QoS Management Middleware: A Separable, Reusable Solution

Denise Ecklund[§1], Vera Goebel[1], Thomas Plagemann[1], Earl F. Ecklund Jr. [§1], Carsten Griwodz[1], Jan Øyvind Aagedal[2], Ketil Lund[3], Arne-Jørgen Berre[2]

[1] Department of Informatics, University of Oslo, Norway
{denisee, goebel, plageman, earle, griff}@ifi.uio.no

[2] SINTEF Telecom and Informatics, Oslo, Norway
{Jan-Oyvind.Aagedal, Arne.J.Berre}@informatics.sintef.no

[3] UniK - Center for Technology at Kjeller, University of Oslo, Norway
ketillu@unik.no

Abstract. Research in the area of end-to-end Quality of Service (QoS) has produced important results over the last years. However, most solutions are tailored for specific environments, assume layered system architectures, or integrate QoS management within the respective service components, such that the QoS management functionality is not easily reusable. Furthermore, proprietary QoS solutions are not interoperable and QoS management for logical objects is not supported. In this paper, we present a separable and reusable QoS management service for end-to-end QoS in a distributed environment. This QoS middleware extends the classical feedback controller with QoS-aware agents. We describe the resulting *seven-agent QoS manager*, a generic management protocol, and define interfaces between the agents, platform services, and QoS-aware application components. Wrappers can be used to interface the QoS middleware with all types of legacy distributed service components, both QoS-aware and QoS-unaware.

1 Introduction

Software development is - despite all recent advances in software engineering - still a (time) costly and error prone task. This is especially true for the development of distributed applications and systems, because distribution generally means that the software components must function well in heterogeneous environments. For example, application components can be written in different programming languages on different operating systems and communicate with each other over different kinds of networks. The main task of middleware is to mask out this heterogeneity and provide a transparent view onto distributed systems to enable developers to implement correct software components easier and faster. Heterogeneous components also have different

§ At the time of this work these authors were senior guest researchers at UniK – Center for Technology at Kjeller, University of Oslo.

D. Shepherd et al. (Eds.): IDMS 2001, LNCS 2158, pp. 124-137, 2001.

non-functional requirements, like performance aspects and error tolerance. In response, OSI defined the concept of QoS for communication services [38] in the late 70s. In the 90s, operating systems services were defined to enable end-to-end QoS [23, 28]. In the classic understanding of QoS, service users express their needs in the form of QoS specifications and negotiate a QoS contract with the service provider. The service provider uses QoS management services like QoS negotiation, monitoring, admission control, resource reservation and allocation, and adaptation to establish and fulfill the QoS contract. Considerable research on QoS management services has resulted in numerous proposals, standards, and implementations of mechanisms to perform these services, e.g., ODP framework [17], ISO QoS framework [16], TINA QoS framework [33]. In the OMODIS[1] project [10], we reviewed the state-of-the-art in QoS management in order to develop a QoS framework for multimedia database management systems (MMDBMS). We observed the following problems in related works:

- Most QoS works (see [2]) are based on layered system architectures: Communication sub-systems are traditionally structured in layers, e.g., physical, link, network, and transport layer. The current trend in distributed applications and other subsystems is to structure them as a set of configurable components. In general, a *component* can be a system, service, or resource in a distributed environment. Component configurations are a generalization of layers in which (1) the semantics of service user and service provider are not predefined, and (2) there is no *a priori* knowledge about who will use the service of a component. Therefore, components require more generalized QoS management services than layered architectures.
- Most QoS works are tailored for specific environments: Domain-specific QoS management solutions limit the problem scope to reduce complexity, e.g., [21], or to improve performance, e.g., [11, 22]. Rather than providing a unifying solution, these single-use services simply increase the heterogeneity problem in distributed systems.
- QoS management is implemented as part of the service components themselves: Early works have integrated configurable QoS services into communication protocols and end-systems to meet new application requirements and to support end-to-end QoS guarantees, e.g., [5, 25, 37]. Others have proposed that all QoS service issues be managed by the application [34] or by end service systems [13, 33, 11, 15]. QoS mechanisms that handle concrete resources, like CPU time or disk bandwidth, must consider the particular properties of the resource. However, this typically results in QoS management techniques and mechanisms that are specific to that type of service and are thus not easily reusable.
- QoS management in Database Management Systems (DBMSs) and other complex services is an open problem: QoS work in file systems and storage systems has focused on scheduling disk bandwidth and allocating buffer space, e.g., [1, 32], leaving richer DBMS services unmanaged. For example, DBMS transactions must lock logical objects to assure data consistency. These objects must be regarded as logical resources and must be considered by QoS management services.

