QoS Allocation Algorithms for Publish-Subscribe Information Space Middleware

Joseph Loyall, Matthew Gillen, Praveen Sharma

BBN Technologies
Cambridge, MA

Abstract. Information spaces have emerged as a powerful concept for providing managed exchange of information between members of communities of interest (COIs), including information brokering and dissemination by publish-subscribe-query middleware. To support COIs with real-time or critical information exchange requirements, information spaces require quality of service (QoS) management algorithms that consider the complex system dynamics within information spaces, that allocate multiple resources, and that scale to information spaces of reasonable size. This paper presents two algorithms for multi-resource QoS allocation within information spaces. The first algorithm always provides an optimal allocation and includes optimizations that enable it to scale to information spaces of moderate size. The second algorithm is an approximation algorithm that provides near optimal solutions in most situations and scales to much larger information spaces. The paper also presents analyses and experimental results of the effectiveness and efficiency of the algorithms.

Keywords: Quality of service, multi-resource allocation, publish-subscribe-query information spaces

1 Introduction

The concept of i*nformation spaces* has emerged to support information exchange within communities of interest (COIs), collections of users that are related by shared interests or participation in a common mission [23]. Information spaces consist of the following:

- Middleware services for brokering and managing information exchange
- A collection of information producing and consuming clients
- The clients' shared vocabulary
- The set of managed information objects (MIOs) that clients exchange [2].

This work was supported by the USAF Air Force Research Laboratory under contract FA-8750-05-C-0267.

In the information space model [12], *clients* are information publishers and consumers, communicating anonymously with other clients via an *information management system (IMS)* [2] with managers that monitor and control the information space. Information published into the information space is in the form of typed managed information objects (MIOs) consisting of payload and metadata. Consumers make requests for future (*subscriptions*) or past (*query*) information using predicates over MIO types and metadata values. Information spaces provide topic-based information exchange, brokering, discovery, and shared understanding [12]. Clients do not need to be aware of one another, the source of information they consume, or the consumers of information they publish.

The IMS that we utilize in this work, *Apollo* [24], builds upon work in distributed object, component, and service oriented middleware. It provides a set of services that allow the registration of subscription predicates (specified using XQuery [26]), matching of metadata for published MIOs (specified using XML [27]), and delivering matched MIOs to clients using the Java Message Service [20]. Client-side distribution middleware exposes publication, subscription, and query interfaces conforming to the *Joint Battlespace Infosphere Common API (CAPI)* [10] using SOAP messages over HTTP or HTTPS.

We have developed quality of service (QoS) management middleware for information spaces with dynamic interoperability and real-time requirements. Our QoS management capability extends existing IMS middleware to manage the production, delivery, and consumption of information that meets client needs within available resources, to mediate competing demands for resources, and to adjust to dynamic conditions. Our *QoS Management System (QMS)* middleware, illustrated in Figure 1, builds upon our previous work in QoS management for distributed object and component systems [7, 15, 16, 17, 18, 19, 28]. The QMS is multi-layered middleware, described in more detail in [14], with an information space QoS manager (ISQM)[1] that provides aggregate QoS allocations and policy for clients and operations throughout an information space. The ISQM is collocated with the information brokering service and provides policy to local QoS managers

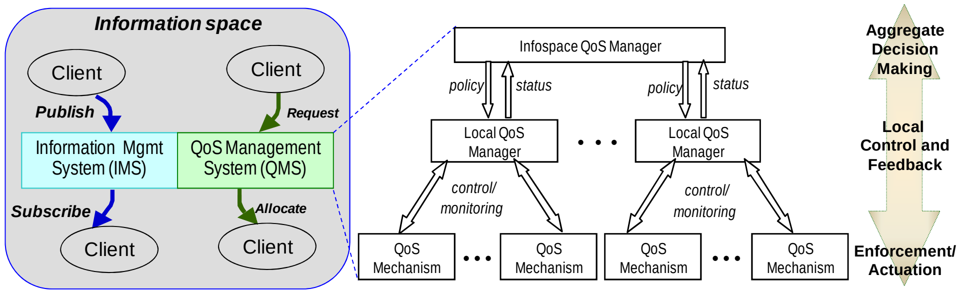

Fig. 1. The QMS layered architecture provides QoS management for an information space IMS.

[1] The ISQM is called a *System Resource Manager (SRM)* in [14], a historical term that is not as accurate with regard to its function. Likewise, the LQM element is referred to as a local resource manager (LRM) in that document.

(LQM), each of which enforce the policy at a local control point, making local decisions as necessary to achieve and maintain the desired QoS. The QMS also includes QoS mechanisms that control and monitor resource usage and shape information elements under control of an LQM. The QMS manages QoS in dynamic information spaces with clients that come and go, and goals, roles, and priorities that change with time and circumstances.

One of the challenges of providing QMS middleware is the development of algorithms for allocating QoS levels and their associated resources among the varying numbers of clients, operations, and applications using an information space. These *Multi-Resource QoS (MRQ) allocation algorithms* must consider the complex system dynamics of information spaces, be efficient enough to be used in real-time QoS management, and scale to the sizes of envisioned information spaces. Multi-resource QoS allocation is NP-hard[2], partially because of the following characteristics:

- There are complex system dynamics among the QoS needs within an information space. That is, how one resource is allocated can impact the demand positively or negatively for other resources. For example, a client who is interested in compressing information to lower bandwidth usage may require a higher amount of CPU.
- There is frequently no direct correlation between how important an application is and the amount of resources it needs.
- The relative ordering of QoS levels does not necessarily reflect the relative amount of resources that each level uses. That is, a higher QoS level (e.g., with higher precision, rate, or accuracy of information exchange) does not imply more resource usage than a lower QoS level and, in fact, might use more of some resources and fewer of others.
- Resource bottlenecks can change dynamically. That is, addressing a bottleneck caused by a highly constrained resource can result in a bottleneck in another resource.

This paper describes a set of multi-resource QoS allocation algorithms that we have developed for use within our prototype QMS. The MRQ algorithms are used by the information space QoS manager to select aggregate QoS allocations that are then enforced and maintained by the local QoS managers. The ISQM runs the algorithms and selects new QoS allocations when there are significant changes in the information space situation (e.g., change in the number of clients, missions, or resource availability) or when the LQM cannot locally keep the QoS behaviors within the constraints indicated by the ISQM. The ISQM's MRQ algorithms select QoS levels for clients in information spaces based on a benefit/cost ratio, i.e., the amount each choice increases the overall *utility* of the information space (the benefit) compared to the number and amount of resources that it uses (the cost). The algorithms described in this paper consider discrete QoS levels for each *control point* or *application* (terms that we use interchangeably), attempting to maximize utility across the entire information space within the available resources.

