$2K^Q$ QoS Service

January 24, 2001

1 INTRODUCTION

$2K^Q$ *QoS Service* is proposed as a next generation QoS management framework for the distributed component-based applications in a ubiquitous, heterogeneous, dynamic computing and communication environments. It includes not only QoS setup or QoS provision during an application's run-time, but also QoS programming and compilation for different application domains during an application development.

The rest of the report is organized as follows. In Section 2, we describe the $2K^Q$ QoS Service architecture. In Section 3, we describe the application QoS model used in the architecture. In Section 4, we present the detailed system design of the architecture. In Section 5, we present the IDL interfaces, and the detailed implementations. In Section 6, we describe how to compile and execute the $2K^Q$ QoS Service in a distributed component-based environment. In Section 7, we demonstrate how the $2K^Q$ QoS Service can be deployed in an active space. In Section 8, we discuss the related work. Finally, in Section 9, we list the work needed to be continued.

2 $2K^Q$ QoS SERVICE ARCHITECTURE

The $2K^Q$ QoS Service architecture consists of two phases: the *distributed compilation phase* and the *run-time instantiation phase.*

The *distributed compilation phase* provides the programming environment and the QoS compilation which help an application developer to develop a QoS-aware application in different domains into the heterogeneous, and ubiquitous distributed component-based systems systematically and easily. The distributed compilation phase generates the meta data which helps to reduce the overhead of QoS setups corresponding to different constraints, and QoS adaptations corresponding to

QoS violation, user mobility, and changes of user preference, effectively. The *run-time instantiation phase* instantiates a distributed component-based application into the distributed nodes corresponding to the suggested meta data generated during the distributed compilation phase, and the dynamic constraints such as current resource availability, the specific QoS requirements, the service component locations, and the execution environments such as different active spaces.

The *run-time* $2K^Q$ is a component-based, two-tier distributed middleware system consisting of the quality-aware resource management, and the quality-aware service management as shown in Figure 1. It provides the mechanisms which helps the distributed compilation and the run-time instantiation to ensure an end-to-end QoS provision. The *quality-aware resource management* is

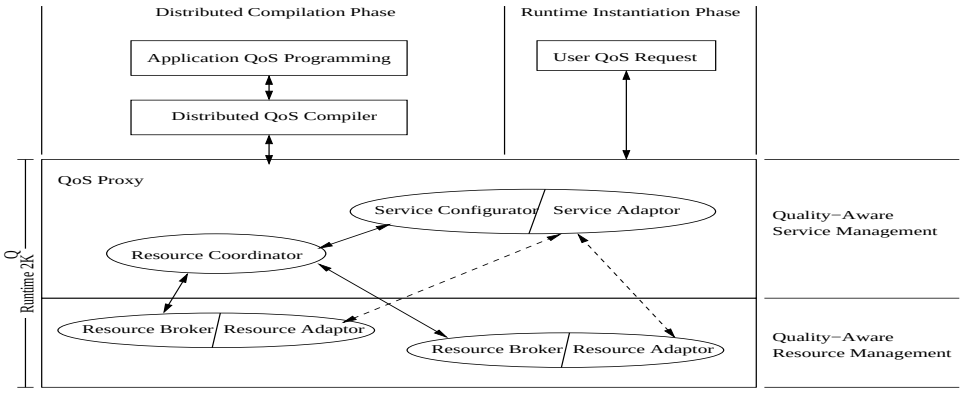

Figure 1: Reconfigurable Component-based Run-time System ($2K^Q$)

based on the concept of **resource brokers** which have the capabilities of the QoS-aware resource probing, admission, negotiation, reservation, allocation, enforcement and data adaptation. The run-time $2K^Q$ quality-aware resource management relies on the QualMan system [1] which delivers the above capabilities for CPU [2], memory and network resources. The *quality-aware service management* uses the **QoS proxy** entity to provide capabilities such as the distributed service component instantiation, the service configuration selection, the dynamic service component location discovery, the resource coordination among distributed service components, and the service component reconfiguration (functional adaptation). In the following subsections, we describe the distributed compilation phase and the run-time instantiation phase in mode details.

2.1 DISTRIBUTED COMPILATION PHASE

In the distributed compilation phase, the main entities are *QoS specifications*, and *distributed QoS compiler*.

QoS specifications are needed in a QoS middleware architecture because of the following reasons: (1) QoS specifications connects an end-application to the underlying QoS middleware; (2) an application uses QoS specifications to specify its QoS requirements; (3) the underlying QoS middleware provides quality of services based on the specified QoS specifications. How the QoS specifications look like depends on the design and the objectives of a specific QoS middleware architecture. For example, in QuO project [3], QoS specifications are specified via a suite of description languages based on aspect-oriented programming [4]. In QoSME [5], QoS specifications are specified via a Quality of Service Assurance Language (QuAL) which provides the abstractions for QoS management for an application developer to specify QoS specifications for an application. In CORBA, QoS specifications are specified via the pre-defined interfaces of different QoS services such as QoS messaging [6], audio/video streaming service [7], fault tolerant CORBA [8], and real-time CORBA [9]. In Agilos middleware [10], QoS specifications are defined via rule-based and membership functions. In Q-RAM project [11], QoS specifications are specified via user utility functions. While QoS specifications are specified differently corresponding to their associated QoS middleware architecture, QoS specifications share the following characteristics: (1) they are application specific; (2) they need translations from the application level QoS parameters into the underlying system QoS parameters to ensure QoS provisions; (3) their associated QoS middleware architectures are tailored toward specific type of applications.