[1] The OMODIS project is funded by the Norwegian Research Council (NFR), Distributed IT Systems (DITS) program, 1996-2002.

Recent works have contributed positively to the state-of-the-art with feedback control based adaptation, e.g., [32, 30, 4], new solutions to QoS mapping, e.g., [12, 26, 36, 29], QoS negotiation and pricing [35], and generalized frameworks for object-oriented or component-based distributed systems [34, 6].

However, all current solutions propose a static architecture[2] comprised of a set of autonomous QoS managers. This is not effective because: (1) autonomous QoS managers can initiate conflicting adaptation strategies that are local in scope and that fail to achieve their collective QoS goals; and (2) static, pre-defined negotiation ordering among components can result in failure to reach a contract agreement. The problem of static QoS management architectures has also been identified within service-specific QoS management facilities [20]. To address the full set of identified problems, we have proposed a runtime-reconfigurable hierarchy of QoS managers [7]. Within the hierarchy, each application component is directly managed by one local QoS manager. Higher-level QoS managers direct the local managers, to co-ordinate adaptations and dynamically order QoS negotiations based on a broader view of the QoS tradeoffs among managed components and a hierarchy of QoS policies. Higher-level managers can accelerate negotiation convergence among local managers, thus reducing the cost of QoS management. Runtime adaptation includes modifying the management hierarchy by adding and removing QoS managers, updating QoS policies and strategies, and reorganizing manager relationships within the hierarchy. QoS managers are middleware services, separate and distinct from the application components they manage. Granularity of the managed component is defined by the component implementer. A local QoS manager should be associated with each separable service component that can efficiently support the minimal set of QoS management interfaces. Our scalable solution uses two types of reusable QoS managers. In this work, we focus on the local QoS manager, as a fundamental building block of our hierarchical, reconfigurable solution. We define the architecture of a local QoS manager and its interactions with a managed component. This architecture defines an abstraction over different component types and over different QoS management solutions. The primary contribution of this work is to provide a separable, reusable middleware solution that can manage any type of service component, use existing (legacy) QoS management solutions, and be organized in a dynamically configurable management hierarchy.

The remainder of this paper is organized as follows: We present the QoS management requirements for a distributed application. Based on these requirements, we describe the design of our reusable QoS manager. Then, we present the QoS-aware management agents within the QoS manager. Next, we demonstrate the feasibility of our approach in an example execution sequence. In the following section, we define the necessary interfaces between the QoS manager and a managed component. We conclude with a summary of the contributions of this work and an outline of our future work in this area.

[2] Note that a static architecture does not exclude the use of dynamic mechanisms like dynamic routing.

2 Requirements

We consider session-oriented applications in a distributed environment. Two popular system architectures supporting such applications are: the *N-tiered Server Architecture*, for synchronous multi-server applications, and the *Multi-agent Collaboration Architecture*, for mobile computations and asynchronous applications, such as workflow.

Fig 1 shows a simple, two-tiered server architecture, with six major components and seven QoS managers. Each component has a local QoS manager, and in this simple case, a single session-level QoS manager coordinates the local QoS managers. During a session, the client system submits requests to the application server over Net1. The application server requests data from a MMDBMS over Net1, and html pages from a web server over Net2. The application server performs post-processing on the data and sends the resulting data streams to the client system for presentation. The quality of service received by the client depends on the aggregate QoS provided by the three servers and the two networks.

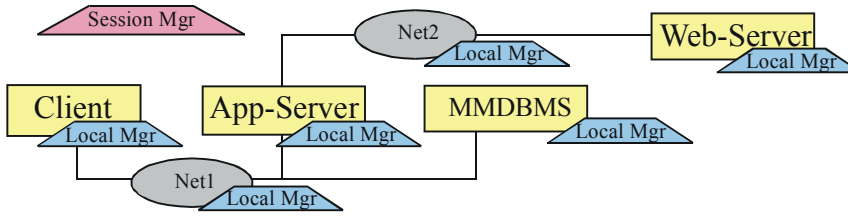

Fig. 1. Example two-tiered server architecture with six components and seven QoS managers.

By controlling the characteristics that affect its QoS, a system can predictably and effectively deliver satisfactory performance to its users. Delivering QoS to a client (user) is managed by QoS contracts, each matching a QoS offer of a component with a QoS requirement of the client.

QoS management services are responsible for establishing, controlling, and maintaining QoS contracts for long-running sessions. Typical QoS management services include: negotiation of end-to-end QoS contracts, detection of QoS violations, initiation of service adaptation(s) to regain QoS conformance, and renegotiation of active QoS contracts when service adaptation is insufficient. In addition to the functional requirements for QoS management services, QoS management must provide a general, minimally-invasive, low overhead, reusable solution for all types of components.