Because multi-resource QoS allocation is NP-hard, there is a tension between optimality and timeliness in the algorithms. Optimality refers to the ability of an algorithm to pro-

[2] Lee et al have reduced the problem to the 0-1 knapsack problem [11].

duce the highest utility QoS allocation possible within the available resources. Timeliness refers to the amount of time needed to determine a QoS allocation. For the class of MRQ problems, one can arrive at an optimal solution by examining a search space of all combinations of the applications and all the QoS levels in which they can operate, but examining this search space can take exponential time.

In this paper, we describe two algorithms for multi-resource QoS allocation in information spaces that manage the tradeoff between optimality and timeliness in different ways:

- *Optimizing Brute-Force* always provides an optimal solution but potentially runs in exponential time. The algorithm includes two optimizations that can prune the search space and reduce the runtime significantly in some situations.
- *Greedy Approximation* is an approximation algorithm based on 0-1 integer programming. The algorithm produces a near optimal allocation in many scenarios and runs in polynomial time.

We evaluate each algorithm's efficacy (how close to optimal the allocations computed by the algorithms are) and efficiency (how quickly the algorithms can produce an allocation).

The rest of this paper is organized as follows. First, we describe the MRQ algorithms, including an analysis of their efficiency. We then present our efficacy and efficiency experiments, including the experimental setup and metrics. Following this, we present some related work. Finally, we summarize our results.

2 Information Space QoS Allocation Algorithms

The MRQ algorithms that we present in this section select an allocation of QoS levels for all control points in the information space. Each *control point* represents a logical set of related points at which QoS can be affected, such as the information consumption, processing, and production for a single application[3]. The algorithms consider the resources needed by each QoS level at each control point[4], and attempt to maximize a measure of overall benefit (i.e., a *utility function*) defined for an information space within the available resources. One can determine an optimal solution by examining a search space of all possible allocations, but this is an exponential search in general and infeasible for all but

[3] Although each of these (consumption, processing, and production) can be controlled separately, choices made at each will affect the others. Thus they require a consistent logical QoS level, e.g., the rate and format of data inputs (consumption) must take into account the speed of information processing and production.

[4] The algorithms need the list of applications, their QoS levels, and their resource usage as input. The QoS levels should be defined to represent the QoS characteristics of most importance to the end user, from the most desirable level of QoS to the least acceptable level of QoS. The resource usage can be determined by off- or on-line profiling, or by analysis in some cases (e.g., bandwidth used by a periodic publisher can be calculated by multiplying the number of information objects per second that are published times the size of each object).

modestly sized information spaces. Therefore, we took two simultaneous approaches: (1) developing optimizations that can reduce the search space, and (2) developing an approximation algorithm that runs in less than exponential time in the worst case. This results in an *Optimizing Brute-Force* algorithm that produces optimal solutions and a *Greedy Approximation* algorithm that produces approximate solutions but runs in polynomial time.

The goal of each MRQ algorithm is to select an *allocation* of QoS levels for applications that simultaneously:

- Is *feasible*, i.e., fits within the resources available in the information space. An infeasible allocation cannot be deployed and hence is not an acceptable solution.
- Maximizes information space *utility*, i.e., allocates the applications of most importance to the overall COI goals and provides higher QoS where it is most useful to the COI.

The utility for any given client corresponds to a higher perceived user perception, which generally increases as throughput and information quality (e.g., resolution, precision) increase and as latency and jitter decrease. However, when tradeoffs must be made, particular QoS attributes will be more desirable than others and these tradeoffs are captured in the sets of QoS levels for each client. For example, a user that is watching video is willing to sacrifice some initial latency (for buffering) for a significant decrease in jitter. The QoS levels for that user would attach a much higher utility value to a level that introduced some delay but maintained a steady rate than to one with lower delay but greater variance in the rate. For the overall information space, the utility function must combine the utilities for the levels of each of the information space, but also attach a greater weight to the more important users. That is, just as the least important attributes for a given user should be degraded when necessary, the ISQM should degrade QoS for the least important users when necessary. While the best utility function to use can vary for given situations, goals, or domains, a reasonable utility function to use for information spaces is one that calculates utility based on the criticality of the applications that are run and the QoS level at which they are run. That is, the utility is increased by any of the following factors: (1) running more applications (i.e., servicing more clients), (2) running higher priority applications, and (3) running any application at a higher QoS level. For an information space with A applications, we define utility as follows:

$$Utility = \sum_{i=1}^{A} \left(w_c C_i\right)\left(w_q Q_i\right) \tag{1}$$

where:

- C_i *(>= 0)* is the relative criticality of application *i* compared to other applications.
- Q_i *(>= 0)* is the relative quality of QoS level *i* compared to other QoS levels for the same application or control point.
- w_c and w_q are weighting factors (to control the tradeoff of running more applications or applications at higher QoS levels).

The feasible allocation with the highest utility is considered the *optimal* allocation. Notice that there could be multiple allocations with equal utilities, so there could be multiple optimal solutions. For the experiments described in Section 3, we use a scenario generator that generates utility measures for each combination of application and QoS level, simulating in one value the criticality, QoS level value, and relative weights of these terms.

The above utility function and our experiments do not explicitly consider resource efficiency, so that two allocations could have equal utilities even if one uses fewer resources than the other[5]. However, keeping resources in reserve could lead to more effective QoS management in dynamic information spaces because wholesale reconfigurations will be reduced if there are resources available to handle overload situations or the addition of new applications. We accomplish this by adding a *reserve factor* to the utility function, i.e., a numerical measure of the benefit for having resources available, as follows:

.
$$Utility = \left(\sum_{i=1}^{A} (w_c C_i)(w_q Q_i) \right) + w_r R \qquad (2)$$

where R is a measure of the resources available, and w_r is a (non-negative) weighting factor to control the tradeoff of using available resources now to run more applications (or higher QoS levels) or keeping the resources in reserve.

