, D. Schmidt, Y. Krishnamurthy, and I. Pyarali, “Flexible and adaptive qos control for distributed real-time and embedded middleware,” Proc. 4th Int. Middleware Conference, June 16-20, 2003, Rio de Janeiro, Brazil.

40. R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquier, and J. Loyall, “An object-level gateway supporting integrated-property quality of service,” Proc. 2nd Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC), May 2-5, 1999, Saint Malo, France.

41. Scientific Advisory Board (Air Force), “Building the joint battlespace infosphere volume 1: summary,” SAB-TR-99-02, December 17, 1999.

42. Scientific Advisory Board (Air Force), "Building the joint battlespace infosphere volume 2: interactive information technologies," SAB-TR-99-02, December 17, 1999.

43. Springsource Community, Spring, http://www.springsource.org/.

44. J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "Vest: an aspect-based composition tool for real-time systems," Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS), May 27-30, 2003, Washington, DC.

45. C. Stewart and K. Shen, "Performance modeling and system management for multi-component online services," Proc. 2nd Symp. on Networked Systems Design & Implementation (NSDI), May 2-4, 2005, Boston, MA.

46. A. Uszok, J.M. Bradshaw, M. Breedy, L. Bunch, P. Feltovich, M. Johnson, and H. Jung, "New developments in ontology-based policy management: increasing the practicality and comprehensiveness of KAoS," Proc. IEEE Workshop on Policies for Distributed Systems and Networks, June 2-4, 2008, Palisades, NY.

47. P. Vienne and J.L. Sourrouille, "A middleware for autonomic QoS management based on learning," Proc. 5th Int. Workshop on Software Engineering and Middleware, September 5-6, 2005, Lisbon, Portugal.

48. P. Wang, Y. Yemini, D. Florissi, and J. Zinky, "A distributed resource controller for QoS applications," Proc. 7th IEEE/IFIP Network Operations and Management Symposium (NOMS), April 10-14, 2000, Honolulu, HI.

49. D. Wichadakul, K. Nahrstedt, X. Gu, and D. Xu, "2K: an integrated approach of QoS compilation and reconfigurable, component-based run-time middleware for the unified QoS management framework," Proc. Int. Middleware Conference, November 12-16, 2001, Heidelberg, Germany.

50. World Wide Web Consortium, "OWL web ontology language overview," W3C Recommendation, February 10, 2004. http://www.w3.org/TR/owl-features/.

51. World Wide Web Consortium, "XML path language (XPath) version 1.0," W3C Recommendation, November 16, 1999. http://www.w3.org/TR/xpath.

52. World Wide Web Consortium, "XQuery 1.0: an XML query language," W3C Recommendation, December 14, 2010. http://www.w3.org/TR/xquery.

53. World Wide Web Consortium, "XSL transformations (XSLT) version 1.0," November 16, 1999. http://www.w3.org/TR/xslt.

54. C.W. Zhang, S. Su, and J.L. Chen, "Genetic algorithm on web services selection supporting QoS," *Chinese Journal of Computers*, 29(7):1029-1037, 2006.