Cluster Computing 4, 95–107, 2001

QoS and Contention-Aware Multi-Resource Reservation

DONGYAN XU, KLARA NAHRSTEDT and DUANGDAO WICHADAKUL
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

Abstract. To provide Quality of Service (QoS) guarantee in distributed services, it is necessary to reserve multiple computing and communication resources for each service session. Meanwhile, techniques have been available for the reservation and enforcement of various types of resources. Therefore, there is a need to create an integrated framework for coordinated multi-resource reservation. One challenge in creating such a framework is the complex relation between the end-to-end application-level QoS and the corresponding end-to-end resource requirement. Furthermore, the goals of (1) providing the best end-to-end QoS for each distributed service session and (2) increasing the overall reservation success rate of all service sessions are in conflict with each other. In this paper, we present a QoS and contention-aware framework of end-to-end multi-resource reservation for distributed services. The framework assumes a reservation-enabled environment, where each type of resource can be reserved. The framework consists of (1) a component-based QoS-Resource Model, (2) a runtime system architecture for coordinated reservation, and (3) a runtime algorithm for the computation of end-to-end multi-resource reservation plans. The algorithm provides a solution to alleviating the conflict between the QoS of an individual service session and the success rate of all service sessions. More specifically, for each service session, the algorithm computes an end-to-end reservation plan, such that it guarantees the highest possible end-to-end QoS level under the current end-to-end resource availability, and requires the lowest percentage of bottleneck resource(s) among all feasible reservation plans. Our simulation results show excellent performance of this algorithm.

Keywords: resource reservation, QoS, resource contention, distributed service

AMS subject classification: distributed computing

1. Introduction

Emerging distributed services are expected to provide application-level functionalities with certain Quality of Service (QoS). Examples of distributed services include media data distribution and processing, E-commerce, and virtual scientific laboratory services. Meanwhile, advances in resource reservation and scheduling make it possible to provide end-to-end QoS guarantee in distributed services. For different computing and communication resources, techniques have been available to make reservations and to enforce them during runtime. For example, for CPU capacity, there exist the Dynamic Soft Real Time (DSRT) CPU scheduling framework [1] and the hierarchical CPU scheduler based on Start-time Fair Queuing [2]; for network bandwidth, RSVP protocol [3] and various packet scheduling algorithms [4,5] are responsible for bandwidth reservation and enforcement, respectively; and for disk I/O bandwidth, Cello [6] provides a disk scheduling framework. However, what is missing is a higher-level framework that coordinates the reservations of *multiple* resources required in a distributed service.

One challenge in creating such a multi-resource reservation framework is the complex relation between the end-to-end application-level QoS and the corresponding end-to-end resource requirement. First, each resource contributes to the end-to-end QoS, but not in the simple and linear fashion. Second, a distributed service may achieve multiple levels of end-to-end QoS, which in the general case are represented by multi-dimensional and partially-ordered QoS vectors. Correspondingly, the end-to-end resource requirements to achieve these QoS levels are also represented by partially-ordered resource requirement vectors. Therefore, when determining the amounts of resources to be reserved for a distributed service session, each resource can not be treated individually. Instead, trade-offs must be considered among the requirements of all resources – not only for different levels of end-to-end QoS, but also for the same end-to-end QoS level.

Another challenge in the multi-resource reservation framework is the presence of resource contention: sessions of different distributed services try to reserve from the same pool of resources in the environment, causing some reservations to fail. Therefore, one goal of the framework is to increase the overall success rate of multi-resource reservations in the environment. However, to realize the goal, there exist two difficulties. First, each resource can potentially become the bottleneck resource in a multi-resource reservation, due to the fluctuation of resource availability over time. Any solution that assumes a specific bottleneck resource may not work in all situations. Second, the goal of increasing overall reservation success rate is in conflict with the goal of bringing the highest possible end-to-end QoS to each individual service session.

In this paper, we present a multi-resource reservation framework which deals with these challenges. The framework assumes a fully reservation-enabled environment, where each type of resource can be reserved. The framework is both QoS and resource contention aware. It consists of (1) a component-based QoS-Resource Model to capture the relation between application-level QoS and resource require-

ment, (2) a runtime system architecture to perform multi-resource reservations, and (3) a runtime algorithm to compute an end-to-end multi-resource reservation plan for each distributed service session. In the component-based QoS-Resource Model, a distributed service is modeled as a set of collaborating *service components*. Each service component is associated with a set of *input QoS* levels and *output QoS* levels. The input/output QoS of each participating service component transitively contributes to the end-to-end QoS of the distributed service. The runtime algorithm is based on this model. Finally, in the runtime system architecture, a *QoSProxy* and one or more *Resource Brokers* run for each end host. The QoSProxies execute the runtime algorithm, and dispatch the computed end-to-end reservation plan to the Resource Brokers.

To alleviate the conflict between the end-to-end QoS of individual service sessions and the overall reservation success rate of all service sessions, our runtime algorithm computes a reservation plan for each distributed service session such that: (1) it guarantees the highest possible end-to-end QoS level under the current end-to-end resource availability, and (2) it requires the lowest percentage of bottleneck resource(s) among all feasible reservation plans – the algorithm dynamically identifies the bottleneck resource in each reservation plan. Based on this algorithm, we also propose two heuristics: one to handle more complex dependencies between service components, and one to further improve the overall reservation success rate. Our simulation results show excellent performance of the algorithm: comparing with a contention-unaware algorithm, our algorithm constantly results in higher reservation success rate, while achieving the highest possible level of end-to-end QoS for each successful service session.

The rest of the paper is organized as follows. Section 2 introduces the component-based QoS-Resource Model. Section 3 describes the runtime system architecture for multi-resource reservation. Section 4 presents the runtime algorithm for the computation of end-to-end reservation plans. Section 5 shows the performance of the algorithm based on simulation results. Section 6 compares our framework with related work. Finally, section 7 concludes this paper.

2. A component-based QoS-Resource Model

2.1. Component-based distributed services

With distributed object techniques, a distributed service can be implemented as a set of collaborating *service components*, rather than as a monolithic program. A service component is a functional unit participating in the service delivery. Figure 1 shows an example of a distributed *Video Streaming + Tracking* service. A client requesting this service is allowed to provide images of certain objects, which will be recognized and tracked in the requested video. The tracking of objects is performed by a specialized tracking proxy. Therefore, the video server first streams a video to the tracking proxy, which performs real-time object tracking, and then forwards the video stream with tracking results to the requesting client. In this service, there are three service components: the *VideoSender* component running on the video server, the *ObjectTracker* component running on the tracking proxy, and the *VideoPlayer* component running on each requesting client.

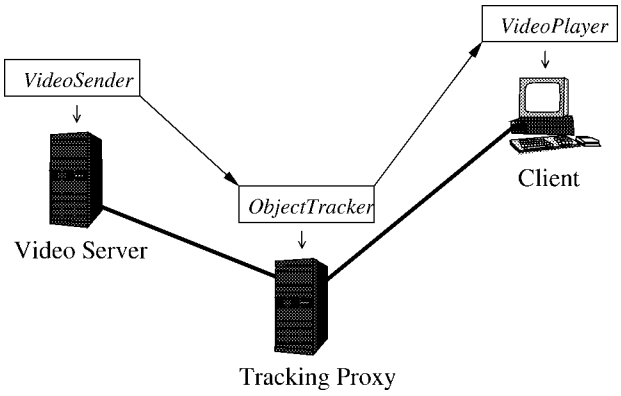


Figure 1. An example of component-based distributed service.

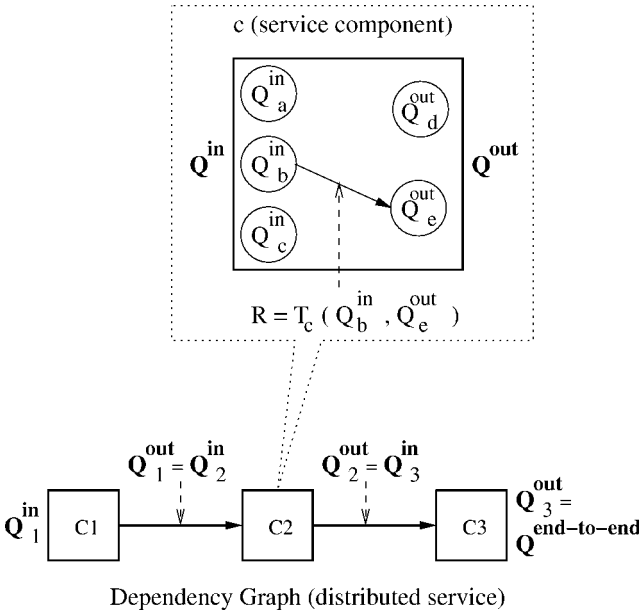


Figure 2. The QoS-Resource Model.