2.1 The Optimizing Brute-Force Algorithm

The Optimizing Brute-Force algorithm searches a combinatorial decision tree built from the control points and their QoS levels. As depicted in Figure 2, each level of the decision tree represents a control point (e.g., CP-a, CP-b, CP-c, etc.) and each branch represents a QoS level choice at its parent's control point (e.g., CP-a has QoS level choices 1 and 2, CP-b has QoS level choices 3 and 4, and so forth). Each non-leaf node represents an allocation of control points and QoS levels for the nodes above it in the graph (e.g., CP-b0 represents an allocation of QoS level 1 to CP-a, while CP-c2 represents an allocation of QoS level 2 to CP-a and QoS level 3 to CP-b). The leaf nodes represent combinations of an entire set of control points and QoS levels (i.e., the complete set of potential allocations) in an information space.

Without optimizations, a brute-force search would traverse the tree recursively and examine each leaf node for feasibility and utility. If a node is feasible, its utility is compared with the highest utility of previously evaluated feasible solutions. If the utility of the node is higher, then it becomes the new best solution. The best solution after evaluating all the leaf nodes is the optimal allocation, i.e., the feasible solution with the highest utility.

[5] However, resource efficiency is considered by the Greedy Approximation algorithm's *effective gradient* computation, described in Section 2.2.

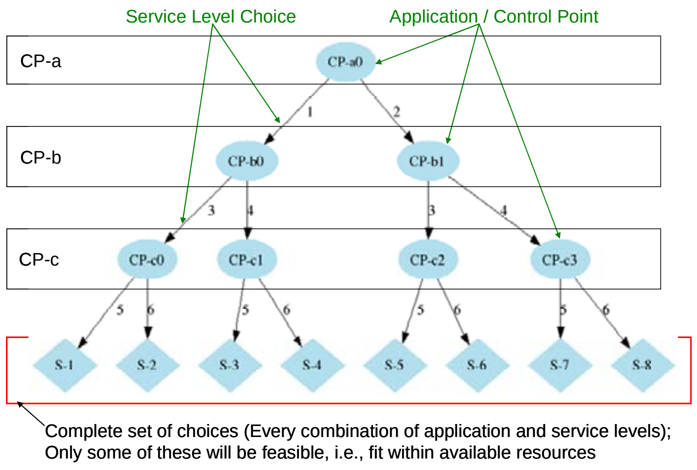
Service Level Choice
Application / Control Point
CP-a
CP-b
CP-c
CP-a0
CP-b0
CP-b1
CP-c0
CP-c1
CP-c2
CP-c3
1
2
3
4
5
6
S-1
S-2
S-3
S-4
S-5
S-6
S-7
S-8
Complete set of choices (Every combination of application and service levels);
Only some of these will be feasible, i.e., fit within available resources

Fig. 2. Decision tree that the Brute-Force algorithm creates and traverses to allocate resources.

The brute-force search with no optimizations runs in $\Theta(q^a)$ where q is the number of QoS levels for each control point[6], and a is the number of control points[7]. For the Optimizing Brute-Force algorithm, we use the following optimizations to prune the search space, significantly in some cases.

Pruning Using an Infeasibility Check. This optimization utilizes the fact that as the algorithm traverses down from the root node to leaf nodes, the number of applications and QoS levels represented in the nodes increases. Consequently if the partial allocation represented by any non-leaf node is not feasible (i.e., it requests more resources than are available), then all the nodes in the subtree under the non-leaf node are also infeasible (because each will add applications to the already infeasible partial allocation). The entire subtree can be bypassed. This optimization works well (i.e., it leads to significant pruning) when many of the leaf nodes represent infeasible allocations.

Pruning Using a Utility Check. This optimization utilizes the fact that as the algorithm traverses down from the root node to leaf nodes (increase in depth of a tree), the utility associated with each node will be more than that of its parent node. At each point in the traversal of the tree, the algorithm walks the path of highest utility first (essentially following the branches of highest QoS levels whether they are feasible or not). If the leaf node reached is lower utility than the best solution reached so far, the entire subtree can be pruned, since all other paths would lead to even lower utility. This optimization works well when the algorithm finds a high utility feasible solution early, enabling pruning of many subtrees with lesser utility.

[6] Assuming the same number of discrete QoS levels for each application.

[7] Θ notation is a tight upper and lower bound on the algorithm execution, i.e., the algorithm will check every node of the tree, i.e., exactly q^a *nodes.*

The Optimizing Brute-Force algorithm uses both of the above optimizations together, along with ordering the tree to maximize the pruning possible. However, in the worst case, the algorithm finds many feasible nodes and relatively low utility solutions, resulting in little or no pruning. In these cases, the algorithm may still end up examining nearly the entire tree. Therefore, the Optimizing Brute-Force algorithm is $O(q^a)$.

2.2 The Greedy Approximation QoS Management Algorithm

Our Greedy Approximation algorithm is based on a 0-1 integer programming algorithm in [21]. 0-1 integer programming tries to maximize the *objective function*:

$$\sum_{i=1}^{m} p_i x_i \tag{3}$$

subject to

$$\sum_{i=1}^{m} H_{ij} x_i \le L_j \tag{4}$$

for j = 1, 2, …, n, where:

- Each x_i is an application at a particular QoS level
- p_i is the priority of the application
- H_{ij} is the resource usage of x_i
- L_j is the vector of the capacity of the resources
- m is the number of applications and n is the number of resources.

Our Greedy Approximation algorithm greedily allocates QoS levels to applications contending for resources using an *effective gradient* measure, a ratio of the benefit that each application provides and the cost that it incurs. The algorithm measures the benefit of an application as the value it contributes to the objective function above. It measures the cost of an application at a particular QoS level as the amount of resources requested. The algorithm aggregates the resources into a single dimension and assigns a *penalty* to increase the cost associated with requesting a highly contended resource (i.e., a resource for which a significant amount has already been allocated to other applications).

Our algorithm extends the algorithm in [21] in the following ways:

1. We have two variable dimensions that need to be considered. Each application can have multiple QoS levels from which to choose. In the algorithm in [21], each application has one service level. Our algorithm treats each combination of application and QoS level as a separate viable choice, while ensuring that only one QoS level can be chosen for each application.

2. We compute an initial penalty vector for resource usage. The algorithm in [21] only computes penalties as the algorithm progresses, which can lead to significantly suboptimal allocation. That is, it treats all resources equally and completely available at the beginning. In reality, some resources are more likely to become bottlenecks (e.g., because more applications request them or applications request higher amounts of them) than others. Our algorithm performs an initial pass and assigns an initial penalty to resources, making it cost more to request highly contended resources.
3. We guarantee a solution by including a *starvation* choice at each level, i.e., a QoS level that uses no resources and provides no benefit and represents starving a particular application if there are not enough resources to run it at any level.

