Policy Specification and Enforcement for Quality-of-Service in Service-Oriented Information Management

Larry Bunch[1], James Edmondson[2], Joe Loyall[3], Doug Schmidt[2], Asher Sinclair[4], and Marco Carvalho[1]

[1] Florida Institute for Human and Machine Cognition, 40 S. Alcaniz St. Pensacola, FL, USA
{lbunch, mcarvalho}@ihmc.us
[2] Vanderbilt University, 2015 Terrace Place, Nashville, TN 37203, USA
{jedmondson, schmidt}@dre.vanderbilt.edu
[3] BBN Technologies, 10 Moulton Street, Cambridge, MA 02138, USA
jloyall@bbn.com
[4] US Air Force Research Labs, 525 Brooks Road, Rome, NY 13441, USA
Asher.Sinclair@rl.af.mil

Abstract. This paper presents a policy-driven approach to quality-of-service (QoS) management that works with existing service-oriented data dissemination middleware, such as the Java Messaging Service (JMS). Our approach includes services that (1) support specifying and enforcing the QoS preferences of individual clients, (2) mediate and aggregate QoS management on behalf of competing users, and (3) shape information to improve real-time performance. This paper makes two contributions to research on QoS-enabled data dissemination middleware. First, we describe how our policy-driven approach bridges users to the underlying dissemination middleware and enables QoS control based on rich and meaningful context descriptions including users, data types, client preferences, and information characteristics. Second, we present experimental results that quantify the improved control, differentiation, and client-level QoS enabled by our approach.

Keywords: Quality-of-Service, Information Management, Policy, Service Oriented Architecture, Dissemination Middleware.

1 Introduction

Contemporary middleware platforms, such as Service-Oriented Architecture (SOA) and Publish-Subscribe-Query Information Management (IM) services, provide powerful abstractions to develop distributed systems of significant distribution, scale, and functionality. These middleware platforms, however, generally provide limited support for *quality-of service* (QoS), which is defined by the perceived state of user satisfaction, i.e., the difference between the service delivered by a system and what is expected by users.[2] In general, the lower the difference, the higher the QoS.

One drawback with this broad definition of QoS is that large-scale distributed information systems have many users whose requirements may conflict. Moreover, the aggregate QoS requirements of all users may conflict with QoS requirements for

[1] This definition differs from the narrow networking perspective that defines QoS in terms of traffic engineering, i.e., network resource reservation or differentiated services [1].

individual users. For example, in a disaster response scenario, an overall search and rescue goal may be better serviced by degrading the service provided to individual users, e.g., due to their relative lower importance to the aggregate goals.

This paper presents the *QoS-Enabled Dissemination* (QED) policy-driven approach to QoS management for service-oriented data dissemination middleware that (1) enables the specification and enforcement of the QoS preferences of individual clients, (2) mediates and aggregates QoS management on behalf of competing users, and (3) shapes information to improve overall operational goals and user experiences. Our prior work on QED described the QED architecture and an early prototype [2], incorporating its QoS management capabilities in SOA middleware [3], and applying QED to tactical information management systems [4].

This paper describes heretofore unexamined topics pertaining to QED's *QoS Policy language*, which can attach policies to application and middleware observable attributes of entities, operations, and information (as opposed to network observable attributes of packets or classes of traffic). Policies are parsed and disseminated to enforcement points where (1) QoS can be affected (e.g., where information or processing can be delayed) and (2) policies can be enforced rapidly along with the processing and information flow. We have prototyped the QED QoS Policy language by extending the Phoenix [5] information system, which provides web services developed to provide JMS-based capabilities for information submission, information brokering and discovery, repository, query, type management, dissemination, session management, authorization, service brokering, and event notification.

In addition to describing the structure and functionality of QED's QoS policy language support, we present the results of experiments. These experiments quantify QED's ability to enforce policies in a way that differentiates important information from unimportant information (according to user specifications). The experiments also show how QED dynamically responds to situations in which the servers are CPU-bound, bandwidth-bound, or in need of higher soft real time performance, while still enforcing user-defined QoS policies.

2 Motivating Scenario

To motivate our work on the QoS policy language in QoS, consider a search and rescue operation in a flood ravaged territory; similar to what recently occurred in Nashville, Tennessee during May of 2010. Search and rescue crews deployed sensors or unmanned aerial vehicles (UAV) to aid in the search for survivors trapped in homes, on isolated high ground, or in other critical situations. During their rescue missions, each sensor or UAV publishes images of the environment around them, tagged as an AerialImage data type, to interested search and rescue crews on the ground, in boats, helicopters, etc. Each UAV may be assigned a grid on a digital map or may be free roaming, but we assume that ground personnel can detect where the images are geographically located. Ground-based search and rescue operators may need to publish images showing potential survivors in a larger format while other images showing no survivors could be reduced to a small format or potentially have the payload stripped entirely (to stop unimportant messages from flooding the dissemination mechanisms).

Along with image publishers, crews may also deploy simple XML transmitters that relay important and unimportant rescue events to crews, including GPS information, team objectives, or other descriptive information. When the system is under stress, the operators may only want to disseminate important events and not bother with unimportant events. Moreover, for both image and XML payloads, operators are interested in fresh data (not necessarily the most complete data), as long as important landmarks are still visible in survivor area images and enough pertinent information is relayed in XML data that rescuers are not hampered in rescue efforts.

As shown in Table 1, each user and data type combination may result in a different perceived importance level (according to an administrator). Likewise, each XML or Image payload may be reduced in size to accommodate high traffic or an over-utilized system capacity situation. Decisions for when to shape XML or Image payloads should be configurable by an administrative user.