The service component is a natural choice of granularity for modeling the relation between application-level QoS and corresponding resource requirement. Each service component may perform its function at different QoS levels, depending on the availability of the resource(s) required by this service component. Furthermore, for a distributed service, the QoS achieved by each participating service component contributes to the end-to-end QoS in a transitive manner.

2.2. The QoS-Resource Model

The QoS-Resource Model is illustrated in figure 2. We define the model first from the angle of a service component, and then from that of a distributed service.

- *A service component c* is associated with an *input QoS* $\mathbf{Q^{in}}$ and an *output QoS* $\mathbf{Q^{out}}$. Instances of both $\mathbf{Q^{in}}$ and $\mathbf{Q^{out}}$ are *QoS vectors* of multiple application-level QoS parameters. For simplicity (and is often the case in practice), we assume that each parameter takes discrete values. Therefore, both $\mathbf{Q^{in}}$ and $\mathbf{Q^{out}}$ are enumerable. To compare two QoS vectors, they must have the same set of QoS parameters. For two QoS vectors Q_a and Q_b, $Q_a \leqslant Q_b$ holds if and only if for each QoS parameter, the corresponding value of Q_a is not larger than that of Q_b.

  Each service component c is also associated with a *Translation Function* T_c. T_c computes the following: given an input QoS Q^{in}, in order to achieve an output QoS Q^{out}, what is c's resource requirement?, i.e., T_c is defined by $T_c : \mathbf{Q^{in}} \times \mathbf{Q^{out}} \rightarrow \mathbf{R}$. Instances of the resource requirement $\mathbf{R}$ are represented by *resource requirement vectors*. Therefore, given a pair of $(Q^{\text{in}}, Q^{\text{out}})$, we have:

$$R = T_c\big(Q^{\text{in}}, Q^{\text{out}}\big). \tag{1}$$

  The resource requirement vector $R = [r_1, r_2, \ldots, r_M]$, and r_m $(1 \leqslant m \leqslant M)$ is the required amount of the mth resource by service component c. Similar to the QoS vectors, to compare two resource requirement vectors, they must have the same set of resources. For two resource requirement vectors R_a and R_b, $R_a \leqslant R_b$ holds if and only if for each type of resource, the corresponding value of R_a is not larger than that of R_b.
- *A distributed service* is associated with a *Dependency Graph*. The nodes of the Dependency Graph represent the participating service components, while the edges represent the input/output and QoS dependencies between the service components. An edge from service component c_1 to c_2 indicates that the output of c_1 is the input of c_2; and the $\mathbf{Q^{out}}$ of c_1 is equivalent to the $\mathbf{Q^{in}}$ of c_2. Especially, the $\mathbf{Q^{in}}$ of the source node (for example, c_1 in figure 2) represents the original quality of the source data involved in this service; while the $\mathbf{Q^{out}}$ of the sink node (for example, c_3 in figure 2) represents the final end-to-end QoS provided by this distributed service.

For example, for the *Video Streaming + Tracking* service described in section 2.1, its Dependency Graph is a three-node chain: *VideoSender* → *ObjectTracker* → *VideoPlayer*. For service component *VideoSender*, both its $\mathbf{Q^{in}}$ and $\mathbf{Q^{out}}$ are in the form of [*Frame_Rate, Image_Size*], and its resource requirement $\mathbf{R}$ is in the form of [*CPU, Disk_IO_Bandwidth*]. For service component *ObjectTracker*, its $\mathbf{Q^{in}}$ is the $\mathbf{Q^{out}}$ of *VideoSender*, while its $\mathbf{Q^{out}}$ has the form of [*Frame_Rate, Image_Size, Number_of_Trackable_Objects*], and its $\mathbf{R}$ is in the form of [*CPU, Network_bandwidth (between the video server and the tracking proxy)*]. For service component *VideoPlayer*, its $\mathbf{Q^{in}}$ is the $\mathbf{Q^{out}}$ of *ObjectTracker*, while its $\mathbf{Q^{out}}$, which is the end-to-end QoS of this service, has the form of [*Frame_Rate, Image_Size, Number_of_Trackable_Objects, Buffering_Delay*], and its $\mathbf{R}$ is in the form of [*CPU, Network_bandwidth (between the tracking proxy and the client)*].

Our runtime algorithm for the computation of end-to-end multi-resource reservation plans is based on this QoS-Resource Model. However, before presenting the algorithm, we first describe the runtime system architecture in which the algorithm will be run and the reservations will be made.

3. System architecture for multi-resource reservation

The runtime system architecture of our framework is shown in figure 3. We assume a fully reservation-enabled environment. In this environment, service components of different distributed services run on the end hosts. For each end host, a *QoSProxy* and one or more *Resource Brokers* are deployed.

- A Resource Broker is responsible for making and enforcing the reservations for a certain resource. The resource can be a local resource, such as CPU, memory, or disk I/O bandwidth, or it can be an end-to-end network resource between this host and a remote host. In the latter case, the end-to-end network resource is managed in a two-level manner. At the higher level, the Resource Broker on one of the end hosts[1] treats the network links between the two end hosts as one resource. At the lower level, the RSVP-enabled *bandwidth broker* on each router treats each network link as a separate resource. As we show later, this two-level approach to end-to-end network resource reservation is compatible with our runtime algorithm. The basic operations of a Resource Broker include: (1) reporting current availability of the corresponding resource, (2) making and enforcing reservations for this resource, and (3) terminating or canceling reservations for this resource.

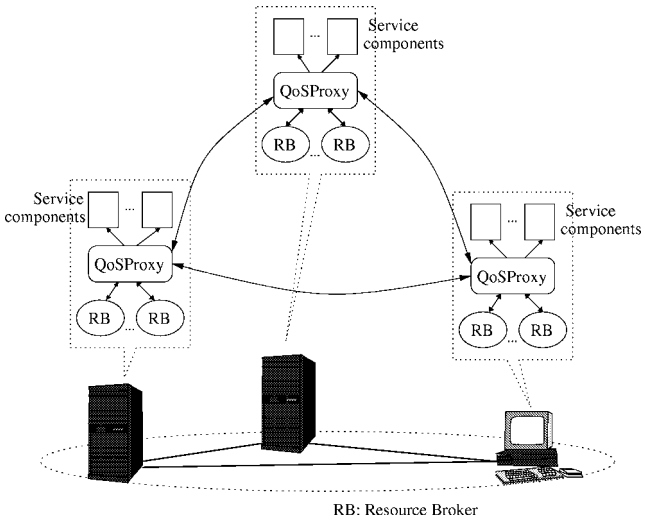


Figure 3. Runtime system architecture for multi-resource reservation.

[1] To be compatible with RSVP, we assume that the network Resource Broker on the *receiver* side initiates an end-to-end network bandwidth reservation.

- The QoSProxy is the coordinator of multi-resource reservation activities. More specifically, for a distributed service session, QoSProxies of the hosts involved in the session coordinate to compute the end-to-end multi-resource reservation plan, according to our runtime algorithm (to be described in section 4). Other basic operations of a QoSProxy include: (1) collecting resource availability information from local Resource Brokers, (2) after a multi-resource reservation plan is computed, dispatching the plan to its local Resource Brokers, and (3) starting the local service component(s) participating in the service session, when the end-to-end multi-resource reservation is completed.

The system architecture is service-independent. However, to plan and to perform multi-resource reservations for a distributed service, the definition of the QoS-Resource Model for this service has to be understood by the architecture. In our framework, the QoS-Resource Model definition is stored in the QoSProxies of one or more end hosts involved in the service. In a centralized approach, the definition will be stored and accessed by the QoSProxy of the main server for the service, while in a distributed approach, the $\mathbf{Q^{in}}$ and $\mathbf{Q^{out}}$ levels and the Translation Function of each service component will be stored and accessed by the QoSProxy of the host where the service component runs. In either approach, a Translation Function is provided by the developer of the corresponding service component as a "plug-in" function, which will be called by a QoSProxy when computing the end-to-end multi-resource reservation plan. We assume the centralized approach for the rest of the paper.

4. Algorithm for computing multi-resource reservation plans

After introducing the model and architecture for multi-resource reservation, we now present the runtime algorithm for computing end-to-end multi-resource reservation plans for distributed service sessions. The algorithm is the main contribution in our framework.

4.1. Basic algorithm

The key ideas in the algorithm are *QoS-awareness* and *contention-awareness*:

- *QoS-awareness*. As defined in the QoS-Resource Model, each service component may accept multiple levels of $\mathbf{Q^{in}}$, and achieve multiple levels of $\mathbf{Q^{out}}$. Each pair of (Q^{in}, Q^{out}) is associated with a certain resource requirement vector. When the algorithm computes an end-to-end resource reservation plan, it will select appropriate $\mathbf{Q^{in}}$ and $\mathbf{Q^{out}}$ levels for each service component, so that they will lead to the highest possible end-to-end QoS level. This is under the constraint that the resource requirement of each service component is satisfied by the current end-to-end resource availability.
- *Contention-awareness*. A resource may be requested by multiple service sessions on a competitive basis. Therefore, resource contention exists. The degree of resource contention varies from resource to resource and from time to time. Moreover, the failure to reserve one resource leads to the reservation failure for the whole distributed service session. To increase the overall reservation success rate of all service sessions, our runtime algorithm dynamically determines an end-to-end reservation plan for each service session. This plan is selected from multiple feasible reservation plans, such that it will reserve the lowest percentage of bottleneck resource(s). Therefore, each computed end-to-end reservation plan is "disciplined" in the aspect of its bottleneck resource consumption. Note that the bottleneck resource in each reservation plan may be different and even change over time. It will be dynamically identified by our algorithm.