To provide a QoS middleware architecture, which supports *different* types of QoS-aware applications, in ubiquitous computing environments, we introduce the following QoS specifications: (1) *application description*; (2) *set of application functional graphs*; (3) *service component description*; and (4) *user to application-specific translation template (UtoA template)*. These QoS specifications are represented via a set of entities comprising a QoS-aware application *programming environment* for an application developer to develop an application with QoS considerations systematically and flexibly during the distributed QoS compilation phase. The proposed QoS specifications are extended beyond a specific type of applications. They are programmable entities which are cus-

tomizable to specific applications. The UtoA template helps the distributed QoS compilation to perform different translations flexibly and appropriately.

Distributed QoS compilation Phase is needed because applications in a ubiquitous, highly dynamic, distributed environments might require different delivery forms, different QoS translations, structures and parameters. Hence, we need to examine, translate (and sometimes even instrument) the application and find out the QoS relations so that when the application enters the run-time phase with the goal to achieve a QoS contract, the end-to-end QoS instantiation can be done efficiently. This is similar to the language compilation concept where an application source code is optimized and translated into an object code by the language compiler, so that during the run-time we get a highly optimized and high-performance program. The distributed QoS compiler's "object code" is used by the run-time system for setup of end-to-end QoS-compliant paths, QoS degradation during resource fluctuation, or application reconfiguration corresponding to the changes of system resource availability, the changes of execution environments such as service component mobility, and service component migration, or the changes in user preference.

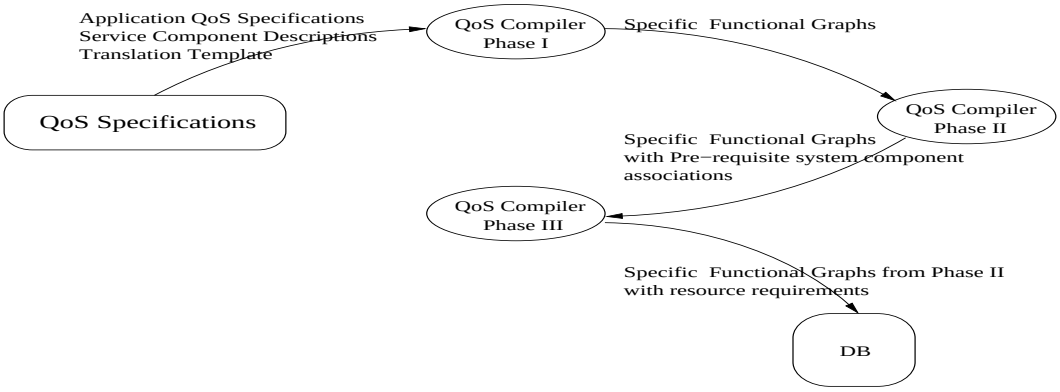

Figure 2: QoS Compiler Model

The distributed QoS compiler model consists of three phases as shown in Figure 2. The previously described QoS specifications[1] are the *"source code"* of the distributed QoS compiler. The *distributed QoS compilation* is then performed on the QoS specifications as follows: (Phase I) QoS compiler translates QoS specifications into specific functional graphs; (Phase II) QoS compiler associates each specific functional graph with the pre-requisite system component(s) and their system level QoS parameters based on a translation model; (Phase III) QoS compiler translates from com-

[1]Application developer specifies QoS specifications via the QoS Editor, our visual QoS programming tool [12].

piled specific functional graphs (acquired in phase II) into resource requirements, and generates a data structure consisting of the translated results associated with the application description and service component descriptions as the *"ready-to-use" ("object code") information*, so called *"QoSCSpec"*. The QoSCSpec helps the end-to-end QoS instantiation to be done efficiently during the application run-time instantiation.

The *QoS compiler phase I* is introduced to generate all possible compositions (configurations), of an application, as alternative delivery forms corresponding to the run-time instantiation's constraints such as resource availability, execution environment and user preference. The QoS compiler phase I is based on the generic service components' type substitutions, and the inter-component QoS consistency check.

The *QoS compiler phase II* is introduced to support QoS provisions in different application domains (e.g. multimedia, messaging) which have different semantics of QoS requirements in the unified QoS middleware framework. It automatically decides an appropriate set of system components (e.g. resource brokers, middleware QoS services) and their suitable QoS parameters for an application functional graph corresponding to the application level QoS parameters specified in the application's UtoA template. The QoS compiler phase II is based on the pre-requisite system component association and the code instrumentation.

The *QoS compiler phase III* is introduced to translate the associated configurations generated in phase II into distributed system resource requirements. It is dealing with coordination among distributed resource brokers, resource negotiation, and resource translation. This phase is needed to ensure an end-to-end QoS guarantee from the resource provision point of view. Note that the resource translation in our QoS middeware architecture relies on the optimistic assumption that its result is the minimum resource requirements to instantiate a configuration. If the QoS violation occurs due to the changes of resource availability, the QoS adaptations will take place. The QoS compiler phase III is based on the analytical translation, and the distributed probing protocol.