In several situations (e.g., Analyst subscription to AerialImages and GPSTrack), system administrators should be able to specify that fresh data is more important than ordered (and possibly stale) data, which can help rescuers be up-to-date with the latest conditions and help prune unimportant information. Moreover, when the system is under duress (e.g., CPU or bandwidth capacities are being exceeded), the middleware should be able to shape XML or Image payloads and prioritize data delivery or processing to maximize the overall utility of the system for current operations.

Table 1. Users, Datatypes, and Importance of Info During a Search and Rescue Mission.

User	Operation	Data Type	Importance
All users	Publish	GPSTrack (XML)	Low
UAV	Publish	AerialImage (Image)	Low
Analyst	Subscribe	GPSTrack (XML)	Low
	Subscribe	AerialImage (Image)	Medium
	Subscribe	Capacity (XML)	Low
	Publish	RescueImage (Image)	High
	Publish	ShelterInfo (XML)	Low
GroundCrew	Subscribe	RescueImage (Image)	High
	Subscribe	GPSTrack (XML)	Medium
	Subscribe	ShelterInfo (XML)	Medium
Shelter	Publish	Capacity (XML)	Low

3 QoS Policy Specification in QED

The section describes QED's QoS Policy specification language, which enables users to describe system contexts in terms of meaningful, domain-specific concepts, such as *subscriptions to UAV imagery by ground rescue crews*, and map these contexts to high-level 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. QoS policies in QED are independent from the underlying information management system implementation. These policies are also formal and readily accessible by software for reasoning and enforcement.

3.1 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*) $\rightarrow$ 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* in our scenario. The system context may also be extended to include *Resources*, such 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) $\rightarrow$ (f(O,M,E) $\rightarrow$ (v,i,P)), as described in Sections 5 and 6.

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 it 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 rescue mission manager. 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 mission-level measure of the relative value of an operation, type, or client to a mission. This value is used, along with other factors such as cost, to prioritize processing and dissemination of information. In our motivating scenario, the importance captures that *analyst publications and ground crew subscriptions to rescue images* are the most vital to overall mission success.

QoS preferences, *P*, define a named set of limits and tradeoffs among aspects of QoS, such as deadlines for delivering information objects 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 via KAoS policy framework [6] using the extensible OWL semantic web language [7]. 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.

Fig. 1. graphically depicts the OWL classes used by the KAoS-based implementation of system context, 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 compo-

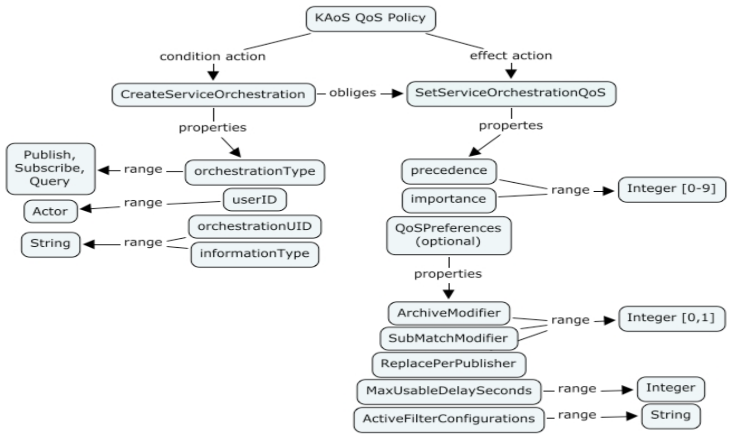


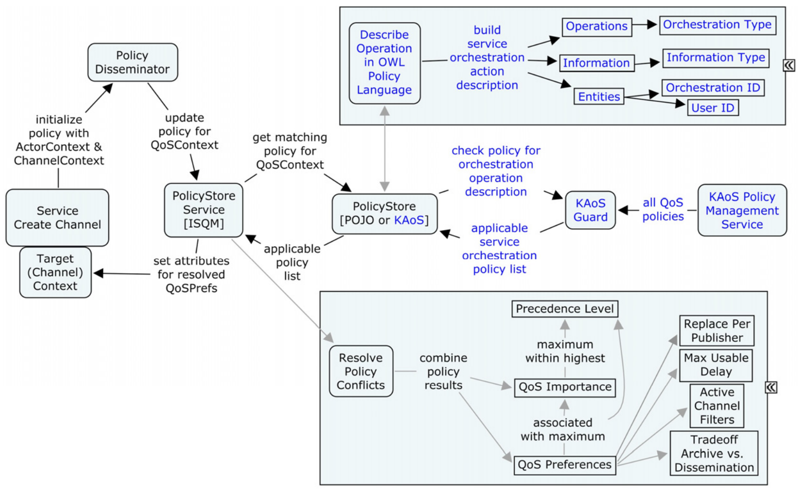
Fig. 1. Graphical Representation of OWL Ontology Describing Service Orchestration Creation Context, QoS Settings, and Attributes of the QoS Preferences Construct

nent performs a policy check, it creates an instance of this 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, Query, and Archive. Likewise, the *informationType* is a domain-specific identifier for the type of information such as a Rescue Request. 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. The *orchestrationUID* is a String property containing the unique ID of the 'target' context associated with the orchestration, this is typically a session identifier provided by the information management system.