3 Reusable QoS Manager

One of the major tasks of QoS management is to control components such that their cooperating services, i.e., observable behaviors, correspond to a client's QoS require-

ments. The traditional concept of feedback and closed-loop control theory defines models and mechanisms to perform such control tasks. Feedback control has been investigated as an approach to QoS management, but limited to layered services, domain-specific environments, or components with integrated controllers [32, 30, 4]. Even when applied in a general environment of interacting components, classical feedback control is not a complete solution for QoS management because it works best in fully deterministic environments. Therefore, we extend the traditional feedback controller [8] with QoS management agents that respond to the non-deterministic aspects of the runtime environment. In the following, we define the type of components we intend to manage, explain the deficiencies of the classical controller architecture, and introduce extensions to form a complete QoS manager for a general distributed environment.

3.1 Managed Components

In an open distributed system, a *Managed Component* (MC) is any system, service, or resource that presents itself as an atomic entity to a QoS manager. Components may be *QoS-aware* or *QoS-unaware*. Example QoS-aware components include: *Self-adapting components* [32], *QoS-mechanistic components* [25], and *QoS-managed subsystems* (such as QuO [34] for communications, command, and control applications, and QoS-A [5] for network services) in addition to QoS-support mechanisms.

QoS mechanisms contained in QoS-aware components are service-specific algorithms for maintaining QoS contracts held by that component. For example, a QoS-aware network service can implement flow filtering for multimedia streams, and channel sharing for specific types of network traffic. These mechanisms are not generic QoS management services, but a generic, reusable QoS manager can control such components using wrappers, that map portions of a common QoS management interface to such service-specific mechanisms.

3.2 Classical Feedback-Based Controllers

The classical closed-loop, feedback controller consists of four agents: *estimator*, *regulator*, *prober*, and *observer* [8]. The managed component is modeled by a *component state vector* that contains values for the state variables, capturing component state at a particular point in time. The estimator sets the values of the component state vector based on a pre-defined component model for the managed component and runtime input from the observer and the prober. The regulator uses the estimated state vector to generate effective input control values to the component.

The classical controller is not sufficient for QoS management. The primary deficiency arises from the fact that classical controllers are most effective in deterministic environments. An open environment of cooperating components is non-deterministic in several respects: some components cannot be predictably controlled, components do not have a deterministic workload, the migration of collaborating agents is based on runtime data, and components do not have dedicated access to external, shared resources in their base platforms. In addition, the classical controller monitors and

controls a component based on attributes that summarize the component's state in general. These attributes may not relate directly to the QoS provided by the component. And the attributes do not support fine-grained control of QoS on a per-client basis. Also a highly complex component model may be needed to support the rapid and extreme forms of change required to adaptively support QoS in a non-deterministic environment.

3.3 Architecture of the QoS Manager

To address these shortcomings, we define a QoS manager consisting of seven QoS management agents: a *four-agent controller* and *three QoS-aware agents*. The QoS-aware agents create and maintain QoS contracts on behalf of a component. We also extend the traditional controller behavior as follows:

- Performance data gathered by the observer and the prober is partitioned and associated with specific component activities. A *component activity* is a separately identified unit of service performed by the component. An activity can correspond to one client or a set of related clients. QoS performance is monitored per component activity. Aggregate statistics may be computed over sets of activities.
- Estimator and regulator use a dynamically sized state vector to exert control on a per-activity basis. As activities are initiated and terminated, the subset of state variables used to control one activity are added to or deleted from the total component state vector.

The three QoS-aware management agents perform QoS-specific management services, including negotiation, admission control and reservation, and service adaptation. QoS management policies and QoS profiles for components are computed offline and periodically installed by system administrators. The QoS management agents use this pre-computed information as well as runtime information on QoS performance to manage QoS on behalf of components. The agents also make effective use of the classical controller by mapping between QoS goals (stated as QoS contracts) and the state vector, which the controller uses to control the component. The architecture of the seven-agent QoS manager is shown in Fig 2.