2.2 RUN-TIME INSTANTIATION PHASE

The **run-time instantiation** is handled by the *run-time* $2K^Q$ based on *QoSCSpec* generated by the QoS compiler. The run-time instantiation needs to provide (1) a *QoS setup* protocol for the application's setup of end-to-end QoS guarantees according to specified user QoS levels, and (2)

adaptation services during the applications execution in case of resource fluctuation, user mobility, and changing of user preference. There are three phases during the QoS setup: *service polymorphism*, *service discovery*, and *service reservation*.

Service polymorphism [13] is dealing with service configuration selection. It consults the QoSCSpec repository to get all possible configurations corresponding to a user request for an application constrained by a requested user QoS level, execution environment, and current resource availability. The service polymorphism, then, selects the best configuration among the returned configurations from the repository. If the selected configuration consists of an undefined location (defined by the wildcard (*)) for a specific service component, the *service discovery* [14] will be activated to discover the specific service component's best location. When specific service components are set in their locations (e.g., distributed target nodes), then the *service reservation* [15] for the end-to-end QoS provision is taken place.

During the run-time instantiation, if the available system resources fluctuate, or the user who is the owner of the running application moves, or changes his/her QoS level preference, the service polymorphism will select the next best configurations among the previously returned configurations, and the resting steps repeat. Hence, *QoSCSpec* helps the run-time system to reduce the overhead of run-time instantiation, to ensure the end-to-end QoS provision, and to adapt appropriately. While we propose the run-time instantiation as described, the QoSCSpec is generic, expressive and independent. Different run-time instantiation mechanisms can deploy the compiled information (QoSCSpec) in their systems flexibly.

3 APPLICATION QoS MODEL

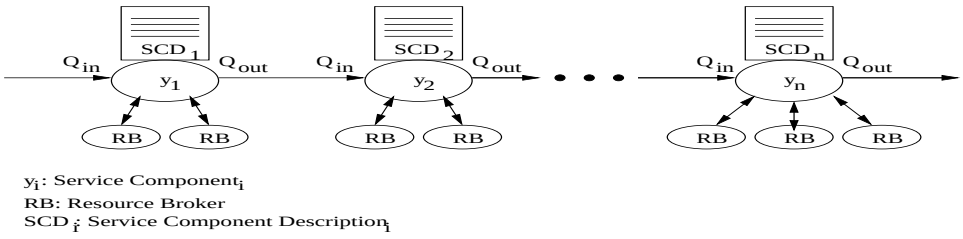

Figure 3: Task-flow Model

The application QoS model in our $2K^Q$ QoS Service architecture is based on the task-flow

model. The task-flow consists of sequences of service components y_i comprising the application's functional graph, where each service component is described by a service component description (SCD_i), and consists of input quality vector (Q_{in}), internal supporting quality vector (Q_{int}), and output quality vector (Q_{out})[2]. A service component's Q_{out} can be described by the same service component's Q_{in} and Q_{int} as follows:

$$Q_{out}^{y_i} = f(Q_{in}^{y_i}, Q_{int}^{y_i}) \tag{1}$$

Each service component requires multiple resources allocated by resource brokers (RBs) as shown in Figure 3. After introducing the $2K^Q$ QoS Service from the architectural point of view. In the following section, we describe the system design of the $2K^Q$ QoS Service.

4 SYSTEM DESIGN

The $2K^Q$ QoS Service system design as shown in Figure 4 is based on the distributed QoS compilation and the run-time instantiation phases described in the overall architecture. The QoS editor and the distributed QoS compiler implement the distributed compilation phase. The run-time $2K^Q$ system implements the run-time instantiation phase with additional support for the resource-aware QoS configuration translation. In the following, we describe the main system components of the $2K^Q$ QoS Service in details.

4.1 QoS Editor and Distributed QoS Compiler

4.1.1 QoS Editor

QoS editor provides a programming environment for an application developer to develop a QoS-aware application. The programming environment consists of different graphic user interfaces which allow the application developer to specify different QoS specifications such as the hierarchical functional graphs, the service component description, the application description and the user to application-specific template. The detailed design of the QoS editor can be found in [12].

[2] Q_{in}, Q_{int}, and Q_{out} are vectors of application level QoS parameters.