The QoS Settings are represented by an operation *SetServiceOrchestrationQoS* with the following attributes:

- *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, such as allowing compression before disseminating any images then overriding this specifically for rescue images where the fidelity must be maintained. The QED user interfaces currently assign a precedence automatically based first 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 to 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 degrade the performance of a client's information flow in the face of resource bottlenecks. This OWL object property defines a range over the class



Policy Disseminator
initialize policy with ActorContext & ChannelContext
update policy for QoSContext
Service Create Channel
Target (Channel) Context
PolicyStore Service [ISQM]
set attributes for resolved QoSPrefs
get matching policy for QoSContext
applicable policy list
PolicyStore [POJO or KAoS]
Describe Operation in OWL Policy Language
build service orchestration action description
Operations
Orchestration Type
Information
Information Type
Entities
Orchestration ID
User ID
check policy for orchestration operation description
KAoS Guard
all QoS policies
KAoS Policy Management Service
applicable service orchestration policy list
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

Fig. 2. The ISQM performs policy checking and conflict resolution each time the Submission and Dissemination Services create a service orchestration. The resulting QoSPreferences are set as attributes on the associated ChannelContext.

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) *ArchiveModifier*/*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 are a list of string values that identify the shaping operations the QED information shaping will perform.

3.2 QoS Policy Management and Decision Making

QED defines a Policy Store component in the IM architecture to provide transactional management of high-level policies. We also introduce a new *Infospace QoS Manager* (ISQM) component to the IM architecture that maps these high-level policies to an orchestrated combination of low-level control settings. These low-level QoS controls, which we refer to as *Local QoS Managers* (LQM), are specific to the components and services comprising the system and therefore distributed and heterogeneous.

The primary role of the ISQM is to manage the overall QoS policies for the information services and their users. 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. Each time the ISQM performs a policy check, it compares the observed attributes and their values against the policy context definitions to decide the applicable set of policies, as shown in Fig. 2.

QED Policies are maintained in the pluggable Policy Store. Two implementations are provided in QED: one based on the KAoS policy services and another based on POJO (Plain old java objects). 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 lists of individual users and information types.

Since multiple policies may apply to the context of a single ISQM 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 ISQM 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:

- *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* = $max(p, max(i_p, max(v_p)))$.

3.3 Dissemination of QoS Policies

A *QoSContext* object is created when a client initiates a session, which occurs when a client use *createSession* to create a new session with the *SessionManagementService*, representing the authentication step. The client then gets a reference to the service with which it needs to interoperate, e.g., the *SubmissionService* for a publication or the *DisseminationService* for a subscription, through a *ServiceBrokeringService* or other means. The client then creates a channel instance to the service, e.g., by calling *createOutputChannel* for publication, and passing a *SessionTrack* containing the client's *session*. This last step creates a *QoSContext* object, which contains the Phoenix Contexts for one of a set of orchestration context patterns, as shown in Fig. 3 for a *Publication Orchestration* instance. Our QED prototype defines four core orchestration context patterns as part of the QED configuration, corresponding to the publication, archive, subscription, and query operations.

When a new QoSContext is instantiated, the ISQM looks up the set of policies that pertain to that context and reduces it to the smallest non-contradictory set of applicable policies. The relevant parts of the policy (e.g., the importance and QoS preferences) are stored as attributes on Phoenix contexts referenced in the QoSContext. The use of context attributes in Phoenix contexts referenced in QoSContexts serves the following purposes:

• Policy lookup and parsing, which can take some time, are attached to the relatively infrequent operations of creating new QoS contexts (when new client sessions are created).
• Policy enforcement, which is in the mainline of IO publication, processing, and dissemination, is very quick because it consists of attribute lookups on local Phoenix contexts.
• Policy updates to attributes in Phoenix context attributes in a QoSContext are transactional, helping maintain consistent policies.
• Policy distribution is automatic and efficient, since Phoenix contexts are visible to the LQMs.

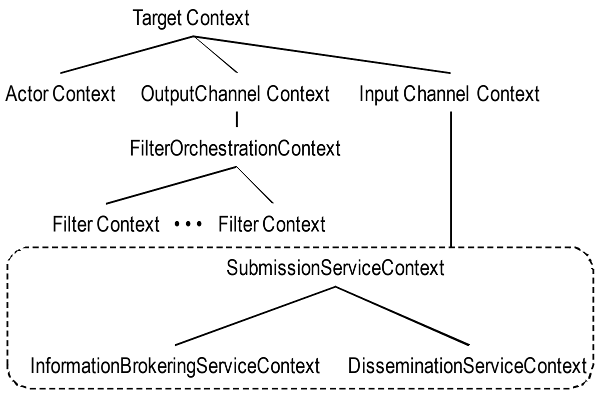


Fig. 3. Example of a QoSContext object representing a client publication session.

4 QoS Policy Enforcement

This section describes the mechanisms that QED provides to enforce QoS Policies specified via the language techniques presented in Section 3.

4.1 Task management

Achieving predictable performance requires managing the execution of all CPU intensive operations, such as service invocations, for each CPU (or equivalent virtual machine, VM) onto which clients and services are distributed, including the following capabilities:
• Prioritized scheduling of operations based on importance and cost (e.g., time to execute).
• Limiting the size of the thread pool to a number of threads that can be executed on the CPU (or a portion allocated to the VM) without overloading it.
• Scheduling according to an appropriate policy, such as *strict* or *weighted fair*.