After computing the initial penalty, the greedy approximation algorithm computes the total number of application-QoS level combinations as described in 1 above. It iterates over the following steps until either all the applications have been assigned a QoS level or there are no more resources left to allocate to any remaining choices:

1. It computes the effective gradient for each application-QoS level combination as the ratio of benefit divided by cost. The benefit is the utility that a given application at a given QoS level provides. The cost is the resources requested adjusted by the penalty.
2. It selects the application and QoS level combination with the highest effective gradient and eliminates further consideration of the other QoS levels for this application.
3. It allocates the resources needed by the application and QoS level combination selected in step 2, removing those resources from the available resources.
4. It prunes the list of application-QoS level combinations of any infeasible choices.

2.2.1 Analysis of the runtime of Greedy Approximation.

Pseudocode for the Greedy Approximation algorithm follows:

```
1: initializeList(CP-QoSLevelList)
2: while (CP-QoSLevelList not empty) {
3:   next = find_max_gradient(CP-QoSLevelList);
4:   addToUsedResources(next.resourceUsage)
5:   removeChosenCP's Other Service Levels
6:   removeInfeasible(CP-QoSLevelList)
7: }
```

Step 1 is the creation of the initial penalty vector. It makes a single pass through the list of every control point and QoS level choice, *CP-QoSLevelList*, i.e., *a*q* elements where *a* is the number of control points and *q* is the number of QoS levels. The loop bounded by steps 2 and 7 is executed at most *a* times, since step 5 removes at least *q-1* elements from the list each time. Step 6 could remove more, so the actual number of times through the loop could be fewer than *a* times. Steps 3 and 4 are linear time operations on the current list of control points × QoS levels and resources, respectively.

Therefore, the worst case runtime is equal to *(aq) + a(arq)*, or $O(a^2qr + aq)$, where:

- *a* is the number of applications,
- *q* is the number of QoS levels, and
- *r* is the number of resources

Furthermore, notice that the operation in step 6 affects the runtime of future iterations. If step 6 prunes a significant number of infeasible allocations from the *CP-QoSLevelList*, then the number of times through the loop is significantly reduced. In scenarios where 100% of solutions are feasible, step 6 will never remove anything and the algorithm will run in worst case time. In scenarios where step 6 removes most of the elements because many allocations are infeasible, the algorithm will run much faster. Regardless, in worst case its runtime is polynomial or, more precisely, *quadratic* in the number of applications.

2.3 Applying the QoS Management Algorithms to Dynamic Information Spaces

As illustrated in Fig. 1, the algorithms described above are used by the ISQM layer of a multi-layered QoS management architecture. The ISQM uses the allocation algorithms to select a set of QoS levels to apply at the control points throughout an information space. The QoS levels are enforced at local control points by LQMs, which control the rate, size, processing, and other controllable attributes of information through the system.

The multi-layered approach also allows for QoS enforcement at different granularities of time. At the lowest layers, QoS mechanisms and LQMs maintain QoS levels by adjusting parameters like rate, compression level, and scaling factor as frequently as they need to, with feedback control to avoid thrashing. The execution of the QoS allocation algorithms and subsequent distribution of new QoS levels is expected to be much less frequent in general and associated with discrete events affecting the entire information space, such as changes in information space makeup (new clients or clients leaving), resource availability, or goals and priorities. In cases where the effects of changes can be limited, running the allocation algorithms and distributing new policies might be avoided altogether. For example, a new client that is relatively lower importance than other existing clients need not lead to recalculation of QoS levels for other clients. Likewise, if a client leaves, the resources that it is using can be kept in reserve rather than reallocating the information space, unless there is a critical need for higher QoS somewhere.

This motivates an important area for future research, namely that of limiting the effects of changes in allocations. That is, if a change to state occurs requiring the ISQM to run the QoS allocation algorithms to choose an allocation of QoS levels across the information space, it is desirable for the selected allocation to require as few changes at individual control points as possible. This means the ISQM needs to evaluate possible allocations not only in terms of their feasibility and utility values, but also in terms of their differences from the last deployed allocation. This is an area that we have not investigated fully yet.

3 Experimental Evaluation of the QoS Allocation Algorithms

We conducted a set of experiments to evaluate the relative performance of the algorithms, in terms of quality of the solution produced and the speed of execution to reach a solution. This section describes these experiments and their results.

3.1 Experimental Setup

We executed the experiments on a personal computer with a 2.80 GHz Intel® Pentium®-4 CPU with 512 MB RAM, running the Linux (Fedora Core Release 6) operating system.

We developed a *scenario generator* that randomly generates scenarios used as input to a *simulator* that we developed to execute the algorithms on the scenarios. Each scenario consists of a set of applications, a set of QoS levels for each application, a utility value for each QoS level, and a set of resources and amount used by each QoS level. The generator accepts the following arguments: the number of applications (control points) in the scenario, the number of QoS levels for each application, the total number of resources in an information space, and the number of resources (to be chosen from the total number of resources) for each QoS level. The generator produces a random value for utility for each combination of application and QoS level, randomly chooses the resources to use for each QoS level from among those available, and selects a random amount of each resource that is requested for each QoS level, generating a discrete uniform distribution of scenarios.

The simulator takes as input a set of scenarios, runs the MRQ algorithms on each scenario, and produces the solution allocation, the utility of the solution, the runtime of the algorithm, and values for the metrics described in Section 3.2.

In general, for each of the experiments described in this report, we use the scenario generator to generate a sizable set of scenarios with the following parameters: 3 QoS levels, 6 resources per QoS level, and 110 total resources. We varied the number of applications. For each application set, we generated 100 scenarios. For other experiments, we will describe the specific experiment design as we describe the experimental results.

3.2 Experimental Metrics

Algorithm Metrics. We collected the following metrics to compute the efficacy and the efficiency of the QMS algorithms:

- *Percent of Optimality:* The optimal solution is the feasible solution with the highest utility. For the solution returned by any algorithm, we compute its percent of optimality by dividing its utility by the utility of the optimal solution. For any given scenario, we use the utility reported by the Optimizing Brute-Force algorithm as the baseline against which the optimality of all the algorithms are compared.

- *Runtime:* We use the simulator to measure how fast each algorithm executes in our experiments. Although the absolute runtime depends on the hardware on which the algorithm is executed, the relative runtimes of various algorithms are comparable because we ran all our experiments on the same machine.