There are two main steps in the runtime algorithm: for each distributed service session, the algorithm (1) constructs a *QoS-Resource Graph (QRG)* based on the QoS-Resource Model of this service, and (2) finds the end-to-end reservation plan based on the QRG. The two steps will be described in the next two subsections.

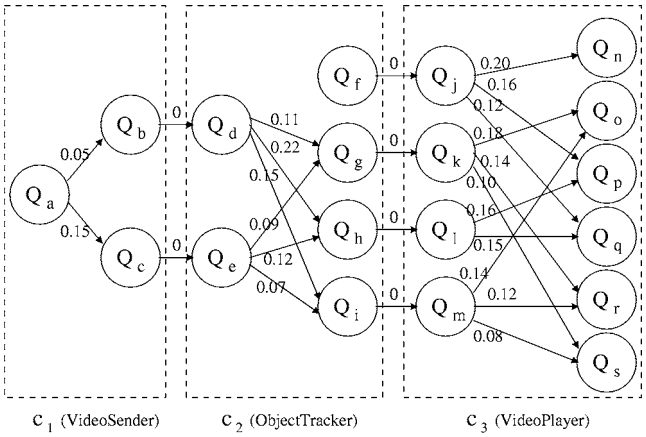
4.1.1. Constructing the QoS-Resource Graph

For a distributed service session, the QoS-Resource Graph (QRG) represents a "snap-shot" of the end-to-end resource requirement and availability, as well as the achievable levels of $\mathbf{Q^{in}}$ and $\mathbf{Q^{out}}$ for each service component. Figure 4 shows an example QRG constructed for a *Video Streaming* + *Tracking* service session. The dotted rectangles in the QRG represent the corresponding service components in the Dependency Graph of this service. A QRG is defined as follows:

(1) *QRG nodes*: For each participating service component c, the possible levels of its $\mathbf{Q^{in}}$ and $\mathbf{Q^{out}}$ are represented as nodes of the QRG. The source node of the QRG (for example, Q_a in figure 4) represents the original quality of the source data associated with this service session. For the service component whose $\mathbf{Q^{out}}$ is the end-to-end QoS (for example, service component *VideoPlayer* in figure 4), its $\mathbf{Q^{out}}$ nodes are sink nodes of the QRG representing the end-to-end QoS levels. Although QoS levels of a service component are partially ordered in general, we assume that the end-to-end QoS levels can be ranked in a *linear* order, based on a user's preference or some other subjective criteria. For example, when two end-to-end QoS levels are not comparable, the user who requests the service session can arbitrate that the QoS level with a smaller *delay* parameter is better than the one with a larger delay.

(2) *QRG edges*: Edges of the QRG are in two categories:

- Edges from $\mathbf{Q^{in}}$ nodes to $\mathbf{Q^{out}}$ nodes of the same service component. For a service component c, an edge from a $\mathbf{Q^{in}}$ node Q_x to a $\mathbf{Q^{out}}$ node Q_y exists, if and



Q_a = [24 frames / second , 320 x 240]
$Q_{b,d}$ = [24 frames / second , 320 x 240]
$Q_{c,e}$ = [24 frames / second , 160 x 120]
$Q_{f,j}$ = [24 frames / second , 320 x 240, 3 trackable objs]
$Q_{g,k}$ = [24 frames / second , 320 x 240, 1 trackable obj]
$Q_{h,l}$ = [24 frames / second , 160 x 120, 3 trackable objs]
$Q_{i,m}$ = [24 frames / second , 160 x 120, 1 trackable obj]
Q_n = [24 frames / second , 320 x 240, 3 trackable objs, 2.0 seconds]
Q_o = [24 frames / second , 320 x 240, 1 trackable obj, 2.0 seconds]
Q_p = [16 frames / second , 320 x 240, 3 trackable objs, 5.0 seconds]
Q_q = [24 frames / second , 160 x 120, 3 trackable objs, 2.0 seconds]
Q_r = [16 frames / second , 160 x 120, 1 trackable obj, 2.0 seconds]
Q_s = [24 frames / second , 160 x 120, 1 trackable obj, 5.0 seconds]

Figure 4. An example QRG (we assume that both *ObjectTracker* and *VideoPlayer* have a hypothetical image intrapolation capability to *scale up* the size of video images, at the cost of higher CPU requirement).

only if the resource requirement vector R^{req}, calculated as $R^{\text{req}} = T_c(Q_x, Q_y)$, can be satisfied by the current availability of the corresponding resources.

- Edges from $\mathbf{Q^{out}}$ nodes of a service component to $\mathbf{Q^{in}}$ nodes of the downstream service component in the Dependency Graph. Each edge simply indicates the equivalence of the two nodes it connects.

(3) *Weights of QRG edges*: A *weight* is computed for each edge in the QRG.

- For an edge belonging to the first category, the weight indicates the *degree of contention* in reserving the resources associated with this edge. For an edge from a $\mathbf{Q^{in}}$ node Q_x to a $\mathbf{Q^{out}}$ node Q_y of service component c, let $R^{\text{req}} = [r_1^{\text{req}}, r_2^{\text{req}}, \ldots, r_M^{\text{req}}]$ be the corresponding resource requirement vector. On the other hand, let $R^{\text{avail}} = [r_1^{\text{avail}}, r_2^{\text{avail}}, \ldots, r_M^{\text{avail}}]$ be the current resource availability vector. R^{req} is computed by calling $T_c(Q_x, Q_y)$, while R^{avail} is obtained by querying the Resource Brokers of these resources. For a network Resource Broker, r^{avail} is the minimum of the link bandwidth availabilities reported by the lower-level RSVP-enabled bandwidth brokers.

  We first define a *contention index* ψ_i for the ith resource to evaluate how "competitive" it is to reserve r_i^{req} amount of resource, under the availability of r_i^{avail}. We adopt a simple definition of ψ as follows:

$$\psi_i = \frac{r_i^{\text{req}}}{r_i^{\text{avail}}}, \quad r_i^{\text{req}} \leqslant r_i^{\text{avail}}. \tag{2}$$

  Intuitively, the larger the percentage of a resource we try to reserve under its current availability, the lower the probability that the reservation will succeed.[2] Then, we further define the weight Ψ of the edge as

$$\Psi = \max_{i=1}^{M} \psi_i = \max_{i=1}^{M} \frac{r_i^{\text{req}}}{r_i^{\text{avail}}}. \tag{3}$$

- For an edge belonging to the second category, the weight is zero, because the edge only represents the equivalence of the two nodes it connects.

4.1.2. Selecting an end-to-end reservation plan

After constructing the QRG, the algorithm will select an end-to-end multi-resource reservation plan for the service session.

The selection is based on two key observations. In the QRG, each edge with a non-zero weight exists if and only if the corresponding $R^{\text{req}} \leqslant R^{\text{avail}}$, i.e., the reservation of resources according to R^{req} is feasible. Therefore, we have the first key observation: *every path from the source node to one of the sink nodes represents a feasible end-to-end multi-resource reservation plan, and the sink node represents the end-to-end QoS level that can be achieved by this reservation plan*. In particular, the highest achievable end-to-end QoS under the current resource availability is represented by such a sink node: it has the highest QoS ranking among all the sink nodes which are "reachable" via a path from the source node. For example, in figure 4, if we assume a linear order of $Q_n > Q_o > Q_p > Q_q > Q_s > Q_r$, then Q_o is the highest achievable end-to-end QoS.

However, to reach the highest achievable end-to-end QoS from the source node, there exist multiple paths, i.e., there exist more than one feasible end-to-end reservation plan to achieve the highest possible end-to-end QoS. To make a choice, we first define Ψ_P for each path P as

$$\Psi_P = \max_{(\text{each edge } e \text{ on path } P)} \Psi_e. \tag{4}$$

It is easy to see that Ψ_P represents the highest contention index of any resource on path P, i.e., the contention index of the *bottleneck resource(s)* on P. Note that on different paths, the bottleneck resource may be different. Then, to minimize resource contention, our algorithm should select a path, such that its Ψ_P is the smallest among all the paths leading to the sink node representing the highest possible end-to-end QoS. In other words, our algorithm should select an end-to-end reservation plan, such that it requires the lowest percentage

[2] In fact, there are other definitions of ψ which also exhibit this property. Fortunately, it is straightforward for our algorithm to adopt a different ψ definition in the future.