To manage these tasks, QED provides *Local Task Managers*, whose design is shown in Fig. 4. Each Local Task Manager manages the CPU intensive operations for a given CPU or VM using priority scheduling. 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 (e.g., service invocation) are performed, tasks are created and submitted to the Task Manager, where they are inserted into a binned priority queue using a configurable number of bins (queues), each representing a priority. Task creators calculate an importance (derived from a policy applied to the operation, information type, and/or client) and cost for the task. The Task Manager takes importance and cost as inputs and generates a priority (bin assignment). Binned queues also allow QED to support a weighted-fair policy, which is hard to implement in a heap-based implementation. The tradeoff is that we have a fixed granularity with which to distinguish tasks.

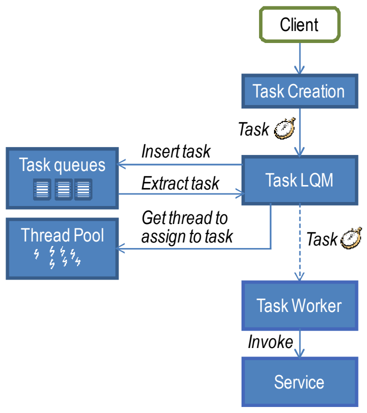


Fig. 4. QED Local Task Manager Design

The 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. 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.

4.2 Bandwidth Management in QED

The Bandwidth Manager is a host-level entity that assigns bandwidth slices for inbound and outbound communications based on policy provided by the aggregate QoS manager. For SOA architectures, bandwidth is managed at the level of information objects, not packets, as it is usually done in network-level QoS, since the loss or delay of an individual packet could invalidate an entire information object, which is possibly much larger than individual data packets.

The inbound and outbound managers are referred to as the Submission LQM and Dissemination LQM, respectively. The current version of the Bandwidth Manager provides a static bandwidth allocation per interface and to each of the LQMs.

The Submission LQM, shown in Fig. 5, 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.

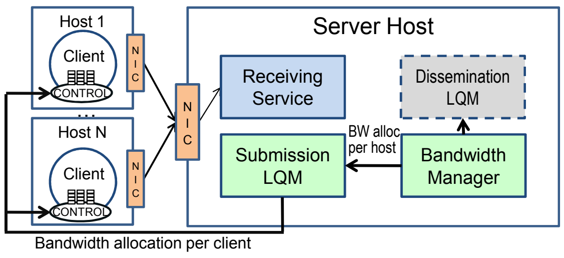


Fig. 5. QED Submission LQM Design

The Dissemination LQM shown in Fig. 6 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 Fig. 6, the resulting “write-to-client” call from a service invocation is treated as a managed task. When outgoing messages are to be sent to a client, the 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.

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

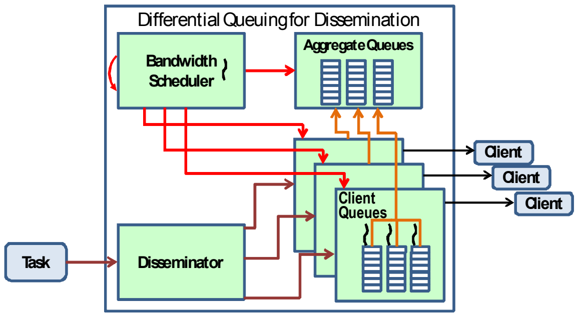


Fig. 6. QED Dissemination LQM Design

sending the information will consume). At this point, the priority is used to determine which client bin should be used to enqueue the data.

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

The Bandwidth Scheduler 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 support for strict and weighted fair algorithms. 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.

5 QoS Monitoring

The QoS Monitoring service collects statistics about a wide range of metrics, both at the system level as well as at the application level. QoS Monitoring relies on the Cross-layer (XLayer) [8] substrate capability of storing time-series containing sets of values for each desired metric and providing real-time statistics resulting from the collected data. Moreover, using the XLayer the metric information can be seamlessly propagated between the IM services and the clients (e.g., publishers and subscribers).

5.1 Monitoring Service

The Monitoring Service is the main component of QED's QoS Monitoring. It is implemented as a Phoenix service (extending *BaseService*) with related stub and connector, operating as a high-level interface for the monitoring functionalities provided by the XLayer substrate. In particular it contains two sets of functionalities: the first set manages the registration of new metrics provided by monitoring components and enables the update of existing metrics. The second set of operations allows other services to retrieve statistics about the currently monitored metrics, either by polling or via the subscription mechanism.

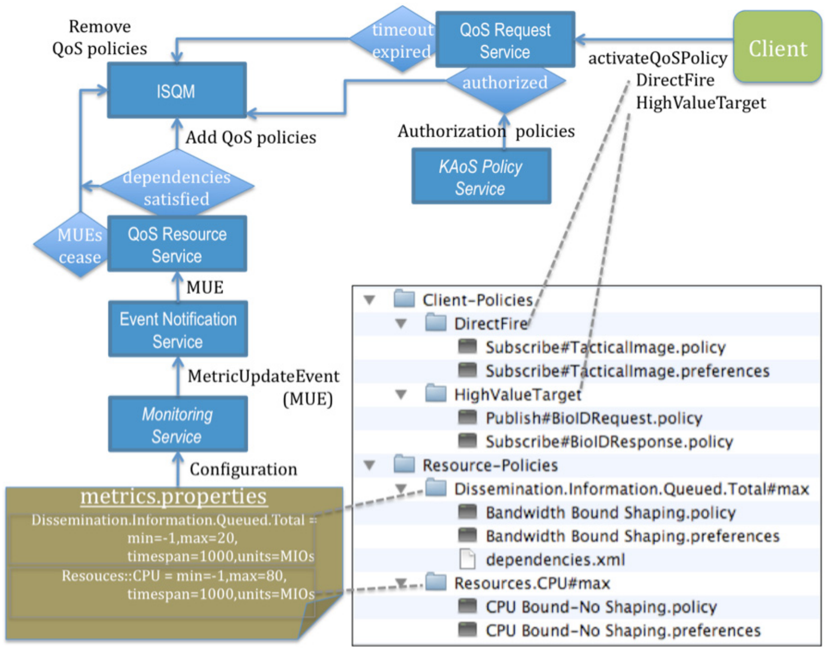
The Monitoring Service is implemented to be flexible. Other services in Phoenix may define their own monitoring components and dynamically add them to the main Monitoring Service. Also, each monitoring component is allowed to define customized metrics it wishes to monitor, registering them with the Monitoring Service and specifying itself as a provider. Once a metric is registered the Monitoring Service returns (and stores) a reference to the related Metric Recorder. The Metric Recorder exposes the API to update the values for the metric to which it is related.