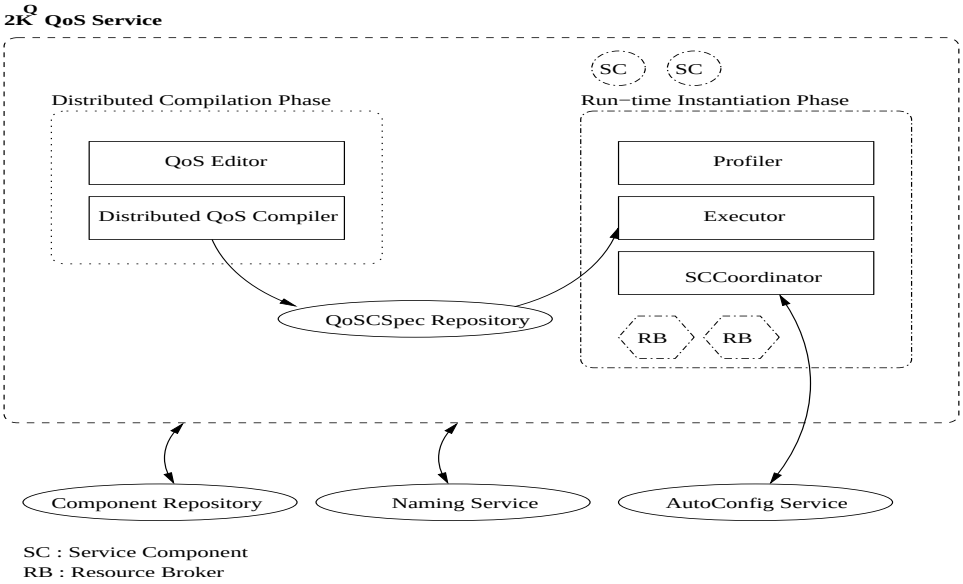

Figure 4: $2K^Q$ QoS Service System Design

4.1.2 Distributed QoS Compiler

Distributed QoS compiler implements three translators: the *symbolic QoS configuration translator* (QoS compiler phase I), the *pre-requisite system component translator* (QoS compiler phase II), and the *resource-aware QoS configuration translator* (QoS compiler phase III). The inputs of the symbolic QoS configuration translator are the QoS specifications specified via the QoS editor. The inputs of the pre-requisite system component translator are symbolic QoS configurations which are the compiled results from QoS compiler phase I. The inputs of the resource-aware QoS configuration translator are the symbolic QoS configurations associated with their appropriate system components and specific QoS parameters which are the compiled results from QoS compiler phase II. The results of the resource-aware QoS configuration translator are the mappings of compiled results (acquired from QoS phase II) and their distributed system resource requirements. The *QoSCSpec generator* generates a QoSCSpec for an application consisting of the results from the QoS compiler phase III associated with the application description and service components descriptions.

4.1.3 QoSCSpec Repository

QoSCSpec repository consists of all compiled results from the distributed compilation phase which can be retrieved by the run-time instantiation phase. The QoSCSpec repository can be

considered as a service which allows the distributed QoS compiler to register a new QoSCSpec, allows the run-time instantiation to retrieve a QoSCSpec with some constraints, and allows a web browser to retrieve the available QoSCSpec (all, or some) to display on the browser.

4.2 Run-time $2K^Q$

4.2.1 Profiler

Profiler is an agent of the run-time system which performs the distributed resource profiling, based on the resource-aware QoS configuration translation, on behalf of the distributed QoS compiler. The Profiler implements the distributed profiling protocol described in [16]. It disseminates service components in a configuration and their associated system components into the distributed nodes, instantiates the distributed service components, starts the profiling, gathers the distributed profiling results, and returns them to the distributed QoS compiler.

4.2.2 Executor

Executor is an agent of the run-time system which performs the distributed instantiation of an application's configuration corresponding to a user request with specific QoS requirements. The application's configuration is selected by the Executor based on the static information suggested in the QoSCSpec and the dynamic information such as the current resource availability of distributed nodes in the system. Also, the Executor is responsible to the functional adaptation such as service component reconfiguration if the QoS violation occurs and cannot be reconciled by the data adaptation in the resource brokers.

4.2.3 SCCoordinator

SCCoordinator is the helper agent of the Profiler and the Executor. It helps the Profiler and Executor to manage the dynamic downloading, the instantiation, the service component reconfiguration among the distributed nodes systematically. The SCCoordinator is required to be in all distributed nodes supporting the $2K^Q$ QoS Service. The Executor and the SCCoordinator implement the quality-aware service management in the overall architecture of the $2K^Q$ QoS Service. The Profiler is the added agent supporting the distributed profiling mechanism required by the

distributed QoS compiler.

4.2.4 Resource Brokers

Resource brokers are responsible to resource-specific probing, admission, negotiation, reservation, allocation, enforcement and data adaptation. A resource broker is not needed to support the reservation model. Nevertheless, if an application requires the deterministic end-to-end QoS guarantees the resource brokers along the instantiation path are required to support the reservation model. In our $2K^Q$ QoS Service, we deploy DSRT [2] as our CPU broker. RSVP is an alternative for the bandwidth broker which supports the bandwidth reservation.

4.3 Service Components

A **service component** implements a specific service component, which can be a single functional unit, or a composite functional units, specified in the application functional graph.

Note that the Profiler, the Executor, the SCCoordinator, the Resource Brokers and the Service Components are running on distributed nodes.

4.4 Dependencies

We deployed the Naming Service as part of the resource-aware QoS configuration translation, and deployed AutoConfig Service, Naming Service, and Component Repository as part of our run-time $2K^Q$ implementation.

In the following section, we describe the $2K^Q$ QoS Service from the implementation point of view.

5 IDL INTERFACES AND IMPLEMENTATIONS

5.1 QoS Editor and Distributed QoS Compiler

The QoS editor and the distributed QoS compiler are implemented in a java package named QoSTalk. The distributed QoS compiler is the internal engine of the QoS editor. Hence, all distributed QoS compiler's interfaces are called by the QoS editor. The QoS editor is considered as

the shell of the QoS compiler which abstracts away the internal distributed QoS compiler's interfaces from an application developer and provides the high-level QoS specifications comprising of a QoS-aware application via different GUIs. The detailed information of the QoS editor can be found in [12]. The resource-aware QoS configuration translation contacts the Profiler via the CORBA mapping to java, based on idltojava for win32.