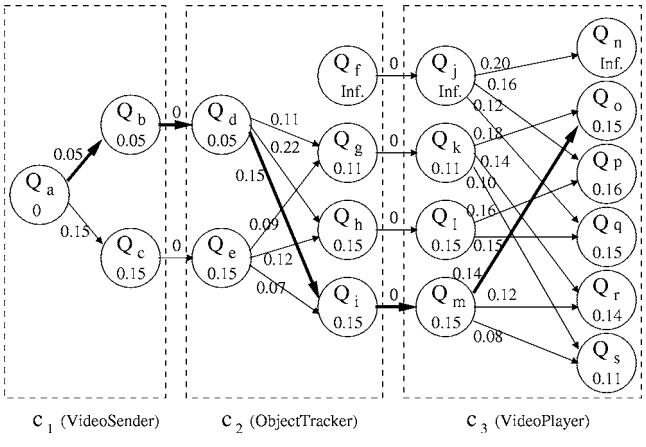


Figure 5. The shortest path from Q_a to Q_o, which represents the end-to-end reservation plan computed by the algorithm.

of bottleneck resource(s) among all feasible plans to achieve the highest possible end-to-end QoS. To select such a path, we have the second key observation: *the path to be selected is the "shortest" path from the source node of the QRG to the sink node representing the highest possible end-to-end QoS, with operator "+" re-defined as "*max*" during the shortest path computation*. Figure 5 shows such a shortest path (by the thicker edges) in the example QRG.

The shortest path can be computed by running Dijkstra's algorithm on the QRG. In figure 5, the value associated with each node is generated during the execution of Dijkstra's algorithm. Especially, for a sink node, the value is equal to the contention index ψ of the bottleneck resource on the shortest path from the source node to this node. With "+" re-defined as "max", we also add the following tie-breaking rule to Dijkstra's algorithm: when there are two $\mathbf{Q^{in}}$ nodes as candidates for the predecessor of a $\mathbf{Q^{out}}$ node [7], if the values in the $\mathbf{Q^{in}}$ nodes and the weights of the two edges have the relation $\max(a, b) = \max(a, c) = a$, we choose the predecessor according to $\min(b, c)$. For example, in figure 5, we choose Q_d instead of Q_e as the predecessor of Q_i.

4.2. Algorithm complexity and execution overhead

The computation complexity of this runtime algorithm is $O(K\mathbf{Q}^2)$. K is the number of participating service components in a distributed service session. $\mathbf{Q}$ is the maximum number of $\mathbf{Q^{out}}$ nodes (QoS levels) of a service component. For example, in figure 5, $K = 3$ and $\mathbf{Q} = 6$ – the number of $\mathbf{Q^{out}}$ nodes of service component c_3. In practice, K and $\mathbf{Q}$ usually have fairly small values, for example, a service having fewer than ten service components, and a service component having tens of $\mathbf{Q^{out}}$ nodes. Therefore, our algorithm is efficient for runtime execution.

In section 3, we assume that the QoS-Resource Model definition of a distributed service is stored in the QoSProxy of the main server of this service, which we will call it the *main QoSProxy*. Therefore, there are three phases to run our algorithm: first, QoSProxies of all participating end hosts report current resource availability to the main QoSProxy; second, the main QoSProxy executes the algorithm locally; and third, the main QoSProxy dispatches different segments of the computed end-to-end multi-resource reservation plan to the participating QoSProxies, respectively. The overhead will involve one message passing round-trip between each participating QoSProxy and the main QoSProxy, and the local execution of the algorithm at the main QoSProxy.

4.3. Extensions to the algorithm

In this section, we propose extensions to the basic algorithm to further improve its performance and applicability.

4.3.1. Trading off end-to-end QoS for overall success rate

The first extension is to further improve the overall reservation success rate among all service sessions. The enhancement is based on the following observations: (1) the basic algorithm is "greedy" in the sense that it always tries to bring the highest possible end-to-end QoS to each service session, and then makes a contention-aware selection among those reservation plans that lead to the highest possible end-to-end QoS. However, if the algorithm can trade off the end-to-end QoS of each service session, a higher overall reservation success rate may be achieved; (2) meanwhile, the basic algorithm uses a snapshot of current resource availability to construct a QRG. However, if the algorithm is also aware of the *trends* in resource availability changes, the overall reservation success rate may be improved.

Based on these observations, we propose a trade-off policy between end-to-end QoS and overall success rate. This policy requires that each Resource Broker also generates and reports an *Availability Change Index* α to the QoSProxy. The Availability Change Index is generated as follows: assume that each Resource Broker keeps an average $r_{\text{avg}}^{\text{avail}}$ of resource availability values reported during the past T amount of time (T may vary for different Resource Brokers). When the QoSProxy queries the Resource Broker, the Resource Broker will report both r^{avail} – the current availability, and α, which is computed as

$$\alpha = \frac{r^{\text{avail}}}{r_{\text{avg}}^{\text{avail}}}. \tag{5}$$

$r_{\text{avg}}^{\text{avail}}$ will be updated after each report. α reflects the trend in this resource's availability: when $\alpha \geqslant 1.0$, the trend is "up" or "unchanged"; when $\alpha < 1.0$, the trend is "down".

During the execution of the algorithm, α of each resource will be attached to ψ of this resource. Especially, after running Dijkstra's Algorithm, each sink node will be associated with both ψ and α of the bottleneck resource on the shortest path from the source node to this node. Our trade-off policy will determine the end-to-end QoS level for this service session as follows: let s_0 be the sink node representing the highest possible end-to-end QoS level, and ψ_{s_0} and α_{s_0} be the ψ and α values associated with s_0, respectively,

- if $\alpha_{s_0} \geqslant 1.0$, then s_0 will still be the end-to-end QoS for this service session as in the basic algorithm;

- if $\alpha_{s_0} < 1.0$, then the end-to-end QoS will be the highest ranked sink node s that satisfies: $\psi_s \leqslant \alpha_{s_0}\psi_{s_0}$.

The intuition behind this heuristics is: when the availability of the bottleneck resource tends to go down ($\alpha_{s_0} < 1.0$), the algorithm chooses a lower end-to-end QoS level, such that the bottleneck resource in the corresponding end-to-end reservation plan has a lower contention index (by a ratio of $1 - \alpha_{s_0}$). In section 5, our experimental results will show that this trade-off heuristics constantly achieves higher overall reservation success rate than the basic algorithm.

4.3.2. Supporting DAG dependency graphs

The second extension is to improve the applicability of the basic algorithm. Before this section, there has been an implicit assumption about a distributed service: its Dependency Graph is a *chain* of service components. The assumption limits the algorithm's applicability. In this section, we extend both the QoS-Resource Model and the basic algorithm, so that they are applicable to a distributed service whose Dependency Graph is a Directed Acyclic Graph (DAG).

(1) *Extension to the QoS-Resource Model.* For a distributed service with a DAG Dependency Graph (as shown in figure 6), we add the following definitions about the QoS dependencies among service components:

- If a service component c has more than one adjacent service component, we call c a *fan-out* service component – for example, c_2 in figure 6. Its $\mathbf{Q^{out}}$ is equivalent to the $\mathbf{Q^{in}}$ of each service component adjacent to it.
- If a service component c is adjacent to more than one service component, we call c a *fan-in* service component – for example, c_5 in figure 6. Its $\mathbf{Q^{in}}$ is defined as the *concatenation* of $\mathbf{Q^{out}}$ of each service component it is adjacent to.

(2) *Extension to the basic algorithm.* The algorithm still first constructs a QRG for each distributed service session. An example QRG generated from the DAG Dependency Graph in figure 6 is shown in figure 7. A feasible end-to-end multi-resource reservation plan is now represented by an *embedded graph* G in the QRG such that: (1) in G, there is only one $\mathbf{Q^{in}}$ node and one $\mathbf{Q^{out}}$ node from each service component, and an edge exists between the two nodes in the QRG; (2) in G, the $\mathbf{Q^{out}}$ node of one service component is equivalent to (or contributes to, if its adjacent service component is a fan-in component) the $\mathbf{Q^{in}}$ node of its adjacent service component(s). Therefore, the goal of our algorithm is to find such a G that: (1) the sink node in G represents the highest reachable end-to-end QoS level in the QRG; and (2) the value of Ψ_G is the smallest among all embedded graphs representing the feasible end-to-end reservation plans – Ψ_G is defined as:

$$\Psi_G = \max_{(\text{each edge } e \text{ in } G)} \Psi_e. \qquad (6)$$

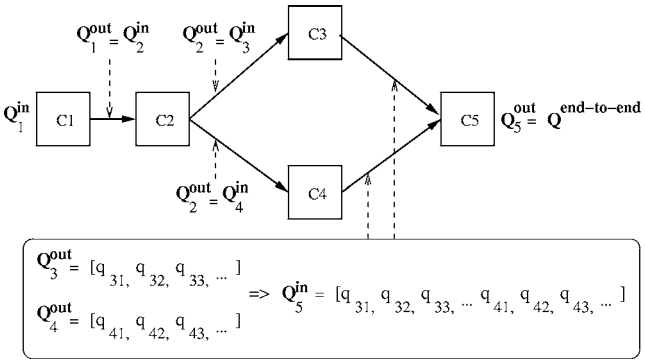


Figure 6. An example of DAG dependency graph and the extension to the QoS-resource model.