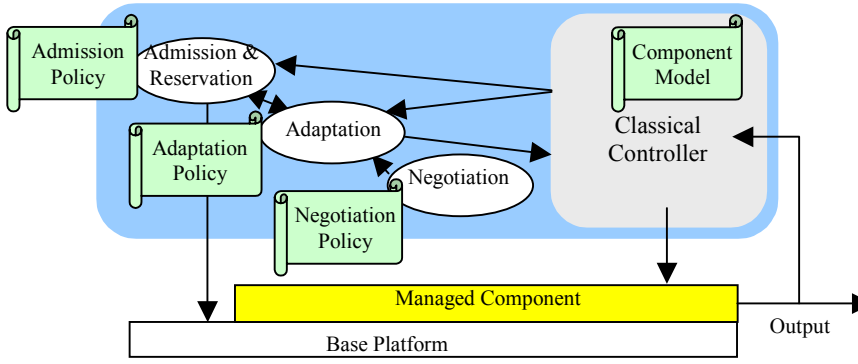

Fig. 2. Architecture of the seven agent QoS Manager.

4 QoS-Aware Management Agents and Platform Services

4.1 Negotiation, Renegotiation Agent

The *QoS Negotiation Agent (QNA)* implements a general negotiation protocol [12, 19] between two QoS managers, to construct an acceptable QoS contract between two parties, one acting as a client and the other as a server. To carry out a successful external negotiation, the QNA uses the services of the local QoS agents and operates on pre-computed, component-specific information, such as: *QoS profiles of the local component*, which can be defined offline and modified at runtime using an interpreted QoS specification language [24, 3]; *QoS mapping functions*, which define a many-to-many mapping of QoS characteristics between a client's QoS requirements and a server's QoS offers; and *QoS negotiation policy*, which governs how QoS requirements and QoS offers can be modified during the negotiation process, and includes the policy for translating actual costs into offered prices when agents of separate domains negotiate [18, 35].

Before committing to a QoS contract, the QNA uses the local QoS adaptation agent (QAA) and the admission control and reservation agent (ACRA) to analyze resource requirements and reserve the necessary local resources. The QAA proposes component configurations each of which could support the client's QoS requirements and determines the resource requirements of those various configurations. The ACRA admits or rejects a proposed configuration. If admitted, the ACRA reserves the required resources. A later section presents a detailed example of local interactions during negotiation.

4.2 Adaptation Agent

The *QoS Adaptation Agent (QAA)* is responsible for defining *quality-based configurations* of the local component. Quality configurations are derived from component configurations that support the functional requirements of a client request. Each quality configuration should be able to support the contracted level of QoS, possibly at a different cost. Traditionally, localized adaptation has been proposed as a mechanism to maintain QoS when violations occur within an ongoing session. Adaptation can also be applied: (1) during initial QoS contract negotiation to achieve an optimum, initial component configuration, and (2) across QoS-conformant sessions to compensate for other sessions that are experiencing QoS violations. To define a quality configuration from a functional configuration, the QAA needs the following pre-computed information: *Local component description*, which documents component structure and functionality; *Adaptation policies*, which control how the local component can be reconfigured; *Adaptation strategies*, which transform one component configuration into another configuration that should better meet specific QoS requirements; and *Adaptation evaluation metrics*, which compute the worth of a proposed component configuration.

The QAA obtains the current component state from the estimator agent and uses this as additional input to the configuration evaluation process. Once an adapted con-

figuration has been approved for use, the QAA must represent that configuration in the form of a component state vector that can be used by the regulator to make the component conform to the adapted configuration. The QAA updates only the subset of the component state vector associated with the affected activities. If the QAA is unable to define an acceptable quality-based configuration, the QAA returns a list of unachievable QoS requirements and the specific reasons for failure. Example failure conditions include: transient or permanent resource insufficiency, inadmissible client requester, minimum cost exceeds client's price limits, etc. Ultimately, the session-level QoS manager uses this information to direct contract re-negotiation or to reject the end-client request.

4.3 Admission Control and Reservation Agent

The *Admission Control and Reservation Agent (ACRA)* admits client requests based on component-specific admission policies and resource availability, and reserves all required resources. To accomplish these tasks, the ACRA needs the following component-specific and request-specific information: *Raw resource requirements*, expressed as a runtime-computed resource schedule for the admitted request; *Raw resource status*, expressed as a runtime-computed aggregate resource schedule over all clients on the local base platform, and the cost to reserve and use those resources; and pre-computed, but updateable, *Admission policies*, which define conditions for admitting a client request for service by the local component.

If the admission policy allows the request to be served and platform resources are available, then the ACRA reserves the required resources. If the request cannot be admitted, the ACRA rejects the request and returns the reasons for the rejection. The session-level QoS manager uses this information to direct contract re-negotiation or to reject the end-client request.

4.4 Platform Resource Management

Components consume platform resources in order to provide their services. The quality of that service depends on the sufficiency of available resources and the ability to use resources according to a schedule. Local resources are managed by one or more local *Platform Resource Managers* (PRMs).