5.2 Run-time $2K^Q$

The Profiler, Executor, and the SCCoordinator are implemented as CORBA components using C++. The available version is running on SunOS 5.7. *Note: The following IDL interfaces are based on Gaia active space. They are modified from the previous version which are not defined in the Gaia module and are not based on the UOB (Unified Object Bus). The following interfaces are not fully implemented. We are in the status of porting our running version on Solaris without UOB to this version running on windows 2000 based on UOB and Gaia active space.*

5.2.1 Profiler IDL interface

```
module Gaia{
  module QoSService {
    module Runtime2KQ {
      interface Profiler : UOB::CORBAComponent{
        QoSService::Runtime2KQ::SCCoordinator::RequestId
            start_binding(in QoSService::Runtime2KQ::
                SCCoordinator::SCsInfo scs_info) ;
        boolean start_probing(in QoSService::Runtime2KQ::
                SCCoordinator::RequestId prid) ;
        boolean end_probing(in QoSService::Runtime2KQ::
                SCCoordinator::RequestId prid) ;
        boolean get_probing_result(in QoSService::Runtime2KQ::
                SCCoordinator::RequestId prid,
                inout QoSService::Runtime2KQ::
                SCCoordinator::ResourceMappings rmaps_out) ;
```

```
      } ; // end Profiler interface
    } ; // end Runtime2KQ module
  } ; // end QoSService module
} ; // end Gaia module
```

5.2.2 Executor IDL interface

```
module Gaia {
  module QoSService {
    module Runtime2KQ {
      interface Executor : UOB::CORBAComponent {
        QoSService::Runtime2KQ::SCCoordinator::RequestId
            request_application(in QoSService::Runtime2KQ::
            SCCoordinator::SCsInfo scs_info,
            in boolean is_fully_defined) ;
        boolean start_application(in QoSService::Runtime2KQ::
            SCCoordinator::RequestId requestId) ;
        boolean stop_application(in QoSService::Runtime2KQ::
            SCCoordinator::RequestId requestId) ;
        boolean adapt_application(in QoSService::Runtime2KQ::
            SCCoordinator::RequestId requestId) ;
      } ; // end Executor interface
    } ; // end Runtime2KQ module
  } ; // end QoSService module
} ; // end Gaia module
```

5.2.3 SCCoordinator IDL interface

```
module Gaia {
  module QoSService {
    module Runtime2KQ {
      interface SCCoordinator : Gaia::UOB::CORBAComponent {
```

```
struct a_qos_param {
  string type ;
  string value ;
} ;
typedef sequence<a_qos_param> QoSParams ;
struct sc_info {
  string node_type ;
  string node_name ;
  string node_category ;
  short node_label ;
  string source_address ;
  string target_address ;
  string hardware_requirement ;
  string software_requirement ;
  string reservationType ;
  long period ;
  long peakProcessingTime ;
  long sustainableProcessingTime ;
  long burstolerance ;
  short memory ;
  short disk ;
  short bandwidth ;
  // QoSParams qos_params ;
} ;
typedef sequence<sc_info> SCsInfo ;
struct resource_mapping {
  string node_name ;
  string reservationType ;
  long period ;
  long peakProcessingTime ;
```

```
  long sustainableProcessingTime ;
  long burstolerance ;
  short memory ;
  short disk ;
  short bandwidth ;
} ;
typedef sequence<resource_mapping> ResourceMappings ;
typedef unsigned short RequestId ;
/**
  Methods for distributed SCCoordinator interactions
  (Distributed Probing Service)
 */
boolean start_component_probing(in RequestId prid,
  in string sc_name, in QoSParams qos_params) ;
boolean stop_component_probing(in RequestId prid,
  in string sc_name) ;
boolean get_component_probing_result(in RequestId prid,
  in string sc_name, inout ResourceMappings rmap_out) ;
/**
  Methods for distributed SCCoordinator interactions
  (Runtime Instantiation)
 */
boolean start_component(in RequestId rid,
    in string sc_name, in QoSParams qos_params) ;
boolean stop_component(in RequestId rid, in string sc_name) ;
boolean adapt_component(in RequestId rid, in string sc_name) ;
/**
    Sharing method(s) for both the distributed probing and the
    runtime instantiation
*/
```

```
        boolean allocate_component(in RequestId rid) ;
      } ; // end SCCoordinator interface
    } ; // end Runtime2KQ module
  } ; // end QoSService module
} ; // end Gaia module
```

5.2.4 Resource Broker interfaces

Currently, we deploy only DSRT [2] as the CPU Broker. DSRT provides its own APIs which can be found in [17]. Note that an implementation of a specific service component uses the service of CPU broker directly via the DSRT's APIs.

5.3 Service Components

To allow the $2K^Q$ QoS Service to manage service components systematically, we propose a service component IDL interface which is used as the common IDL interface for all service components implement a specific service component in an application functional graph. The *ServiceComponent* IDL interface is presented below.

```
module Gaia {
  module QoSService {
    module ServiceComponent {
      interface ServiceComponent : Gaia::UOB::CORBAComponent {
        boolean start(in QoSService::Runtime2KQ::
                SCCoordinator::ResourceMappings rmap) ;
        boolean stop() ;
        string get_probing_result() ;
      } ; // end ServiceComponent interface
    } ; // end ServiceComponent module
  } ; // end QoSService module
} ; // end Gaia module
```

6 $2K^Q$ QoS SERVICE COMPILATION AND EXECUTION

In this section, we describe how to compile and run the $2K^Q$ QoS Service in the distributed nodes. Both the compilation and the execution parts are divided into two sections the QoS editor with the distributed QoS compiler, and the run-time $2K^Q$.