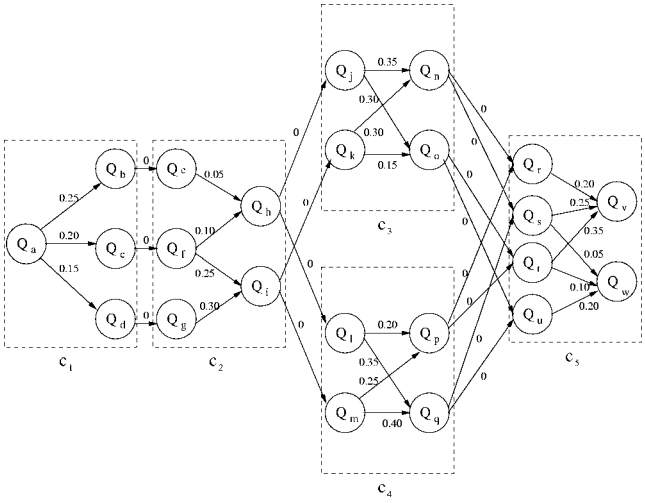


Figure 7. An example QRG based on a DAG dependency graph.

To approach this goal, we provide an efficient heuristics which will result in high end-to-end QoS level and low Ψ_G value. Our heuristics is based on the following two-pass procedure on the QRG:

- *Pass I* on the QRG is similar to Dijkstra's Algorithm, in order to "probe" the shortest paths from the source node to the sink nodes of the QRG. However, a key difference from Dijkstra's Algorithm is: when computing the value associated with a $\mathbf{Q^{in}}$ node of a fan-in service component (for example, node Q_r of c_5 in figure 8), we set the value to be the *maximum* of those associated with the $\mathbf{Q^{out}}$ nodes of the service components it is adjacent to (for example, node Q_n of c_3 and node Q_p of c_4, whose concatenation forms Q_r).
- *Pass II* on the QRG is in the backward direction of pass I. Starting from the reachable sink node with the highest QoS ranking (for example, Q_v), we backtrack the edges toward the source node, according to the result of pass I. This is to determine the embedded graph that represents the resultant end-to-end reservation plan. However, the backtracking may encounter the following problem: at a fan-out service component, the selected edges do not converge at the *same* $\mathbf{Q^{out}}$ node. For example, in figure 8,

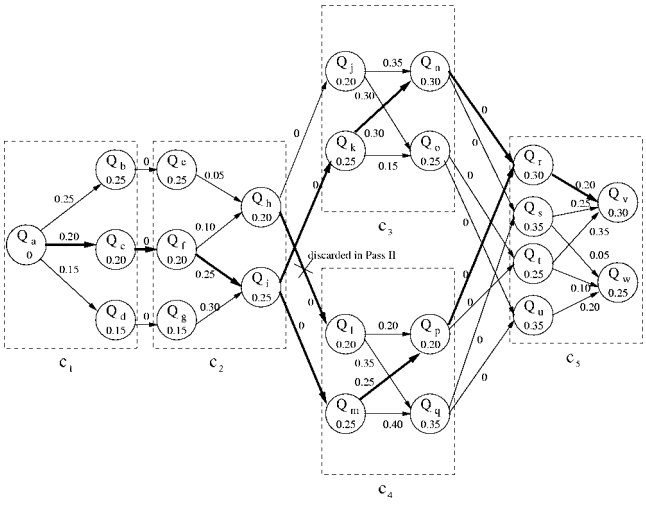


Figure 8. Running the heuristics: the embedded graph represents the resultant end-to-end reservation plan.

the selected edges (the thicker ones in the figure) lead to different $\mathbf{Q^{out}}$ nodes Q_h and Q_i. We use the following policy to resolve this non-convergence *locally*: for the service components adjacent to the fan-out component (for example, c_3 and c_4), fix their $\mathbf{Q^{out}}$ nodes that have been backtracked (for example, nodes Q_n and Q_p); then select such a $\mathbf{Q^{out}}$ node of the fan-out service component: it incurs the lowest degree of resource contention to reach the fixed $\mathbf{Q^{out}}$ nodes of the adjacent service components. For example, in figure 8, Q_i will be selected instead of Q_h, because for Q_i to reach Q_n and Q_p, the highest Ψ_e is 0.30; while for Q_h to reach Q_n and Q_p, the highest Ψ_e is 0.35. The computed end-to-end reservation plan is illustrated in figure 8 by the thicker edges.

Limitations of this heuristics include: (1) for a sink node of the QRG reachable after pass I, the heuristics may not be able to find a feasible end-to-end reservation plan in pass II; (2) due to the local (instead of global) nature of the non-convergence resolution in pass II, the computed end-to-end reservation plan may not have the lowest bottleneck resource contention index among all feasible reservation plans.

5. Performance study

In this section, we study the performance of our multi-resource reservation framework. We simulate a distributed reservation-enabled environment, with multiple distributed services deployed and multiple clients requesting these services. In particular, for runtime computation of end-to-end multi-resource reservation plans, we compare our basic algorithm (referred to as *basic*), our algorithm with the trade-off policy presented in section 4.3.1 (referred to as *trade-off*), and a contention-*unaware* algorithm (referred to as *random*), which *randomly* selects a feasible end-to-end reservation path leading to the highest possible end-to-end QoS level, instead of finding the "shortest-path" in the QRG. The key performance metrics in our simulations are: (1) the overall reservation success rate of all service sessions, and (2) the average end-to-end QoS level provided to successful service sessions.

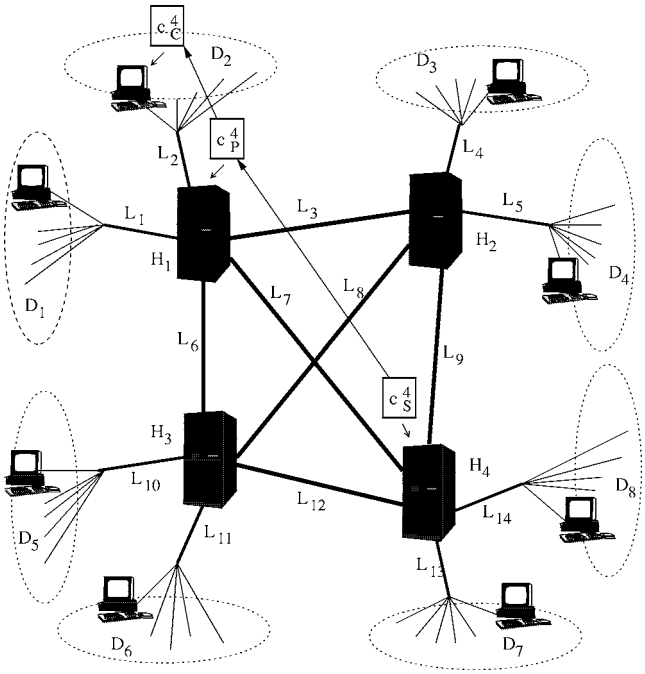
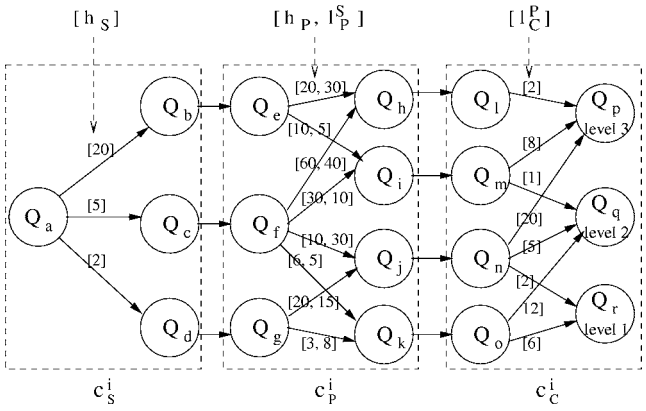


Figure 9. The simulated reservation-based distributed environment.