Contention Metrics. As part of our experiments, we evaluated the effect of *contention* on our algorithms, i.e., how resource rich or resource scarce the scenario is, and collected contention metrics to support this. We use the following contention metric in the experiments described in this paper:

- *Percent of infeasible solutions* measures the total number of infeasible solutions out of the total number of possible solutions (leaf nodes in the search tree created by the Optimizing Brute-Force algorithm). For example, the total number of possible solutions (i.e., possible allocations) for 10 applications and 3 QoS levels is 59,049 solutions. If only 200 solutions are feasible, we compute the percent of infeasibility as (59049-200)/59049. The percent of infeasibility is directly proportional to the level of contention, i.e., the higher the percentage of infeasible solutions, the higher the contention for resources in the scenario.

3.3 Percent of Optimality and Runtime of the Optimizing Brute-Force Algorithm

The Optimizing Brute-Force algorithm always produces an optimal solution (i.e., 100% optimality). Hence, we use this as the baseline algorithm for measuring the effectiveness of the other algorithms.

However, in the worst case Optimizing Brute-Force runs in exponential time. Furthermore, the runtime grows exponentially as either the number of applications or the number of QoS levels increase. Figure 3 shows *boxplots* of the results for an experiment in which we generated scenarios with the number of applications varying from 10 to 110 by steps of 10, with 100 scenarios at each step. Each application had 3 QoS levels, and each QoS level used 6 resources selected randomly from a total of 110 resources.

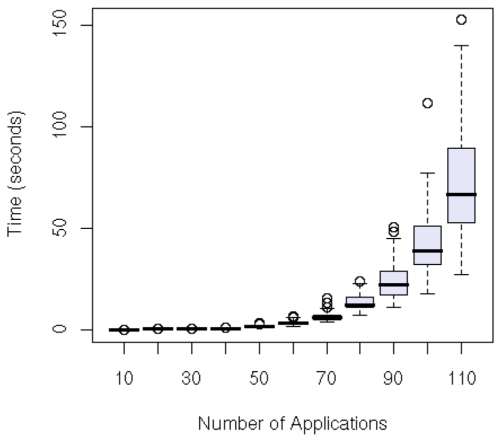

Fig. 3. Impact of varying the number of applications on the runtime of the Optimizing Brute-Force algorithm.

Boxplots [22] are a visual means of examining and comparing sets of data, regardless of their distributions, that readily indicates their medians, variance, and skew. As shown in Figure 3, the box of each dataset displays the interquartile range (IQR), i.e., the range from the first to the third quartile in which the middle 50% of data values lie. The thick black line in the middle of the box represents the median. Vertical lines extending out from the box and ending in horizontal bars, called whiskers, represent the extent of the (non-outlier) observed values. Circles beyond the whiskers represent outliers, i.e., values above 1.5 × IQR + the upper quartile value or less than -1.5 × IQR below the lower quartile value.

As Figure 3 indicates, the runtime is good (near one second) until about 40-50 applications, after which the median runtime and the variance in runtime increase dramatically. The median runtime increases to about 70 seconds at 110 applications, with a worst case runtime of 150 seconds and best case of about 30 seconds. The increased variance is due to the difference in pruning possible from scenario to scenario. The scenarios with the highest runtime allow little pruning, causing the Optimizing Brute-Force algorithm to search nearly the entire space. In contrast, the best measured runtime (about 25 seconds for 110 applications, 6× faster than the worst case time) are for scenarios that allow significant pruning (i.e., many infeasible solutions and/or quickly found high-utility solutions).

Figure 4 depicts the runtime of Optimizing Brute-Force when either the number of QoS levels or the number of applications increases. For this experiment, we generated scenarios that varied the number of QoS levels from 1 to 20 for each number of applications and that varied the number of applications from 1 to 20 for each number of QoS levels. The runtime is acceptable up to about 10 of either, then increases dramatically.

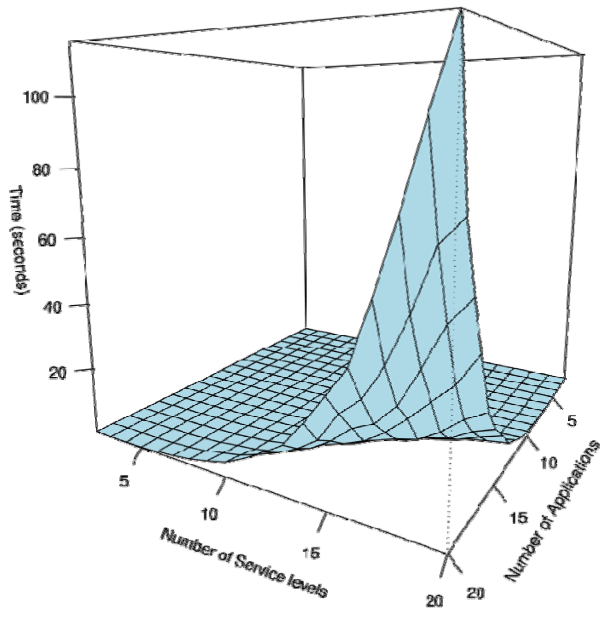

Fig. 4. The impact of simultaneously varying number of applications for a given QoS level and number of QoS levels for a given application when running the Optimizing Brute-Force.

3.4 Percent of Optimality and Runtime of the Greedy Approximation Algorithm

Our experiments indicate that the Greedy Approximation algorithm produces solutions that are close to optimal, with a significant improvement in runtime over the Optimizing Brute-Force baseline. The boxplot in Figure 5 represents an experiment in which we ran the Greedy Approximation algorithm on 50,000 scenarios, with 10 applications[8], 3 QoS levels for each application, 3 resources per QoS level, and 30, 70, 110, 150, and 190 total resources (10,000 scenarios for each level of total resources). The median solution is 96% of optimal (the thick black line near the top of the boxplot), and 75% of the solutions are over 90% of optimal or better (the grey box and above), with all but the outliers producing solutions 80% optimal or better. The worst solution is 40% of optimal.

In experiments designed to identify the source of the low optimality outliers, we determined that contention adversely impacts the effectiveness of Greedy Approximation. Specifically, we observed the median optimality decline to 75% as the level of contention increases significantly. Figure 6 illustrates experiments run with the number of applications varying from 10 to 110, 3 QoS levels, 6 resources per QoS level, and 20 total resources. In these experiments, the median percent of optimality varied between only 75-85%, although the worst case percent optimality is approximately the same as the experiment in Figure 5.