6.1 Compilation

6.1.1 QoS Editor and Distributed QoS compiler

Before compiling, we have to provide the following environment:

1. Need a JDK version 1.2.2 or higher;
2. Set the CLASSPATH to include the directory that contains the QoSTalk package, and the directory of QoSTalk itself. For instance, if we install the QoSTalk in directory 2k, then the CLASSPATH will be included the directory 2k and the directory 2k/QoSTalk;
3. Compile the QoSTalk package by going into the QoSTalk directory, then, call *javac QoSTalk.QoSTalking*.

6.1.2 Run-time $2K^Q$

Currently, we have only the UNIX platform of the run-time $2K^Q$[3]. To compile the run-time $2K^Q$, go into the *2KQ* directory under ACE_wrappers and 2k. Then, change the directory into each individual run-time $2K^Q$ components such as the SCCoordinator, the Profiler, and the Executor. Call *gmake* in these individual directories.

6.2 Execution

6.2.1 QoS Editor and Distributed QoS compiler

The executable of the QoS editor and the distributed QoS compiler demo version is in hawaii.cs.uiuc.edu. To start, at the command prompt call *java QoSTalk.QoSTalking*

[3]Note that we are in the process of porting to windows platform with the underlying modification to be compatible with the Gaia active space, and the Unified Object Bus architecture.

6.2.2 Run-time $2K^Q$

The executables of the run-time $2K^Q$'s demo version are in Nahrstedt domain in the directory /home/klara/share/DEMO/2KQ/2KQ2.1. In the following, we describe how to start the run-time $2K^Q$ step by step.

1. Change the directory (cd) to the above demo version directory by calling
   cd /home/klara/share/DEMO/2KQ/2KQ2.1;
2. Start a Naming Service on a distributed node, for instance, paris.cs.uiuc.edu by calling
   Naming_Service;
3. Start a component repository on the same node as of Naming Service by calling
   ComponentRepository;
4. Start the SCCoordinator on all distributed nodes supporting $2K^Q$ QoS Service by calling
   SCCoordinatorServer;
5. Start the AutoConfig Service on all distributed nodes supporting $2K^Q$ QoS Service by calling
   AutoConfigServer;
6. Start the Profiler on paris.cs.uiuc.edu by calling
   ProfilerServer;
7. Start the Executor on paris.cs.uiuc.edu by calling
   ExecutorServer;
   Note that the distributed QoS compiler currently assumes that the Profiler and the Executor are statically located on a specific machine, paris.cs.uiuc.edu. Step 1 and Step 2 will be skipped if the system already has a Naming Service and a component repository instances running.

Figure 5 represents the example running $2K^Q$ QoS Service in a real system. From Figure 5, there are two phases:

Distributed Compilation Phase:

1. The distributed QoS compiler contacts the naming service to get the object reference of the Profiler (A);

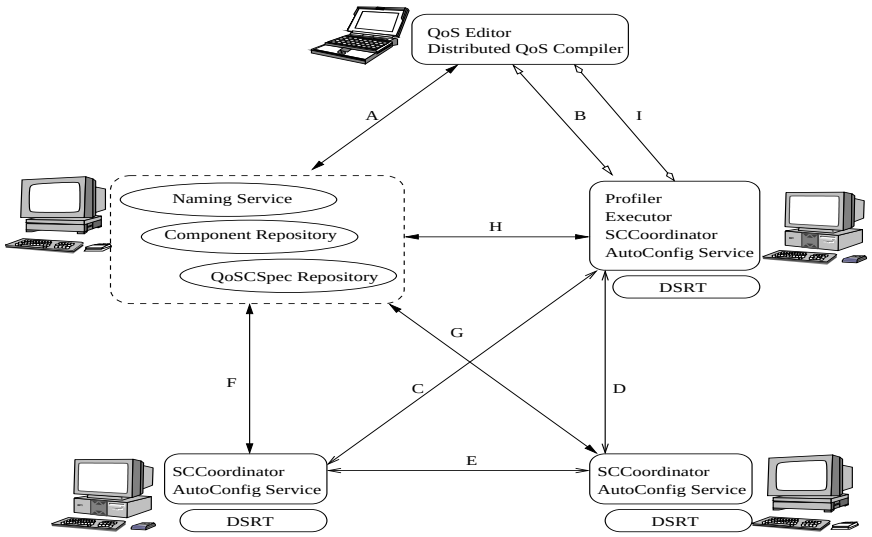

Figure 5: Running $2K^Q$ QoS Service in a Distributed Component-Based System (Example)

2. The distributed QoS compiler contacts the Profiler to collaborate the distributed profiling (B);

3. The Profiler collaborates the distributed profiling on behalf of the distributed QoS compiler (C, D, E);

4. The Profiler returns the distributed profiling results to the distributed QoS compiler (B);

5. The distributed QoS compiler generates and registers the QoSCSpec to the QoSCSpec repository (A).

Runtime Instantiation Phase:

1. A run-time instantiator contacts the naming service to get the object reference of the Executor (A);

2. The run-time instantiator contacts the Executor to collaborate the distributed run-time instantiation (I);

3. The Executor collaborates the distributed run-time instantiation of behalf of the run-time instantiator (C, D, E);

4. The Executor returns the instantiation result to the run-time instantiator (I).

Note:

- Arrows F, G, H represent the interactions of supporting $2K^Q$ QoS Service nodes to the naming service, the component repository and the QoSCSpec repository as part of both distributed QoS compilation and run-time instantiation phases.
- While in the example has only the Profiler and the Executor running only on a machine, in general, the Profiler, and the Executor can run on every machine supporting the $2K^Q$ QoS Service.
- In the current implementation, the run-time instantiator selects an appropriate configuration for an application request on behalf of the Executor (*)

7 $2K^Q$ QoS SERVICE IN ACTIVE SPACES

In this section, we describe how the $2K^Q$ QoS Service can be deployed in active spaces corresponding to different active space definitions.

- **Active Space Definition 1 (Gaia): Active space (as shown in Figure 6) is associated with a physical space and bootstrapped once. The associated services are always running. A user can interact with the active space using the available device connected to a specific underlying system such as UOB.**

  In this definition, The $2K^Q$ QoS Service (the run-time $2K^Q$ part) is considered as an associated service which will be bootstrapped once in an active space. The SCCoordinator, the AutoConfig Service, and the resource brokers will be bootstrapped on all distributed machines supporting quality of service in the active space. The Profiler and the Executor will be bootstrapped at least on a machine in the active space.

  A QoS-aware application running in the active space is developed by an application developer via the QoS editor and the distributed QoS compiler running on a machine accessible to the active space. The compilation result (QoSCSpec) is in the QoSCSpec repository which is accessible from the active space. Two possible associations of the developed QoS-aware application are described in the following.

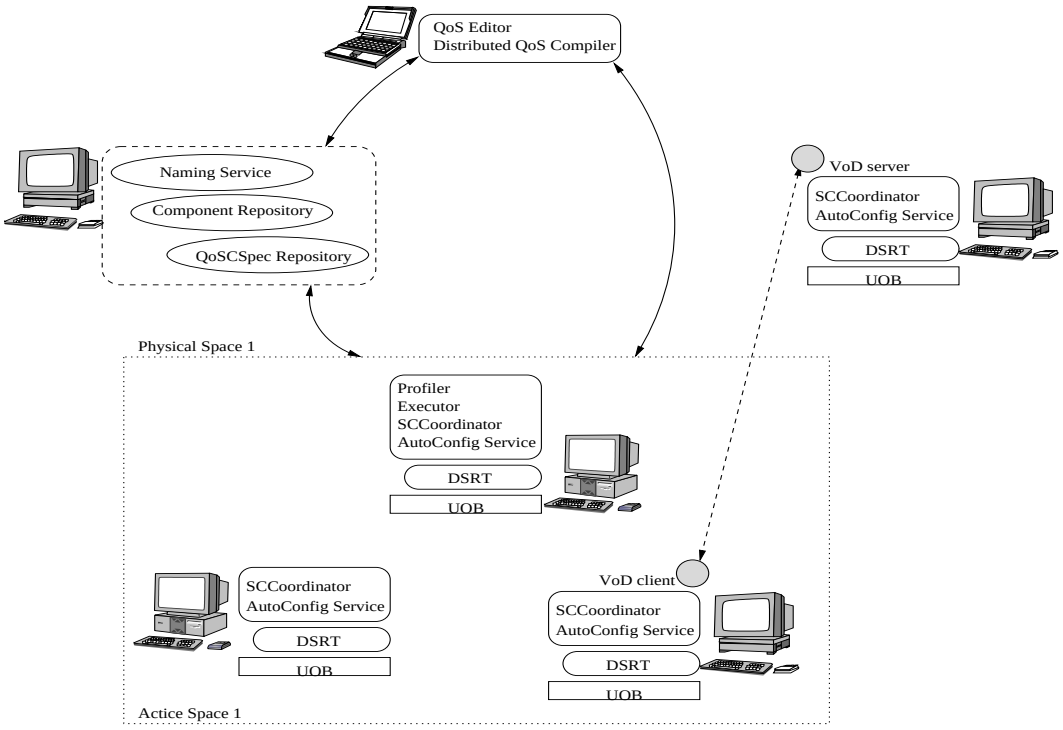

Figure 6: $2K^Q$ QoS Service in Active Spaces Definition 1 (Gaia)

1. The developed QoS-aware application is associated with the active space, and bootstrapped once with the run-time $2K^Q$. Every user entering the active space shares the same configurations of running QoS-aware application, and controls the application via any device connected to a specific underlying mechanism such as UOB. (Modification: the QoSCSpec's application description of a developed application will consist of one more field, the active space ID.)

2. The developed QoS-aware application is associated not only with the active space but also with individual user's profile of preference of the active space. In this case, the application is not bootstrapped once at the starting step of the active space, but it will be instantiated dynamically in the active space corresponding to the detection of a specific user's presence. The application developer programs a QoS-aware application on behalf of an individual user. (Modification: the QoSCSpec's application description of a developed application will consist of two more fields, the active space ID., and the user ID.)

Note that some components comprising the developed QoS-aware application are not required

to be in the active space. This means they can be in other active spaces or in an unbound space. For example, a video on demand server can be in an unbound space while its clients can be distributed in different active spaces.