Distributed applications operate on logical data objects, such as video clips, audio sound tracks, web pages, data folders, and geographical maps. In many applications (such as updating multimedia objects stored in a MMDBMS), a client must obtain a lock before accessing the object. Once locked, the client has exclusive access to the logical object. Physical resources are required to carry out operations on a logical object. Clearly physical resources should not be reserved for a client prior to obtaining the required set of logical locks. A logical lock has an application-specific semantic. Hence, this defines a strict ordering between a component setting logical locks on objects and a QoS manager reserving physical resources on behalf of the component that will access and manipulate those objects.

5 Example Execution Among Local QoS Agents

The following example illustrates local interactions among the QNA, QAA, and ACRA in support of contract negotiation between two QoS managers. Fig 3 shows process flow among the agents and the corresponding agent activity is described below.

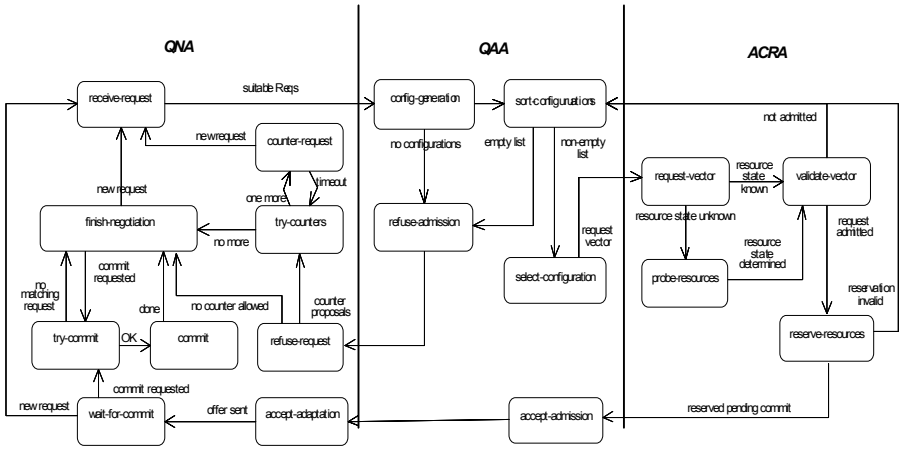

Fig 3. Process flow among local QoS agents supporting contract negotiation.

Receive-request. The QNA receives a set of QoS requirements defined by the requesting application and other service components involved in the application configuration. Based on the requested data and services, the QNA selects, from the list of QoS requirements, the requirements involving the QoS characteristic types that can be served by the component for this specific request.

Config-generation. The QAA receives a subset of QoS requirements. Using an Architecture Definition Language tool [9] or another runtime tool, the QAA obtains component configurations that meet the functional requirements of the request. The QAA matches the QoS requirements to the possible configurations and selects potential component configurations.

Sort-configurations. The QAA sorts the potential component configurations according to the preference of the adaptation policy. If the QAA has received reasons for the refusal of a previously checked configuration, the policy may take this into account and change the order, or the policy may not change the existing order at all.

Select-config. The QAA removes the preferred component configuration from the sorted list and translates it into a component state vector for the configuration. This state vector ("request vector") represents the impact of serving the new request at the requested QoS levels on an unloaded system.

Request-vector. The ACRA receives the request vector. The ACRA estimates the future state of the component, if the request was served at the requested level of QoS.

Probe-resources. The ACRA probes PRMs to obtain information on the current state of raw platform resources. (Raw platform resources are related to component state, but

they are not identical. They include the resource usage of all QoS-managed and QoS-unmanaged components executing on the platform).

Validate-vector. The ACRA examines the predicted future state vector and consults the local admission control policy to determine whether to admit the new request. If the admission policy rejects the state vector, a notification of the refusal including an explanation is sent to the QAA.

Reserve-resources. The ACRA tries to reserve the resources specified in the accepted state vector from the PRMs. If the future state is supported by the PRMs and the requested QoS maintenance mode requires resource reservations, then the resources are reserved, the component state vector is updated, and a notification of acceptance is sent to the QAA. Otherwise, the ACRA sends a notification of the refusal including an explanation to the QAA.

Accept-admission. The QAA has received a positive response. The response is forwarded by the QAA to the QNA.

Accept-adaptation. The QNA informs the caller (application or remote QNA) about the success. It includes two QoS statements in the notification. The first is the QoS offer specifying the level of QoS that the local component has agreed to provide. The second is the QoS requirement specifying the level of QoS required by the local component to ensure that it meets its contracted-level of QoS.

Wait-for-commitment. The caller's request has been answered positively. The QNA maintains all information about the ongoing negotiation.