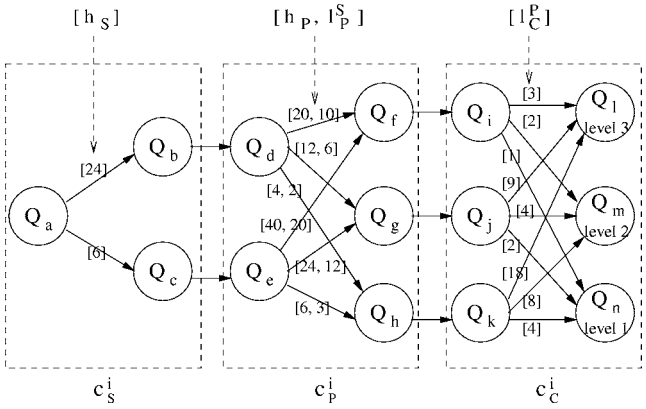
5.1. Simulation setup

The simulated environment is shown in figure 9. There are four high performance computers H_1-H_4, as well as a number of client machines in eight different domains D_1-D_8. These hosts are connected by high speed network links L_1-L_{14}. Four different distributed services S_1-S_4 are deployed in the environment. The QoS-Resource Model for each service is defined as follows:

- For service S_i ($1 \leqslant i \leqslant 4$), the main server is H_i. The Dependency Graph of S_i involves a chain of three service components $c_S^i \rightarrow c_P^i \rightarrow c_C^i$: c_S^i is the server-side service component running on host H_i; c_C^i is the client-side service component running on the requesting client; and c_P^i is a proxy service component running on host H_j ($1 \leqslant j \leqslant 4$ and $j \neq i$) – depending on which domain the client is from. For example, in figure 9, if a client in domain D_2 requests service S_4, then the service session will involve service components c_S^4 on H_4, c_P^4 on H_1, and c_C^4 on the client itself.
- For each of service components c_S^i, c_P^i, and c_C^i, figure 10 shows the $\mathbf{Q^{in}}$ and $\mathbf{Q^{out}}$ levels and the corresponding resource requirements: (a) is for services S_1 and S_4; while (b) is for services S_2 and S_3. The service components require four end-to-end resources: c_S^i requires resource h_S, which is a local resource of the server; c_P^i requires resources h_P and l_P^S, which represent the local resource of the proxy (h_S and h_P assumed to be of the same



(a)

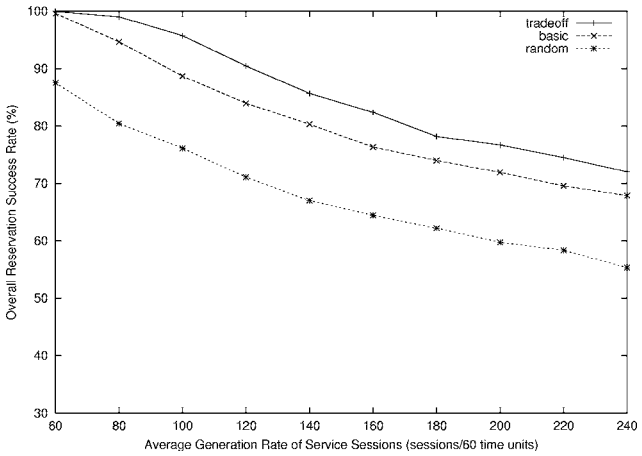


(b)

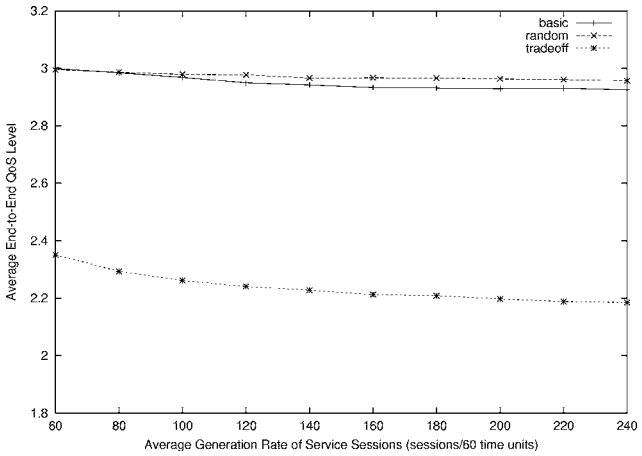
Figure 10. QoS levels and corresponding resource requirements in different services: (a) for services S_1 and S_4; (b) for services S_2 and S_3.

type), and the network link between the server and the proxy, respectively; and c_C^i requires resource l_C^P, which is the network link between the proxy and the client. We also assume that the end-to-end QoS levels are ranked as $Q_p > Q_q > Q_r$ in figure 10(a), and $Q_l > Q_m > Q_n$ in figure 10(b). In the same order, we denote the QoS levels as *level 3, level 2*, and *level 1*, respectively.

In our simulation, service requests from different clients generate service sessions. More specifically, a service session is generated by a client from a randomly selected domain among $D_1 - D_8$. The type of service is selected among the four services except $S_{\lceil i/2 \rceil}$ (i is the index of the domain where the client is from). The service sessions are highly *heterogeneous* in their resource requirement and duration. For resource requirement heterogeneity, figure 10 shows the "base" resource requirement for *normal* service sessions. However, there are also "fat" service sessions whose resource requirement is N times the values in figure 10. N is either 2 or 10; and the ratio between normal service sessions and "fat" service sessions is 1 : 2. For service duration heterogeneity, the duration of a service session is randomly distributed in the wide range between 20 and 600 time units. Those longer than 60 time units are called "long" service sessions, while the rest are called "short" service sessions.



(a)



(b)

Figure 11. Overall reservation success rate and average end-to-end QoS level: (a) overall reservation success rate and (b) average end-to-end QoS level.

The ratio between "long" service sessions and "short" service sessions is 1 : 2. The service sessions are generated according to a Poisson process. We perform multiple runs of the simulation, each run with a different average generation rates – from 60 *sessions per 60 TUs* (Time Units) to 240 *sessions per 60 TUs*. Each run takes a total of 10800 time units.

The initial total amount of each resource is randomly set between 1000 and 4000 units. During each run, we also dynamically change the probability that each service is requested. Therefore, the overall demand for each individual resource changes over time. We create these conditions to test our algorithm's adaptivity in dynamically identifying bottleneck resource(s) and selecting different end-to-end reservation plans.

5.2. Simulation results

5.2.1. Overall reservation success rate and average end-to-end QoS level

Figure 11 shows the overall reservation success rate (a) and the average end-to-end QoS level provided to the *success-*

Table 1
Selected reservation paths and their percentages in QRGs generated from figure 10(a), by *basic* and *tradeoff*, respectively.

Selected path (see figure 10(a))	*basic*	*tradeoff*
$Q_a - Q_b - Q_e - Q_h - Q_l - Q_p$	1.1%	0.1%
$Q_a - Q_c - Q_f - Q_h - Q_l - Q_p$	0.3%	0.0%
$Q_a - Q_b - Q_e - Q_i - Q_m - Q_p$	21.9%	4.4%
$Q_a - Q_c - Q_f - Q_i - Q_m - Q_p$	55.8%	22.2%
$Q_a - Q_c - Q_f - Q_j - Q_n - Q_p$	0.4%	0.0%
$Q_a - Q_d - Q_g - Q_j - Q_n - Q_p$	18.0%	2.3%
$Q_a - Q_b - Q_e - Q_i - Q_m - Q_q$	0.1%	2.6%
$Q_a - Q_c - Q_f - Q_i - Q_m - Q_q$	0.1%	11.0%
$Q_a - Q_d - Q_g - Q_j - Q_n - Q_q$	0.1%	6.5%
$Q_a - Q_c - Q_f - Q_k - Q_o - Q_q$	1.6%	41.8%
$Q_a - Q_d - Q_g - Q_k - Q_o - Q_q$	0.6%	9.0%

Table 2
Selected reservation paths and their percentages in QRGs generated from figure 10(b), by *basic* and *tradeoff*, respectively.

Selected path (see figure 10(b))	*basic*	*tradeoff*
$Q_a - Q_b - Q_d - Q_f - Q_i - Q_l$	1.4%	0.4%
$Q_a - Q_c - Q_e - Q_f - Q_i - Q_l$	11.2%	2.0%
$Q_a - Q_b - Q_d - Q_g - Q_j - Q_l$	1.8%	0.5%
$Q_a - Q_c - Q_e - Q_g - Q_j - Q_l$	17.4%	9.6%
$Q_a - Q_b - Q_d - Q_h - Q_k - Q_l$	6.0%	0.4%
$Q_a - Q_c - Q_e - Q_h - Q_k - Q_l$	61.8%	15.7%
$Q_a - Q_b - Q_d - Q_f - Q_i - Q_m$	0.0%	0.0%
$Q_a - Q_c - Q_e - Q_f - Q_i - Q_m$	0.0%	4.8%
$Q_a - Q_b - Q_d - Q_g - Q_j - Q_m$	0.0%	0.2%
$Q_a - Q_c - Q_e - Q_g - Q_j - Q_m$	0.0%	7.6%
$Q_a - Q_b - Q_d - Q_h - Q_k - Q_m$	0.0%	0.9%
$Q_a - Q_c - Q_e - Q_h - Q_k - Q_m$	0.1%	57.8%

ful service sessions (b). The results are obtained under different average generation rates of service sessions, and by using algorithms *basic*, *tradeoff*,[3] and *random*, respectively. In reservation success rate, *tradeoff* outperforms *basic*, which in turn outperforms the contention-unaware *random*. However, in average end-to-end QoS level, *tradeoff* results in lower average QoS levels, due to the "QoS-success rate" tradeoff. On the other hand, both *basic* and *random* achieve end-to-end QoS levels very close to the highest level (level 3), due to their "greedy" nature on behalf of each individual service session.