The Monitoring Service takes care of interacting with the XLayer substrate, which supports the metrics storage and retrieval at a lower level. Moreover XLayer incorporates a set of built-in system-related metrics (e.g., CPU and memory utilization and network traffic per interface).

6 Resource-Driven QoS Policy

This section describes how the resource-driven QoS policy feature of QED enables the ISQM to add and remove sets of QoS policies as the monitored states of IM services resources increase and abate. This capability supports an autonomic response of the system to recognize when resources are becoming scarce and adjust information handling policy to alleviate the scarce resource conditions. An example of the potential benefit of this feature is QED automatically recognizing that the Dissemination service is bandwidth-bound and adjusting QoS policy to perform additional compression on image types to conserve bandwidth resources.

The *QoS Resource Service* (QRCS) provides the resource-driven QoS policy functionality. This service is configured with sets of QoS policies and related QoS preferences, each named according to the resource condition it represents (e.g., Resources.CPU in Fig. 7). The *#max* or *#min* suffix on the policy set name indicates



```
Dissemination.Information.Queued.Total =
        min=-1,max=20,
        timespan=1000,units=MIOs
Resouces::CPU = min=-1,max=80,
        timespan=1000,units=MIOs
```

Fig. 7. The QoS Resource Service activates and deactivates sets of QoS policies by the name of the folder on disk containing the related policy and preferences files.

which of the metric thresholds must be violated to activate the policy set. A policy set can contain multiple policies and preferences, all of which will be added and removed as the policy set is activated and deactivated respectively.

The QRCS relies upon the Monitoring service (described in Section 5) to recognize the changes in resource states. The Monitoring service is configured at startup time with metrics that apply to resources such as CPU and LQM-reported queue sizes. The Monitoring Service notifies when the resource usage exceeds the Metric's threshold value (if *max*) or falls below the threshold (if *min*). The QRCS subscribes to the notification about the Metrics that apply to the QoS policy sets defined for the QRCS. When the QRCS starts receiving notifications that a resource Metric has crossed its threshold, the QRCS service adds the policies in the policy set with the same name as the Metric. Similarly, when the QRCS stops receiving notification about a Metric it removes the policies in the associated resource policy set after a configurable amount of time.

The QoS Resource Service interacts with the ISQM to add and remove its QoS policies and preferences. In this way, the QRCS relies upon the ISQM for the dissemination and enforcement of the QoS policies that the QRCS manages. The QoS Resource Service operates autonomously according to its configuration; therefore no runtime interaction with the service is necessary. The public interface of the QRCS is primarily intended to provide information about the resource-driven policies that are available and that are in currently active.

7 Experimental Results

To validate our solution approach, we constructed a set of experiments to gauge the effectiveness of our QED prototype in meeting QoS and soft real-time needs in situations similar to the search and rescue operations outlined in the Section 2. We gauge QED effectiveness in several different scenarios: (1) situations in which the server is CPU bound, (2) situations in which the dissemination bandwidth is insufficient for all information, and situations in which (3) images and (4) XML documents are needed quickly and within a reliable time window.

Our hypothesis for the experiments outlined in Section 7.1 and 7.2 is that policy-driven QoS enforcement will result in more delivery of important information than less important information (as defined by the user), which according to our motivating scenario should result in vital information about survivors being delivered before lower priority data. Our hypothesis for the shaping experiments in Section 7.3 and 7.4 is that applying image and XML shaping to large payloads will result in decreased latency and jitter and consequently improved soft real time performance, which should result in fresh survivor information arriving in a timely, prompt manner. Tests were run on three nodes (a publisher node, subscriber node, and a QED Phoenix services node) with dual core 2.8Ghz Intel Xeon processors with 1 GB RAM running Fedora Core 10 and connected via gigabit Ethernet on Vanderbilt ISISlab (www.isislab.vanderbilt.edu), which is a cluster running a version of the Emulab system that provides the ability to restrict bandwidth between nodes and emulate other network conditions where necessary.

7.1 CPU Overload Experiments

Even in quad-core processor systems, CPU usage can often spike to 100% utilization and remain there indefinitely, given a particular system load. The ability of a middleware system to be able to respond to this type of situation and prefer processing of important information is extremely valuable – especially in a real time or mission critical environment. The CPU overload experiments evaluate QED's ability to enforce information importance, informed by user-defined policies.

Setup. We analyze two experiments involving CPU overload. The first scenario mimics a situation where a high importance publisher (1hz) and medium importance publisher (1hz) are being overwhelmed by traffic from a low importance publisher (300hz). The second scenario presents a situation where high, medium, and low importance publishers are overwhelming the system equally, and we want to see if the middleware properly differentiates between them.