Try-commit. The QNA validates that the request matches an existing pre-commitment. If the validation fails, the caller is informed about the error.

Commit. The QNA calls the QAA, which in turn calls the ACRA to commit the pre-committed resources. If the commitment fails, the caller is informed about the error.

Refuse-admission. All possible configurations were rejected by the ACRA. The QAA creates a response, including reasons for the refusal, and sends it to the QNA.

Refuse-request. The QNA receives a negative response. If the negotiation policy does not allow counter-proposals, the caller is informed about the refusal of the request.

Try-counter-request. If the number of negotiation attempts for a given end-user session has reached a policy-defined limit, the QNA informs the caller about the negotiation failure, including the reasons in the response.

Counter-request. The QNA updates the number of attempts and makes a counter-proposal to the caller. The counter-proposal contains QoS requirements that have been modified based on feedback from the QAA and local negotiation policy. The negotiations continue, and the QNA expects another proposal from the caller. A timer is started to discard the session if no response to the counter-proposal arrives within a policy-defined time.

Finish-negotiation. The caller's request has been answered. The QNA discards remaining information about the negotiation and waits for further requests.

6 Interface Specifications for QoS Management

Based on the execution flow, it is clear that the QoS management agents must interact with the managed component at specific times during the QoS negotiation process

(and also when performing other QoS management services). For example, the ACRA cannot reserve resources on behalf of the component without knowing which types and amounts of resources are required to serve the request at the contracted QoS level. The four-agent classical controller must also interact with the component to probe for component state and to control component behavior. Controller interfaces are not QoS-specific, hence they are not presented here. Manager-to-manager interfaces are overviewed in [7]. Table 1 presents the QoS-support interfaces that must be implemented by a managed component or a wrapper encapsulating a managed component, and the QoS-support interfaces that must be implemented by a PRM.

Implementer Interfaces to Support Specified Service		
Caller	Interface Prototype	Functional Description
Managed Component Interfaces to Support QoS Negotiation and Renegotiation		
QAA	[ClientID, {FunctionalConfiguration}] = **ControlAndConfigure** (ClientRequest)	Obtain a set of component configurations that functionally support the client request.
QAA	[ResourceSchedule, CostEstimate] = **LockObjectsAndEstimateResources** (QualityConfig, ClientRequest)	The component sets logical locks and returns the resource schedule required to service the client request.
QNA	Status = **Process** (ClientID, "commit/abort", ReservationHandle)	Informs the component of the final outcome (commit/abort) of the distributed negotiation.
PRM Interfaces to Support QoS-based Admission and Resource Mgmt		
ACRA	{($Rsrc_i$ID, $Rsrc_1$Schedule, $Rsrc_i$Cost)} = **ProbeResourceState** (DomainID, ResourceSchedule, ReservationHandle)	Gets the cost for a new or modified resource schedule and the PRM's current aggregate resource schedule.
ACRA	[ReservationHandle, CostEstimate, RejectionReasons] = **ReserveResources** (ActivityID, DomainID, ResourceSchedule, ReservationHandle)	Reserves a resource schedule for a new request or modifies an existing reservation. The reservation supports future allocations.
MC	[Status, {$AllocationHandle_i$}] = **AllocateResources** (ActivityID, ReservationHandle, {R_iSchedule})	Allocates resources for immediate scheduling or use. The reservation handle may be null.
MC	{($Rsrc_i$ID, $Rsrc_1$Schedule, $Rsrc_i$Cost)} = **ReleaseResources** (ClientID, {$AllocationHandle_i$})	Releases one or more previously allocated resources and updates the resource schedule.
Managed Component Interfaces to Support QoS-motivated Adaptation		
QAA	[ResourceSchedule, CostEstimate] = **EstimateResources** (ClientID, AdaptedConfig)	Sets logical locks and estimates resource requirements to modify an executing client request.
QAA	Status = **MigrateService** (ClientID, "commit/abort")	Informs the component of the final outcome of the adaptation process.

Table 1. Summary of QoS management interfaces.

7 Conclusions

This work is motivated by the fact that each existing solution for end-to-end QoS management suffers from one or more of the following restrictions (or disadvantages): (1) it is tailored for specific environments, (2) it assumes layered system architectures, (3) it implements QoS management within the respective service components such that the QoS management functionality is not easily reusable, (4) it does not interoperate with other proprietary QoS solutions, and (5) it does not manage QoS for logical objects, which are used by complex components, such as a DBMS. The QoS management middleware described in this paper overcomes these problems. By separating QoS management services from the managed component and defining an interface between them, we have created a reusable, generic solution for QoS management of arbitrary components in open distributed and heterogeneous environments. Similar to other recent approaches, we use feedback control in our QoS management middleware. However, classical feedback control was not designed to manage components in a non-deterministic environment, nor to exert control on a per-client basis. Therefore, we have extended the classical four-agent controller architecture, with three QoS-aware agents supporting negotiation, admission control and reservation, and adaptation. We identified dependencies between the QoS agents, the managed component, and the basic platform services. A critical dependency is that components must obtain logical locks on data objects before using platform resources to manipulate those objects. We defined a set of component interfaces to support a reusable QoS manager. In addition, we defined interfaces between the QoS manager and a platform resource manager supporting reservations to a resource schedule.