5.2.2. Selection of end-to-end reservation paths

We record the selected end-to-end reservation path in the QRG for each service session. Our results show that in the QRGs generated from either figures 10(a) or (b), the paths selected by *basic* and *tradeoff* have covered most of the existing paths in that figure. We also confirm via our results that *every* resource in the environment becomes the bottleneck resource on a path for at least once during the simulation. Tables 1 and 2 list the selected reservation paths and their percentages of occurrence, during the simulation

[3] For *tradeoff*, each Resource Broker keeps an average $r_{\text{avg}}^{\text{avail}}$ of r^{avail} values reported during the past 3 time units.

Table 3
Reservation success rates/average QoS levels in each class, achieved by *basic*.

Class/gen. rate	60 ssn.s/ 60 TUs	100 ssn.s/ 60 TUs	180 ssn.s/ 60 TUs
Norm.-short	99.9%/3.00	97.3%/2.99	92.0%/2.98
Norm.-long	99.9%/3.00	97.4%/2.99	92.2%/2.98
Fat-short	99.0%/2.99	73.2%/2.88	40.7%/2.67
Fat-long	98.6%/2.99	72.8%/2.86	38.9%/2.66

Table 4
Reservation success rates/average QoS levels in each class, achieved by *tradeoff*.

Class/gen. rate	60 ssn.s/ 60 TUs	100 ssn.s/ 60 TUs	180 ssn.s/ 60 TUs
Norm.-short	100%/2.35	98.3%/2.27	94.4%/2.21
Norm.-long	100%/2.36	98.4%/2.27	94.1%/2.22
Fat-short	99.9%/2.36	87.6%/2.26	49.6%/2.18
Fat-long	99.7%/2.31	86.8%/2.27	49.1%/2.17

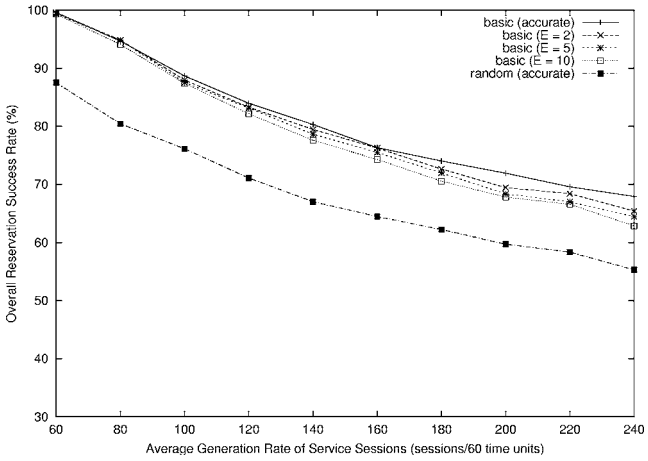
run with average session generation rate of 80 *sessions per 60 TUs*. The results demonstrate the adaptivity of *basic* and *tradeoff* in dynamically identifying bottleneck resource(s) and selecting different end-to-end reservation plans.

5.2.3. Impact of service session heterogeneity

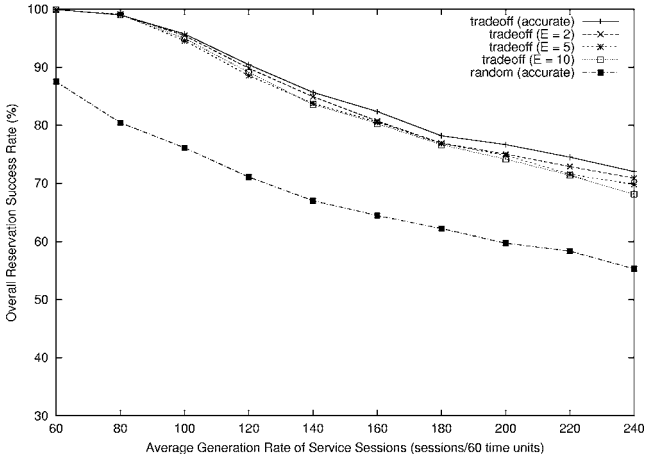
Since the service sessions are heterogeneous in resource requirement and duration, we categorize all service sessions into four classes: *normal and short* sessions, *normal and long* sessions, *fat and short* sessions, and *fat and long* sessions. "Normal"/"fat" as well as "short"/"long" are defined in section 5.1. Tables 3 and 4 show the reservation success rates and average end-to-end QoS levels in each service session class, under different average generation rates of (all) service sessions and by *basic* and *tradeoff*, respectively. We observe that both *fat and short* and *fat and long* classes result in lower reservation success rates and lower average end-to-end QoS levels than the other two classes. However, there is no significant difference between *fat and short* and *fat and long*, as well as between *normal and short* and *normal and long*. The results suggest that the resource requirement heterogeneity has more significant impact than the duration heterogeneity on both key performance metrics.

5.2.4. Impact of resource availability observation inaccuracy

So far in our simulation, we assume and implement the framework as follows: for each service session, the computation of end-to-end reservation plan and the actual reservation take place in an atomic manner. Therefore, the resource availability observation is always consistent and up-to-date, and there will be no change until after the actual reservation. However, this assumption of accurate observation may not hold in real life, due to the concurrency among multiple service sessions as well as the varying latency in the collection of multi-resource availability. To show its impact on the performance of algorithms *basic* and *tradeoff*,

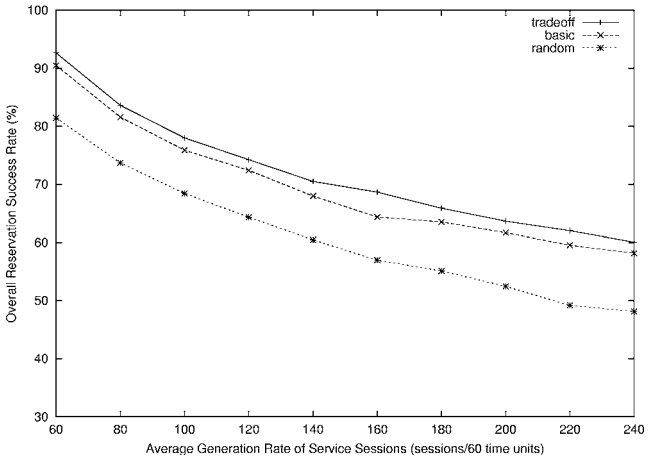


(a)

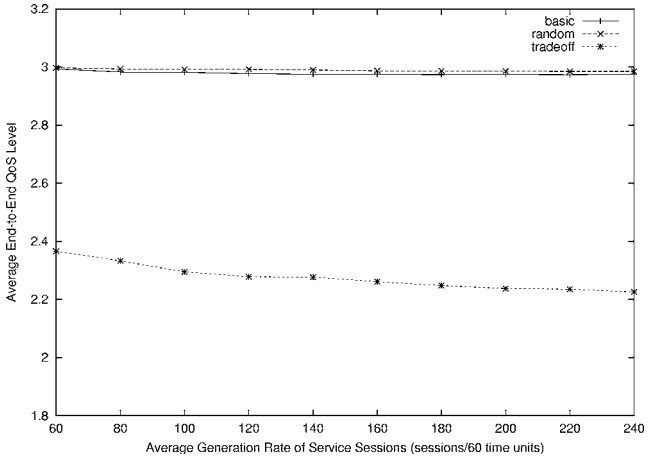


(b)

Figure 12. Overall reservation success rate with inaccurate resource availability observations: (a) algorithm *basic* and (b) algorithm *tradeoff*.



(a)



(b)

Figure 13. Overall reservation success rate and average end-to-end QoS level – under less diversified resource requirement: (a) overall reservation success rate and (b) average end-to-end QoS level.

we re-run the simulation with the assumption lifted. More specifically, for each service session, the availability of any resource may be observed up to E time units ago. Figure 12 shows the overall reservation success rates under different E values achieved by *basic* and *tradeoff*, respectively. For comparison, we also show the success rates achieved by that algorithm and by *random*, both with accurate observations. The results indicate minor to moderate performance degradation of *basic* and *tradeoff*, with the presence of inaccurate resource availability observations. However, the lowered success rates of both algorithms are still significantly higher than those of *random* with accurate observations. Furthermore, between *basic* and *tradeoff*, the lowered success rates of the latter are constantly higher than those of the former.