The metadata for each task submitted to the system is being evaluated against an XPath expression to drive CPU usage to 100% (consequently, evaluating tasks according to user-defined importance makes sense here). We evaluate two different middleware implementations: a baseline version of the Phoenix architecture for reference and a QED version.

Analysis of results. Figure 8 shows the results of the first CPU overload scenario, where high and medium importance publishers are attempting to publish at 1hz while a low importance publisher is overwhelming the system at 300hz, the middleware is able to differentiate high importance traffic at 74% of its optimal. Medium importance traffic is similar to the baseline performance, however. Ideally, both high and medium importance publication rates should trend towards their theoretical limits of 1hz. This scenario outlines room for improvement in the QoS enforcement mechanisms.

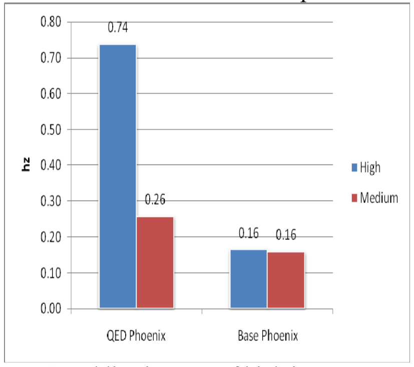


Fig. 8. Publication rate of high importance and medium importance information in the first tested scenario (1hz high, 1hz medium, and 300hz low).

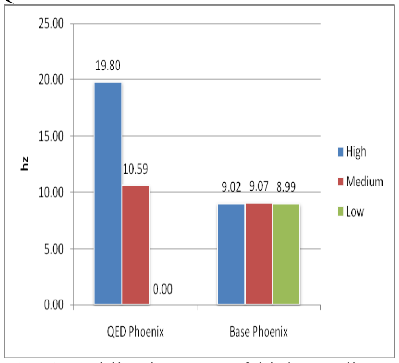


Fig. 9. Publication rate of high, medium, and low importance information in the 2nd scenario (20hz high, 20hz medium, and 20hz low).

Figure 9 shows the results of the second CPU overload scenario, where each high, medium, and low importance publisher is operating at 20hz with cpu-intensive tasks, the high and medium importance traffic is differentiated according to the user's wishes. High importance traffic is getting through at nearly the optimal hz (19.80hz), and medium importance information is able get through in the remaining system capacity (10.59hz). No low importance traffic gets through. The differentiation is significant compared to the Base Phoenix implementation, which lacks QoS features.

7.2 Bandwidth Bound Experiments

Setup. Bandwidth is constricted between the subscribers and the Phoenix node to just 320kbit (40KB). We place 3 publishers operating at 1hz with 1KB payloads on the publisher node and 15 subscribers on the subscriber node. Each subscriber is interested in all information from all publishers, causing a large amount of information to be disseminated through a constricted network connection. We evaluate two different middleware implementations: a baseline version of the Phoenix architecture for reference and a QED version.

Analysis of results. The results for this scenario in Figure 10 Show how the QED Phoenix middleware correctly differentiates the delivery of traffic by importance, according to the user-specified policy file. High importance information is preferred over medium importance information during dissemination, and very few low importance information get through, due to bandwidth constraints.

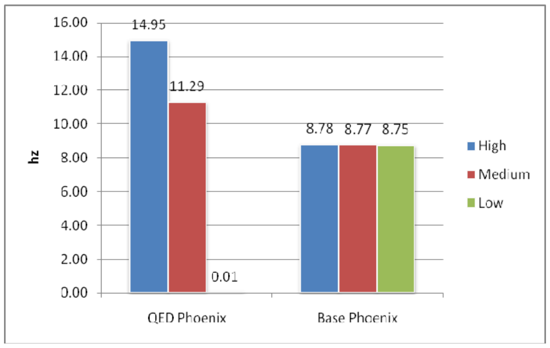


Fig. 10. Delivery rate of high, medium, and low importance information in the bandwidth bound scenario.

7.3 Image Shaping Experiments

We next present experiments to show QED's capability to shape image payloads into smaller images in order to attempt reducing message latency and jitter. For simplicity and analyzability, we did not form long filter chains of image shaping operations. Instead, we analyzed the available image shaping operations individually to show overhead associated with each type of filter.

Setup. Each experiment takes a 280KB USGS satellite image and attempts to convert it into a 35KB–40KB image using scaling, cropping, or compression. We also report overall data throughput rate, which can be an important indicator of system performance. Tests were run on two nodes (a publisher and subscriber) with dual core 2.8Ghz Intel Xeon processors with 1 GB RAM running Fedora Core 10 and connected via gigabit Ethernet. Publication hertz was stepped from 5hz to 10hz to show differences in overhead. All tests were run for 6 minutes.

Analysis of results. Figures 11-16 show results of the various experiments outlined in setup. Quality indicates image compression, Crop indicates cropping an image to a target size around its center, and Resize indicates scaling an image to a particular size. Quality 1 resulted in shaping the image down to 38KB, Crop 480x360 resulted in shaping the image to 36KB, and Resize 480x360 resulted in shaping the image to 40KB. Jitter was calculated as the standard deviation of latency.

The results for unshaped data were not included in the results graphs due to scaling issues. Latency and jitter were orders of magnitude larger for unshaped data versus even the worst case of shaped data (compression). Overall payload delivery rate, however, was also slightly higher. In general, the smaller the resulting payload size, the lower the latency and jitter. The overall payload data throughput goes down, however, because we are sending more packets (and thus suffer more overhead from metadata). Lower overall data throughput may also be an artifact of using pure Java libraries and deep copies of several 40KB payloads vs. one deep copy of 280KB having different performance profiles.