For our ongoing and future work on components, we are implementing the QoS management interfaces as defined in this work. To integrate legacy components, we are investigating wrappers for specialized QoS managers, such as QuO [34] and MULTE [27], as well as for individual QoS-aware services, such as RSVP [14]. Each wrapper defines a model for estimating component state information that is not provided by the legacy system, and makes runtime estimations within acceptable time- and resource-cost. Experiments will be performed using QLinux [31], which provides QoS-aware resource management services in the base platform. We are also continuing our investigation of QoS manager interaction with, and QoS mechanisms within, a compound service component for management of persistent multimedia objects.

References

1. Aberer K., Hollfelder, S., Resource Prediction and Admission Control for Interactive Video, Proc. 8th IFIP 2.6 Working Conf. on Database Semantics - Semantic Issues in Multimedia Systems (DS-8), Rotorua, New Zealand, Kluwer, Jan. 1999, pp. 27-46
2. Aurrecoechea, C., Campbell, A.T. and L. Hauw, A Survey of QoS Architectures, ACM Springer Multimedia Systems Journal, Special Issue on QoS Architecture, Vol. 6, No. 3, May 1998, pp. 138-151
3. Becker, C., Geihs, K., Generic QoS Specifications for COBRA, Kommunikation in Verteilten Systemen, 1999, pp.184-195

4. Bergmans, L., van Halteren, A., Ferreira Pires, L., van Sinderen, M., Aksit, M., A QoS-Control Architecture for Object Middleware, 7th Intl. Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS 2000), Enschede, The Netherlands, Oct. 2000, LNCS 1905, pp.117-131
5. Campbell, A., Coulson, G., Hutchinson, D., A Quality of Service Architecture, ACM Computer Communications Review, Vol. 24, No. 2, April 1994, pp. 6-27
6. Daniel, J., Modica, O., Traverson, B., Vignes, S., Modeling and Enforcement of Quality of Service in Distributed Environments, 2nd Intl. Symp. on Distributed Objects and Applications (DOA '00), Antwerp, Belgium, Sept. 2000
7. Ecklund, D., Goebel, V., Plagemann, T., Ecklund Jr., E.F., A Dynamically-Configured, Strategic QoS Management Hierarchy for Distributed Multimedia Systems, Technical Report of the University of Oslo, Informatics Department, April 2001
8. Franklin, G. F., Powell, J. D., Emami-Naeini, A., Feedback Control of Dynamic Systems, Addison-Wesley Publishing Company, Aug. 1986
9. Proc. First Intl. Workshop on Architectures for Software Systems, D. Garlan Editor, Seattle, WA, April 1995
10. Goebel, V., Plagemann, T., Berre, A.-J., Nygård, M.: OMODIS - Object-Oriented Modeling and Database Support for Distributed Systems, Norsk Informatikk Konferanse (NIK'96), Alta, Norway, Nov. 1996
11. Gopalakrishna, G., Parulkar, G., A Real-time Upcall Facility for Protocol Processing with QoS Guarantees, 15th ACM Symp. on Operating Systems Principles (SOSP), Dec. 1995
12. Hafid, A., Bochmann, G. V., Quality-of-Service adaptation in distributed multimedia applications, Multimedia Systems, ACM Springer, Vol. 6, No. 5, 1998, pp. 299-315
13. Braden, R., Clark, D., Shenker, S., Integrated Services in the Internet Architecture: An Overview, IETF Internet RFC 1633, June 1994
14. Braden,R., Zhang, L., Berson, S., Herzog, S. Jamin, S., RFC 2205, Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification, IETF, Sept. 1997
15. Blake, S., Black, D., Carlson, M., Davies, E., Wang, W., Weiss, W., An Architecture for Differentiated Services, IETF Internet RFC 2475, Dec. 1998
16. QoS – Basic Framework, ISO, ISO/IEC JTC1/SC21 N9309, 1995
17. Working document on QoS in ODP, ISO, ISO/IEC JTC1/SC21 WG7, 1995
18. Karsten, M., Schmitt, J., Wolf, L., Steinmetz, R., Provider-Oriented Linear Price Calculation for Integrated Services, Proc. 7th IEEE/IFIP Intl. Workshop on Quality of Service (IWQoS'99), London, UK, June 1999, pp. 174-183
19. Koistinen, J., Seetharaman, A., Worth-Based Multi-Category Quality-of-Service Negotiation in Distributed Object Infrastructures, Hewlett Packard Software Technology Laboratory Technical Report HPL-98-51 (R.1), July 1998
20. Kristensen, T., Kalleberg, I.B., Plagemann, T., Implementing Configurable Signalling in the MULTE-ORB, To appear in Proc. 4th IEEE Conf. on Open Architectures and Network Programming, Anchorage, AK, April 2001
21. Lazar, A.A., Lim, K.-S., Marcocini, F., Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture, IEEE Journal on Selected Areas in Communication, Vol. 14, No. 7, Sept. 1996, pp. 1214-1227
22. Lee, S.-B., Ahn, G.-S., Zhang, X., Campbell, A.T., INSIGNIA: An IP-Based Quality of Service Framework for Mobile ad Hoc Networks, Journal on Parallel and Distributed Computing, Academic Press, 60, 2000, pp. 374-406
23. Leslie, I.M., McAuley, D., Mullender, S.J., Pegasus – Operating Systems Support for Distributed Multimedia Systems, ACM Operating Systems Review, Vol. 27, No. 1, 1993
24. Loyall J.P., Schantz R.E., Zinky J.A., Bakken D.E., Specifying and Measuring Quality of Service in Distributed Object Systems. Proc. First Intl. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC '98), 20-22 April 1998, Kyoto, Japan