5.2.5. Impact of resource requirement diversity

We also study the impact of resource requirement diversity on reservation success rate. For each service component, the diversity means the difference among the required amounts of each resource, in order to reach a certain $\mathbf{Q^{out}}$ level from different $\mathbf{Q^{in}}$ levels, or to reach different $\mathbf{Q^{out}}$ levels from the same $\mathbf{Q^{in}}$ level. The greater the diversity, the more the options for resource tradeoffs and therefore, the higher the probability for an end-to-end reservation to succeed. We have also conducted extensive simulations, in which we set different resource requirement values in figure 10 for different simulation runs. The settings reflect different degree of resource requirement diversity. Our simulation results confirm the impact of resource requirement diversity on reservation success rate. Nevertheless, our results constantly show the effectiveness of our algorithms. For example, figure 13 shows the results under such an unfavorable setting: for each resource, the requirement values on different edges have the same average as that of the corresponding values in figure 10, however, the ratio between the highest and lowest values is limited to $3:1$, and the other values are evenly distributed between them. Still, both *basic* and *tradeoff* achieve higher reservation success rates than the contention-unaware *random*, although the absolute success rates are lower than the corresponding rates in figure 11(a).

6. Related work

Our framework is motivated by the recent advances in high performance distributed meta-computing (or the Grid) environments, such as Globus [8], Condor [9], and Legion [10]. In these environments, a wide collection of distributed and individually managed computing resources are connected by high speed network links; and they form a virtual and high performance platform for the deployment of numerous value-added and application-level distributed services. Besides high throughput and availability, clients of these services may also require end-to-end QoS guarantees. Our multi-resource reservation framework can potentially be integrated into these environments, in order to bring end-to-end QoS guarantees into these services.

In Globus Project, it has been argued that multi-resource co-allocation should be an integral part of the resource management architecture for the Grid [11,12]. In [11], a resource co-allocation architecture and its mechanisms for allocation, configuration, monitoring, and control are presented. Our framework complements their architecture by introducing the QoS-Resource Model and contention-awareness. An advance resource reservation mechanism is proposed in [12], in addition to the mechanism for immediate reservation. One of our next steps is to extend our multi-resource reservation framework to support advance reservations.

Taking a more theoretical approach, Lee et al. also study the problem of resource allocation for QoS guarantee [13,14]. Particularly, in [13], the problem of apportioning multiple finite resources to satisfy the QoS needs of multiple applications along multiple QoS dimensions is studied. However, their model does not consider multiple service components, which contribute transitively to the end-to-end QoS of an application. In addition, their solution is applied to a *static* set of applications to be executed at the same time. Therefore, it does not consider the dynamic arrival and completion of applications, i.e., it is not contention-aware.

Our QoS-Resource Model evolves from the EPIQ QoS management framework [15,16]. In [15], the concepts of flexible task, input/output quality, and producer/consumer dependency graph are introduced. They correspond to the service component, $\mathbf{Q^{in}/Q^{out}}$, and distributed service Dependency Graph in our model. However, no specific algorithm or protocol is presented in [15] for the computation of end-to-end multi-resource allocation. In [16], a protocol for end-to-end service establishment is proposed. The protocol shares the same performance goals as our algorithm. However, the protocol's performance depends on several key parameters, yet there is no systematic method to determine the optimal values of these parameters. On the contrary, our algorithm is self-adaptive and requires no pre-set parameters except T for Resource Brokers in algorithm *tradeoff*.

In the Darwin Project [17], a hierarchical service and resource brokerage architecture is introduced. In order to compose value-added services, allocation of multiple resources is needed. The signaling protocol for multi-resource allocation is the Beagle signaling protocol [18]. However, this protocol is not contention-aware, and there is no generic model to capture the QoS-resource relation of a service. In our earlier work of Qualman [19], different QoS-aware Resource Brokers are presented. They are responsible for the reservation and enforcement of CPU, network bandwidth, and memory resource, respectively. However, there is no coordination among these brokers, and this problem naturally leads to our current framework of multi-resource reservation.

7. Conclusion

We present a QoS and contention-aware multi-resource reservation framework. The framework consists of a formal QoS-Resource Model, a runtime system architecture for coordinated multi-resource reservation, and a runtime algorithm (and its heuristic extensions) for the computation of end-to-end reservation plans. Our framework is suitable to be integrated into the current high performance distributed meta-computing environments, in order to bring QoS guarantees into the distributed services deployed in these environments. Our simulation results show excellent performance of this framework. In particular, the basic runtime algorithm brings the highest possible end-to-end QoS level to each service session, while achieving significantly higher overall reservation success rate than the contention-unaware algorithm. Our heuristics with the "QoS-success rate" tradeoff policy achieves even higher overall success rate, at the cost of lower average end-to-end QoS level for individual service sessions.

Acknowledgements

This work was supported by the National Science Foundation under contract number 9870736, the Air Force Grant under contract number F30602-97-2-0121, National Science Foundation Career Grant under contract number NSF CCR 96-23867, NSF PACI grant under contract number NSF PACI 1-1-13006, NSF CISE Infrastructure grant under contract number NSF EIA 99-72884, NSF CISE Infrastructure grant under contract number NSF CDA 96-24396, and NASA grant under contract number NASA NAG 2-1250.

References

[1] H. Chu and K. Nahrstedt, CPU service classes for multimedia applications, in: *Proc. IEEE Int. Conf. on Multimedia Computing and Systems (ICMCS '99)* (1999).

[2] P. Goyal, X. Guo and H. Vin, A hierarchical CPU scheduler for multimedia operating systems, in: *Proc. USENIX Operating System Design and Implementation (OSDI '96)* (1996) pp. 107–122.

[3] L. Zhang, S. Deering, D. Estrin, S. Shenker and D. Zappala, RSVP: a resource reservation protocol, IEEE Network (September 1993).

[4] J. Bennett and H. Zhang, Hierarchical packet fair queuing algorithms, IEEE/ACM Transactions on Networking 5(5) (1997) 675–689.

[5] A. Demers, S. Keshav and S. Shenker, Analysis and simulation of a fair queuing algorithm, Journal of Internetworking Research and Experience 1(1) (1990) 3–26.

[6] P. Shenoy and H. Vin, Cello: A disk scheduling framework for next generation operating systems, in: *Proc. ACM SIGMETRICS '98* (1998) pp. 44–55.

[7] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms* (MIT Press/McGraw-Hill, 1990).

[8] I. Foster and C. Kesselman, Globus: A metacomputing infrastructure toolkit, Journal of Supercomputing Applications 11(2) (1997) 115–128.

[9] M. Litzkow, M. Livny and M. Mutka, Condor – a hunter of idle workstations, in: *Proc. IEEE Int. Conf. on Distributed Computing Systems (ICDCS '88)* (1988) pp. 104–111.

[10] A. Grimshaw, A. Ferrari, F. Knabe and M. Humphrey, Legion: An operating system for wide-area computing, IEEE Computer 32(5) (1999) 29–37.

[11] K. Czajkowski, I. Foster and C. Kesselman, Resource co-allocation in Computational Grids, in: *Proc. 8th IEEE Int. Symposium on High Performance Distributed Computing (HPDC '99)* (1999).

[12] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt and A. Roy, A distributed resource management architecture that supports advance reservation and co-allocation, in: *Proc. IEEE/IFIP Int. Workshop on QoS (IWQoS '99)* (1999).

[13] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar and J. Hansen, A scalable solution to the multi-resource QoS problem, in: *Proc. IEEE Real-Time Systems Symposium (RTSS '99)* (1999).

[14] C. Lee, J. Lehoczky, R. Rajkumar and D. Siewiorek, On quality of service optimization with discrete QoS options, in: *Proc. IEEE Real-Time Technology and Applications Symposium (RTAS '99)* (1999).

[15] D. Hull, M. Shankar, K. Nahrstedt and J. Liu, An end-to-end QoS model and management architecture, in: *Proc. IEEE Workshop on Middleware for Distributed Real-Time Systems and Services* (1997) pp. 82–89.

[16] M. Shankar, M. DeMiguel and J. Liu, An end-to-end QoS management architecture, in: *Proc. IEEE Real-Time Technology and Applications Symposium (RTAS '99)* (1999).

[17] P. Chandra, A. Fisher, C. Kosak, T. Ng, P. Steenkiste, E. Takahashi and H. Zhang, Darwin: customizable resource management for value-added network services, in: *Proc. IEEE Int. Conf. on Network Protocols (ICNP '98)* (1998).

[18] P. Chandra, A. Fisher and P. Steenkiste, A signaling protocol for structured resource allocation, in: *Proc. IEEE INFOCOM '99* (1999).

[19] K. Nahrstedt, H. Chu and S. Narayan, QoS-aware resource management for distributed multimedia applications, Journal of High Speed Networks (Special Issue on Multimedia Networking) 8(3–4) (1998) 227–255.

Dongyan Xu. Photograph and biography not available at time of publication.
E-mail: d-xu@cs.uiuc.edu

Klara Nahrstedt. Photograph and biography not available at time of publication.
E-mail: klara@cs.uiuc.edu

Duangdao Wichadakul. Photograph and biography not available at time of publication.
E-mail: wichadak@cs.uiuc.edu