The difference in the number of resources being used by each application and the number of resources available causes the experiments depicted in Figures 5 and 6 to exhibit different contention characteristics. The experiments depicted in Figure 5 (selecting 3 re-

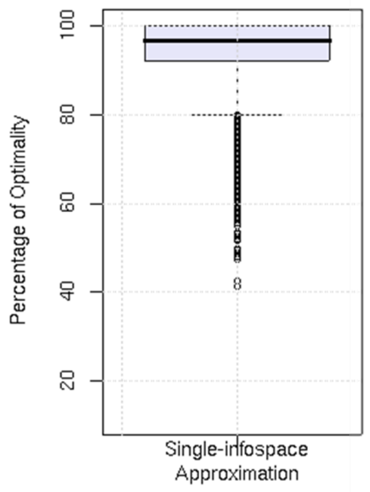

Fig. 5. Optimality of the Greedy Approximation algorithm on 50,000 scenarios with 10 applications, 3 QoS levels per application, 3 resources per QoS level, and 30, 70, 110, 150, and 190 total resources (10,000 scenarios each).

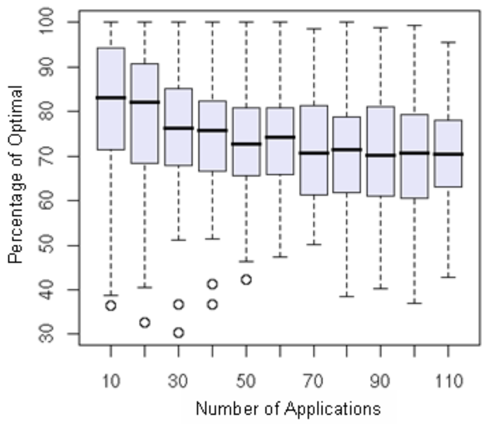

Fig. 6. Optimality of Greedy Approximation on 50,000 scenarios with a varying number of applications, 3 QoS levels per application, 6 resources per QoS level, and 20 total resources.

[8] We had to generate scenarios with a modest number of applications in order to have an optimality baseline against which to compare, since we have to run the Optimizing Brute-Force algorithm on each of the 50,000 scenarios to get the optimal solution.

sources from 30, 70, 110, 150, or 190 resources) had scenarios with the percentage of feasible solutions ranging from under 10% to 100%, whereas in the experiments depicted in Figure 6 (selecting 6 resources from 20 available), all of the scenarios had fewer than 0.5% feasible allocations. This provides evidence that the level of contention affects the optimality of the Greedy Approximation algorithm.

Effectiveness of the initial penalty optimizing factor. As described in Section 2.2, our Greedy Approximation algorithm uses an initial penalty vector. We introduced the initial penalty vector to handle a set of scenarios (that we dubbed *Greedy Achilles' Heel* scenarios) that produced sub-optimal solutions in the base algorithm (without the initial penalty). These scenarios have one or more high utility applications that request a significant amount of a highly contended resource. Since the algorithm without an initial penalty treated all resources equally and completely available at the beginning, these applications would be greedily assigned resources and potentially starve a large number of other applications resulting in a significantly suboptimal solution. To prevent this, we enhanced the algorithm to perform an initial pass and assign an initial penalty to highly contended resources, making it cost more to request these resources.

We conducted experiments to evaluate the effectiveness of the initial penalty enhancement. For this experiment, we generated Greedy Achilles' Heel scenarios with a varying number of applications, 3 QoS levels for each application, and 6 resources selected randomly from 110 resources for each QoS level. We varied the number of applications from 10 to 40 in steps of 10 (again, the upper bound of 40 allows us to run the Optimizing Brute-Force algorithm to get the optimal solution against which to compare). For each number of applications, we had 100 scenarios on which we ran the Greedy Approximation algorithm both with and without the initial penalty. The results show that the initial penalty improves the percent of optimality significantly for this class of scenarios. Without the initial penalty, Greedy Approximation provides a low median percent of optimality ranging from approximately 30% to approximately 42% (Figure 7). When we add the initial penalty to Greedy Approximation, the median percent of optimality on the same set of scenarios improved to a range of 75% to 85% (Figure 8). Notice that the percent of optimality declines as the number of applications increase in both cases, due to an increase in contention (more applications competing for the same number of resources).

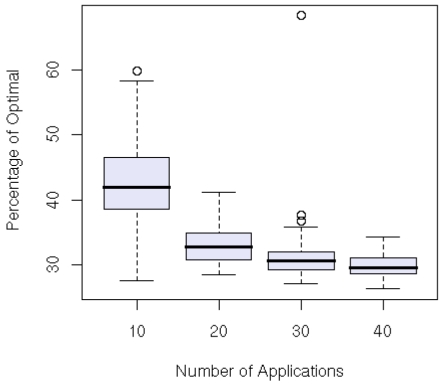

Fig. 7. Percentage of optimality of Greedy Approximation for Greedy Achilles' Heel scenarios *without the initial penalty optimization.*

The effect of the number of applications and the number of resources on the runtime of Greedy Approximation. We also ran experiments that varied the number of applications and the number of resources, the two scenario attributes that we believed might scale to large numbers in realistic scenarios. From the analysis in Section 2.2.1, we expected varying the number of applications to affect the runtime quadratically and varying the number of resources to affect the runtime approximately linearly.

For the experiment varying the number of applications, we increased the applications from 10 to 300 in steps of 10, with 100 scenarios for each discrete number of applications. Each application had 3 QoS levels, and each QoS level used 6 resources selected randomly from a total of 110 resources.

As expected from the analysis above, we observed that the runtime of Greedy Approximation increases polynomially with the increase in the number of applications (Figure 9). As comparison, executing the Greedy Approximation algorithm on a randomly generated scenario with 110 apps took less than 0.10 seconds versus 60 seconds for the Optimizing Brute-Force algorithm. We observed subsecond runtimes for up to hundreds of applications (0.6 seconds for 300 applications).

Figure 10 illustrates the results of an experiment to evaluate the effects of varying the number of resources. In this experiment, we randomly generated scenarios with 100 applications, 3 QoS levels per application, 3 resources per QoS level, and total resources varying from 30 to 180, in steps of 10. We generated 100 scenarios for each discrete number of resources. Our results indicate that the runtime of Greedy Approximation grows approximately linearly as the number of resources increases, as shown in Figure 10, confirming what we expected from the analysis in Section 2.2.1.