25. Nahrstedt, K., Smith, J.M., Design, Implementation, and Experiences of the OMEGA End-Point Architecture, IEEE Journal on Selected Areas in Communications, Vol. 14, No. 7, Sept. 1996, pp. 1263-1279
26. Nahrstedt, K., Chu, H., Narayan, S., QoS-Aware Resource Management for Distributed Multimedia Applications, Journal on High-Speed Networking, Special Issue on Multimedia Networking, Vol. 7(3,4), Spring 1999, pp. 229-258
27. Plagemann, T., Eliassen, E., Hafskjold, B., Kristensen, T., Macdonald, R. H., Rafaelsen, H.O.: Flexible and Extensible QoS Management for Adaptable Middleware, Intl. Workshop on Protocols for Multimedia Systems (PROMS 2000) Cracow, Poland, Oct. 2000
28. Saltzer, J, Reed, D, Clark, D, End-to-end Arguments in System Design, ACM Trans. on Computer Systems, Vol. 2, No. 4, Nov. 1984, pp. 277-288
29. Siqueira, F. Quartz: A QoS Architecture for Open Systems, Trinity College Dublin, TCD-CS-2000-05, Ph.D. Thesis, Feb. 2000
30. Steenkiste P., Adaptation Models for Network-Aware Distributed Computations, 3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC99), Orlando, FL, IEEE Springer, Jan. 1999
31. Sundaram, V, Chandra, A, Goyal, P, Shenoy, P, Sahni, J, Vin, H, Application performance in the QLinux multimedia operating system, Proc. 8th Intl. Conf. on Multimedia, Marina del Rey, CA, Oct. 2000, pp. 127-136
32. Thimm H., Klas W., Walpole J., Pu C., Cowan C., Managing Adaptive Presentation Executions in Distributed Multimedia Database Systems, Proc. Intl. Workshop on Multimedia Database Management Systems, 1996
33. Quality of Service Framework, Telecommunications Information Networking Architecture Consortium (TINA-C), TP_MRK.001_1.0_94, 1994
34. Vanegas, R., Zinky, J., Loyall, J., Karr, D., Schantz, R., Bakken, D., QuO's Runtime Support for Quality of Service in Distributed Objects, Proc. IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, England, Sept. 1998
35. Wang, X., Schulzrinne, H., An Integrated Resource Negotiation, Pricing, and QoS Adaptation Framework for Multimedia Applications, IEEE Journal on Selected Areas in Communications, Vol. 18, 2000
36. Witana, V., Fry, M., Antoniades, M., A Software Framework for Application Level QoS Management, Proc. 7th Intl. Workshop on Quality of Service (IEEE/IFIP IWQoS '99), June, 1999
37. Wolf, L.C., Resource Management for Distributed Multimedia Systems, Kluwer Academic Publishers, 1996
38. Zimmerman, H., OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection, IEEE Trans. on Communications (COM), Vol. 28, No. 4, April 1980, pp. 425-432