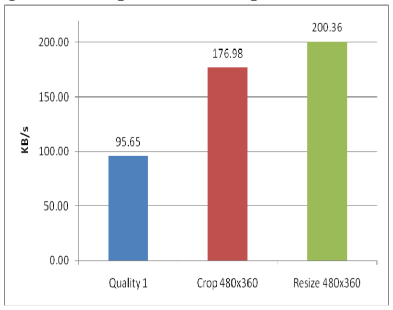


Fig. 11. Payload data throughput rate for image shaping operations at 5hz publication rate.

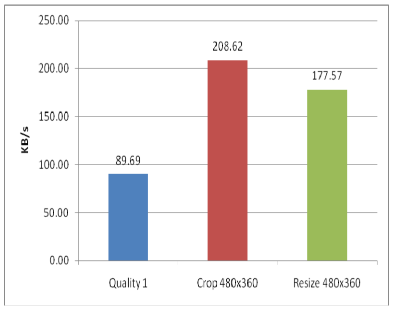


Fig. 12. Payload data rate for image shaping operations at 10hz

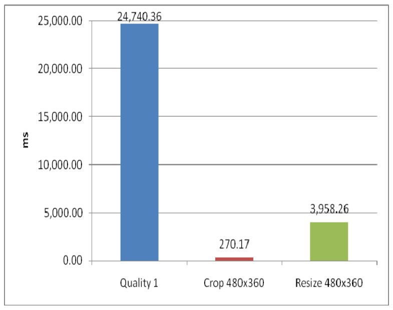


Fig. 13. Average Latency for image shaping operations at 5hz.

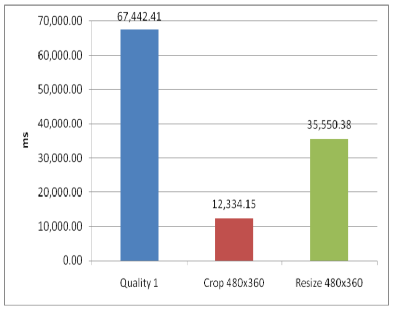


Fig. 14. Average Latency for image shaping operations at 10hz

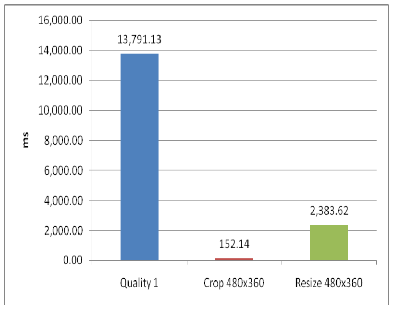


Fig. 15. Jitter for image shaping operations at 5hz.

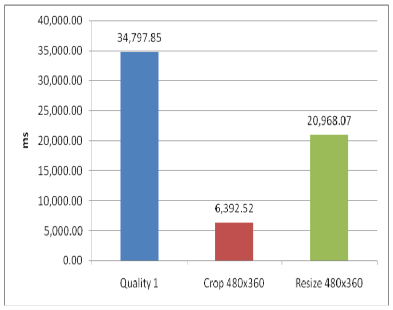


Fig. 16. Jitter for image shaping operations at 10hz.

To meet soft real-time constraints (as required in the motivating scenario), we focus more on data timeliness (latency and jitter) than on overall throughput. In a search and rescue mission, being able to see four images of potential survivors in smaller pictures (but still being able to distinguish landmarks) can be invaluable compared to middleware only being able to support viewing a single, large image showing one of four survivors in needs of rescue.

Another observation from these results is that each image operation is not created equal. Compressing an image to the same size as a resize produces an order of magnitude more overhead and unpredictability than cropping. As we scale up the publish rate of the publishers from 5hz to 10hz, the differences between compressing and cropping become even more pronounced. Resizing the image presents more overhead than cropping as well, but is more likely to result in a usable image (i.e., we will be more likely to recognize landmarks in the area) since cropping discards parts of the image completely.

7.4 XML Shaping Experiments

Below we present experiments which shape XML documents into smaller ones by selectively copying over elements via XSLT based on values of elements (in this case we key off of the "Priority" of the XML element).

Setup. Each experiment took a 300KB XML file containing a list of tracked areas (following our motivating scenario outlined in Section 2), and each area contained four fields: priority, image size, coordinates, and description. An XSLT was constructed to reduce the file into ½ or ¼ of its previous size. The following code shows an example XSLT template which reduces the 300KB file to 72KB. Experiments were conducted at 50hz on the same hardware indicated in Section 5.1 setup.

```
<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>
```

Analysis of results. Figures 17-19 show results of the various XML experiments outlined in setup. The original document was shaped down to half (148KB) and fourth (72KB) sizes. Jitter was calculated as the standard deviation of latency. The outcome of these particular XSLT template transformations is similar to the performance differences in image shaping. The smaller the transformed payload, the better the latency and jitter, but XSLT transforms appear to result in much more predictable soft real time performance versus image shaping.

This result is likely caused by image shaping being a more processor intensive operation than simply copying text elements from an XML document. Performing image shaping with vector processors (like those found in IBM's Cell processor) may result in a better comparisons between image shaping and XSLT, but such hardware was not available to us at the time of the writing of this paper. Our hypothesis, however, was that we could improve latency and jitter through policy-driven XML shaping before dissemination, and these results validate that hypothesis in all XSLT test scenarios.