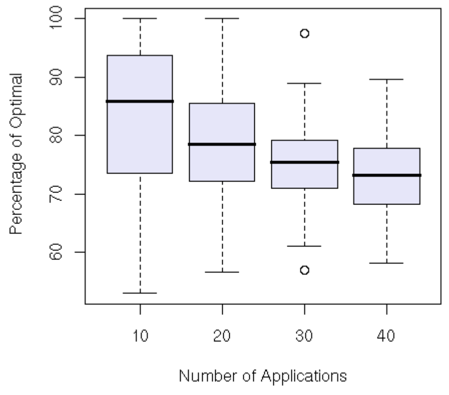

Fig. 8. Percentage of optimality of Greedy Approximation for Greedy Achilles' Heel scenarios *with the initial penalty optimization.*

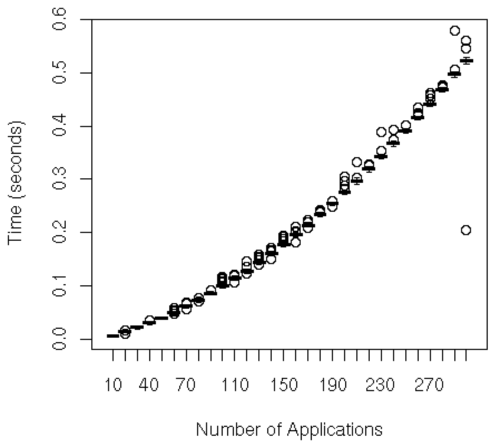

Fig. 9. Runtime of the Greedy Approximation algorithm as the number of applications increase.

4 Related Work

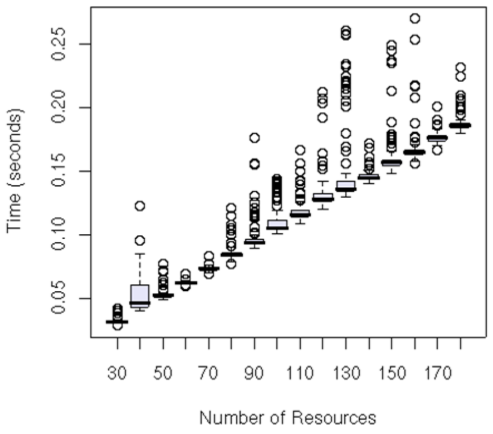

Fig. 10. Runtime of the Greedy Approximation algorithm as the number of resources increases.

The information spaces concept has grown out of the *Joint Battlespace Infosphere (JBI)* [2, 10], a US Air Force initiative supporting network centric warfare concepts. It is related to other network centric warfare initiatives, including the *Global Information Grid (GIG)* and *Net-Centric Enterprise Systems (NCES)*. The GIG will provide the communication, networking, and processing capability to enable the interconnection of warfighters, command personnel, and policymakers [4]. NCES is a set of services (based on Web Services [25]) enabling access to and use of the GIG in warfighting operations [3]. The JBI, as exemplified by the Apollo reference implementation, enables information exchange and management between tactical and enterprise users and is intended to interact with and use NCES services and the GIG as concrete instances emerge.

Dynamic programming [6] is another approach to solving multi-resource QoS allocation problems, treating them as 0-1 knapsack problems. In general, the runtime of this class of algorithms is pseudo-polynomial, or technically an exponential function of their input sizes [9]. This presents a quantization challenge for solving the problem using dynamic programming. A way to develop a polynomial time dynamic programming algorithm is to limit the sizes of the resources by normalizing them and choosing a quantization, i.e., a discrete unit of allocation for each resource. This results in resources being allocated in discrete units (e.g., tenths, hundredths, or thousandths). While this makes the algorithm run much faster, it reduces its effectiveness. For example, a quantization of 0.1 allocates resources in tenths of their total amount available (an application requesting 3% of a resource would get either 0% or 10%). The quantization also places a limit on the number of applications that can share a resource, e.g., a 0.1 quantization means that at most ten applications can share any resource. A finer grain quantization should improve the optimality of the solutions but will increase the runtime significantly. For example, a quantization of 0.01 will allow up to 100 applications to share each resource and will allocate resources in hundredths, but would increase the execution time of the algorithm by at least 10× over that for a 0.1 quantization. For some resources, this would still be a gross quantization. For example, a 100 Mbps link would be allocated in units of 1 Mbps and a 1 Gbps network link would be allocated in units of 10 Mbps. Our experiments showed significantly better efficacy and performance from the Greedy Approximation algorithm.

Einbu provides a method for solving the multi-resource allocation problem by mapping it to the Transportation Problem [5]. The method requires a strictly concave return function, i.e., a utility function in which the additional gain in utility from each additional amount of resources used becomes smaller as the number of resources used increases. While this might be true in many scenarios, it is a limitation that our algorithms do not require. The algorithm is guaranteed to terminate and produce an optimal solution. However, the paper does not analyze the computational complexity of the algorithm.

Xu et al present an algorithm for reserving multiple resources as service requests are made [29]. The algorithm creates a graph for each service request at runtime, with nodes representing QoS levels and edges representing feasible resource requirements. It then runs Dijkstra's shortest path algorithm to determine the suitable resource reservation. The algorithm can run in polynomial time on graphs with no cycles. To handle the more general case, which is NP-complete, the paper presents a two-pass algorithm with heuristics specific to the resource reservation domain. Gopalan and Chiueh present another heuristic algorithm based on tasks with time ordered use of resources [8]. While it is difficult to fully compare just based on the papers, these appear to be viable alternatives to the algorithms we present in this paper.

As an alternative to algorithms based on mathematical foundations or heuristics, Liu et al use a genetic algorithm approach to resource allocation [13]. This approach is based on task scheduling and produces an optimal allocation, but relies on sequential tasks and does not evaluate its runtime performance and suitability to run in-line in a dynamic system.

The work reported in this paper builds upon the authors' previous work in QoS for distributed object and component based middleware [14, 17, 19, 28] and previous work in resource management, such as the Darwin project at CMU [1].

5 Conclusions

For publish-subscribe information spaces to be useful for real-time and critical information exchange, they must include quality of service capabilities. However, traditional QoS mechanisms for resource allocation and differentiated services are not sufficient, unless they include algorithmic means to mediate the conflicting demands for QoS and aggregate QoS control over all the clients and operations of an information space.

In this paper, we have advanced the state of the art in middleware-based multi-resource QoS allocation by defining, evaluating, and prototyping a set of algorithms that allocate QoS levels and resources across large numbers of applications and control points within information spaces. The Optimizing Brute-Force algorithm provides optimal allocations in reasonable execution time for modest numbers of applications (subsecond response up to 40-50 applications in our experiments). The Greedy Approximation algorithm provides approximate solutions, but scales well, with a median of 96% optimality and demonstrated fast execution times to hundreds of applications with subsecond response in our experiments. Greedy Approximation has the fastest runtime, but farther from optimal solutions in highly contentious scenarios (defined by the number of feasible allocations). Conversely, it produces closer to optimal solutions, but takes more time to do so, when contention is low (i.e., there are many feasible solutions).