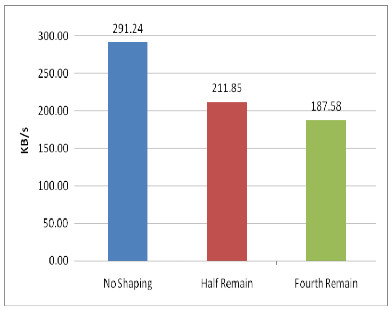


Fig. 17. Payload data rate for xml shaping at 50hz.

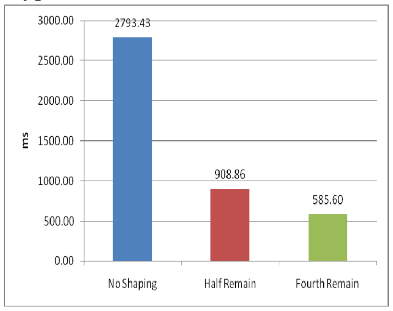


Fig. 18. Average latency for xml shaping at 50hz.

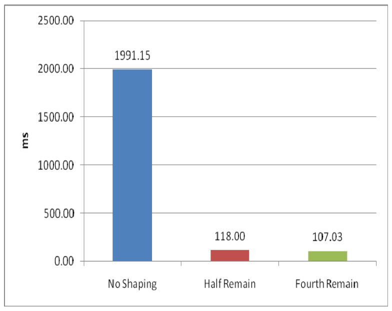


Fig. 19. Jitter for xml shaping at 50hz.

8 Related Work

This section compares QED's policy language support with related work. Our approach to QoS specification, dissemination, and enforcement offers more application-level and aggregate QoS support than offered by QoS parameters in conventional distribution middleware, such as the Data Distribution Service (DDS) and Java Message Service (JMS). It also offers better support for overall QoS than existing services-based policy languages, such as XACML [9].

XACML 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 not sufficient by itself to provide application, aggregate, or mission-level QoS.

What this means is that the PEP-PDP dataflow in the XACML specification language is 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.

In contrast, 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.

Similarly, conventional data dissemination middleware, such as DDS and JMS, provide parameters and features that control aspects of QoS in-line during information dissemination. QED complements these and supplement them with the application level policy, client QoS preferences, and aggregate QoS management that these middleware packages lack.

9 Concluding Remarks

The ability to differentiate QoS according to rich, domain-specific contexts is critical to providing data dissemination middleware that is capable of assuring that the most mission-critical information gets through and is resilient to overload situations [10]. This paper presents a policy-driven approach to QoS management called QED that works with existing data dissemination middleware, such as the JMS and DDS. Our experience demonstrates the importance of monitoring resource utilization to detect and characterize overload situations and respond with appropriate filtering and shaping to maintain service for the most important operations, users, and information.

Empirical results from our QED prototype show the effectiveness of a top-down approach to QoS policy specification and enforcement through middleware components that map mission-oriented system contexts and preferences to a variety of low-level settings that can be efficiently and independently enforced. QED is complementary to existing lower-level QoS capabilities, such as DDS, and provides a framework through which these and service-based QoS capabilities (such as priority task queuing and dissemination bandwidth management) can be orchestrated by an intermediate component to achieve system-wide enforcement of client preferences. The QED QoS policy specification and SOA components described here are extensible and future work includes providing richer and more dynamic context descriptions, such as the distance between a rescue crew and the survivors in our motivating scenario.

References

1. E. Crawley, R. Nair, B. Rajagopalan, H. Sandick, “A Framework for QoS-based Routing in the Internet,” IETF Internet Draft, RFC 2386, August 1998.
2. 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,” In Proceedings of the SPIE Conference on Defense Transformation and Net-Centric Systems, Orlando, FL, April 13-17, 2009.
3. J. Loyall, M. Gillen, A. Paulos, L. Bunch, M. Carvalho, J. Edmondson, P. Varshneya, D. Schmidt, A. Martignoni III, “Dynamic Policy-Driven Quality of Service in Service-Oriented Systems,” IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC), Carmona (Parador de Carmona), Spain, May 5-6, 2010.
4. J. Loyall, A. Sinclair, M. Carvalho, A. Martignoni III, M. Gillen, L. Bunch, M. Marcon, “Quality of Service in US Air Force Information Management Systems.” Proceedings of MILCOM, Boston, MA, October 18-21, 2009.
5. R. Grant, C. Combs, J. Hanna, B. Lipa, and J. Reilly, "Phoenix: SOA Based Information Management Services," Proceedings of the 2009 SPIE Defense Transformation and Net-Centric Systems Conference, Orlando, Fl, April 2009.
6. A. Uszok, J.M. Bradshaw, M. Breedy, L. Bunch, P. Feltovich, M. Johnson, H. Jung, “New Developments in Ontology-based Policy Management: Increasing the Practicality and Comprehensiveness of KAoS,” Proceedings of the 2008 IEEE Conference on Policy, Palisades, NY, June 2-4, 2008.
7. World Wide Web Consortium, OWL Web Ontology Language Overview, W3C Recommendation, February 10, 2004. www.w3.org/TR/owl-features/.
8. M. Carvalho, A. Granados, W. Naqvi, A. Brothers, J. Hanna, and K. Turck, “A Cross-Layer Communications Substrate for Tactical Information Management Systems,” Proceedings of MILCOM, IEEE, San Diego, CA, 2008.
9. 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, 1 February 2005.
10. C. Wang, G. Wang, A. Chen, and H. Wang, "A Policy-Based Approach for QoS Specification and Enforcement in Distributed Service-Oriented Architecture," IEEE International Conference on Services Computing (SCC'05) Vol-1, 2005.