Under an ongoing effort with the US Air Force Research Laboratory, we are currently prototyping these algorithms as part of a practical application of multi-layered QoS management middleware for information spaces, which will give us the opportunity to evaluate these algorithms in the context of realistic scenarios.

References

1. Chandra, P., Fisher, A., Kosak, C., Ng, T.S., Steenkiste, P., Takahasi, E., Zhang, H.: Darwin: Resource Management for Value-Added Customizable Network Service. In: Sixth IEEE International Conference on Network Protocols (ICNP'98), Austin, TX, October 1998.
2. Combs, V., Hillman, R., Muccio, M., McKeel, R.: Joint Battlespace Infosphere: Information Management within a C2 Enterprise. In: The Tenth International Command and Control Technology Symposium (ICCRTS), 2005.
3. Defense Information Systems Agency, Net-Centric Enterprise Services, http://www.disa.mil/nces/.
4. DoD CIO, Department of Defense Global Information Grid Architectural Vision, Vision for a Net-Centric, Service-Oriented DoD Enterprise, Version 1.0, June 2007, http://www.defenselink.mil/cio-nii/docs/GIGArchVision.pdf.
5. Einbu, J.M.: A Finite Method for the Solution of a Multi-Resource Allocation Problem with Concave Return Functions. Mathematics of Operations Research 9(2), 232-243 (1984).
6. Giegerich, R., Meyer, C., Steffen, P.: A Discipline of Dynamic Programming over Sequence Data. Science of Computer Programming 51, 215-263 (2004).
7. Gill, C., Loyall, J., Schantz, R., Schmidt, D.: Experiences Using Adaptive Middleware in Distributed Real-time Embedded Application Contexts: a Dependability Perspective. In: Workshop on Dependable Middleware-Based Systems (WDMS), Part of Dependable Systems and Networks Conference (DSN 2002), Bethesda, Maryland, June 26, 2002.
8. Gopalan, K., Chiueh, T.: Multi-Resource Allocation and Scheduling with Real-Time Constraints. In: Multimedia Computing and Networking (MMCN ’02), San Jose, CA, January 18-25, 2002.
9. Hall, L.: Computational Complexity, The Johns Hopkins University, http://www.esi2.us.es/~mbilbao/complexi.htm.
10. The Joint Battlespace Infosphere website, http://www.infospherics.org/.
11. Lee, C., Lehoczky, J., Rajkumar, R., Siewiork, D.: On Quality of Service Optimization with Discrete QoS Options. In: Fifth IEEE Real-Time Technology and Applications Symposium (RTAS'99), 1999.
12. Linderman, M., Siegel, B., Ouellet, D., Brichacek, J., Haines, S., Chase, G., O’May, J.: A Reference Model for Information Management to Support Coalition Information Sharing Needs. In: The Tenth International Command and Control Technology Symposium (ICCRTS), 2005.
13. Liu, Y., Zhao, S.-L., Du, X.-K., Li, S.-Q.: Optimization of Resource Allocation in Construction Using Genetic Algorithms. In: Fourth International Conference on Machine Learning and Cybernetics, Guangzhou, August 18-21, 2005.

14. Loyall, J., Sharma, P., Gillen, M., Schantz, R.: A QoS Management System for Dynamically Interoperating Net-Centric Systems. In: The SPIE Conference on Defense Transformation and Net-Centric Systems, Orlando, FL, April 9-12, 2007.
15. Manghwani, P., Loyall, J., Sharma, P., Gillen, M., Ye, J.: End-to-End Quality of Service Management for Distributed Real-Time Embedded Applications. In: The Thirteenth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2005), Denver, Colorado, April 4-5, 2005.
16. Schantz, R.E., Loyall, J.P., Rodrigues, C., Schmidt, D.C.: Controlling Quality-of-Service in Distributed Real-Time and Embedded Systems via Adaptive Middleware. Software: Practice and Experience 36(11-12), 1189-1208, (2006).
17. Schantz, R.E., Loyall, J.P., Rodrigues, C., Schmidt, D.C., Krishnamurthy, Y., Pyarali, I.: Flexible and Adaptive QoS Control for Distributed Real-Time and Embedded Middleware. In: The ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 2003.
18. Sharma, P., Loyall, J., Schantz, R., Ye, J., Manghwani, P., Gillen, M., Heineman, G.T.: Using Composition of QoS Components to Provide Dynamic, End-To-End QoS in Distributed Embedded Applications - a Middleware Approach. IEEE Internet Computing 10(3), 16-23, (2006).
19. Sharma, P.K., Loyall, J.P., Heineman, G.T., Schantz, R.E., Shapiro, R., Duzan, G.: Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems. In: International Symposium on Distributed Objects and Applications (DOA), Agia Napa, Cyprus, October 25-29, 2004.
20. Sun Microsystems, Java Message Service, Version 1.1, April 12, 2002. http://java.sun.com/products/jms/docs.html.
21. Toyoda, Y.: A Simplified Algorithm for Obtaining Approximate Solution to Zero-One Programming Problems. Management Science 21 (1975).
22. Tukey, J.W.: Exploratory Data Analysis. Addison-Wesley, Reading MA (1977).
23. U.S. Air Force. A Guide for Communities of Interest (COIs), Implementing the DoD Net-Centric Data Strategy and the Air Force Information and Data Management Strategy, Version 1.0, April 2005.
24. US Air Force Air Force Research Laboratory, Apollo v.1.0 User's Guide.
25. W3C, Web Services Architecture, W3C Working Group Note, February 11, 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.
26. W3C, XQuery 1.0: An XML Query Language, W3C Recommendation, 23 January 2007, http://www.w3.org/TR/xquery/.
27. W3C, Extensible Markup Language (XML) 1.0, W3C Recommendation 16 August 2006, http://www.w3.org/TR/xml.
28. Wang, N., Gill, C., Schmidt, D., Gokhale, A., Natarajan, B., Loyall, J., Schantz, R., Rodrigues, C.: QoS-Enabled Middleware. In: Qusay H. Mahmoud (eds.) Middleware for Communications. Wiley (2004).
29. Xu, D., Nahrstedt, K., Wichadakul, D.: QoS and Contention-Aware Multi-Resource Reservation. Cluster Computing 4(2), 95-107 (2001).