The Naming Service, the Component repository, the QoSCSpec repository and some running service components can be considered as being in an unbound space which is accessible from the active space.

- **Active Space Definition 2: Active space is NOT associated with a physical space. An active space is a virtual space which is created, and owned by an individual user or a group of user. The active space in this definition is considered mainly from a user point of view with his/her targeting required applications. Hence, the active space is represented by distributed running application processes. The user can interact (e.g. create, manage, remove applications, add add applications) to his/her active space via an active space GUI.**

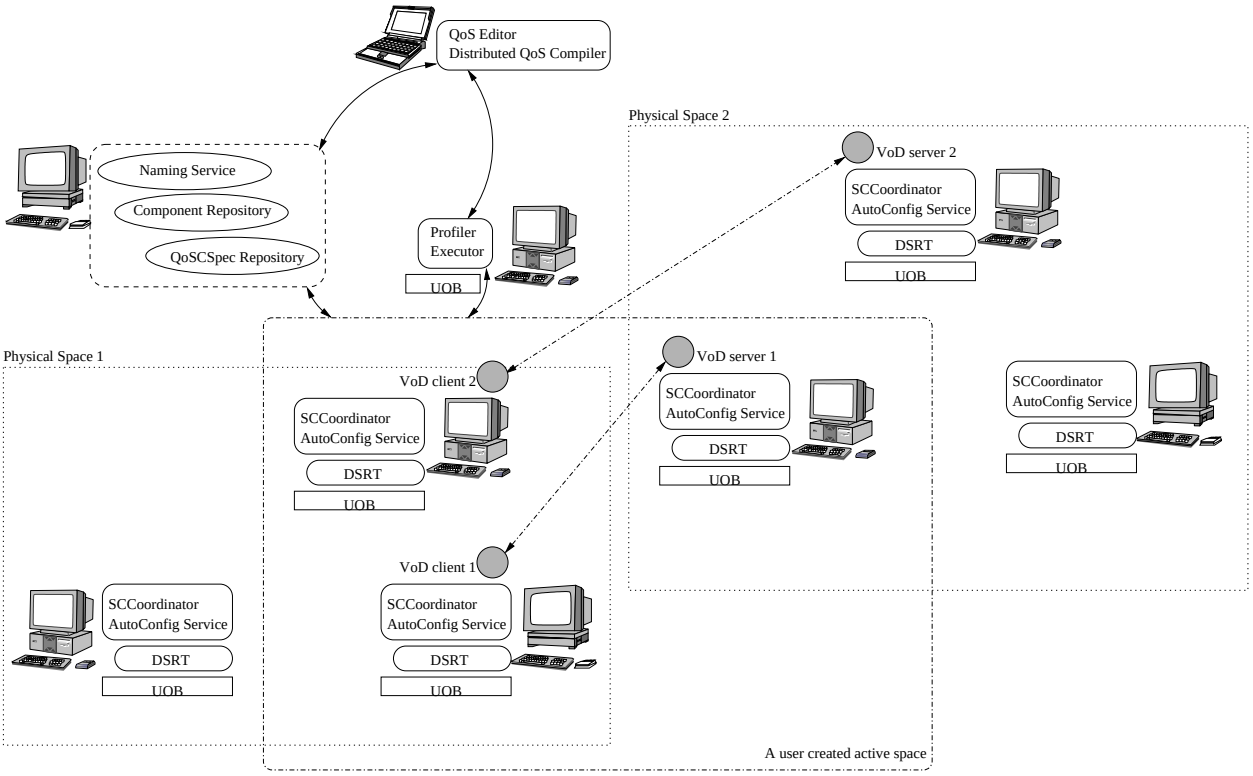

Figure 7: $2K^Q$ QoS Service in Active Spaces Definition 2

In this definition, the active space as shown in Figure 7 still needs a bootstrapped process

similar to the bootstrapped process defined in Gaia active space. However, the bootstrapped process in associated with an individual user created active space. In this scenario, The $2K^Q$ QoS Service (the run-time $2K^Q$ part) is still considered as an associated service in the bootstrapped process. A QoS-aware application running in this active space; however, is developed by an individual user via the QoS editor and the distributed QoS compiler running on a machine which can be an unbound space or in his/her created active space. The compilation result (QoSCSpec) is in the QoSCSpec repository in an unbound space which is accessible from the created active space or can be a private repository associated with the created active space.

In this scenario, the QoSCSpec's application description of a developed application will consist of two more fields, the active space ID., and the user ID.)

Note that some service components comprising the developed QoS-aware application are not required to be in the user created active space. This means they can be in other active spaces or in an unbound space. For example, a video on demand server can be in an unbound space while its clients can be in the user created active space.

The Naming Service, the Component repository, and some running service components can be considered as being in an unbound space which is accessible from the active space.

8 RELATED WORK

Figure 8 represents the comparisons among QoS middleware architectures. In the following, we describe the metrics used in the comparison table.

QoS specification is considered from the application developer point of view. QoS specifications in different QoS middleware architectures are proposed differently corresponding to the architecture's objectives and its underlying supporting QoS mechanisms such as QoS translation, QoS enforcement, and QoS adaptation.

QoS translation includes different types of translations such as a translation from QoS specification into a set of different compositions of specific service components representing a distributed QoS-aware application, a translation of application QoS parameters into the association of specific system components and their QoS parameters, and a translation of application QoS parameters

into system resource requirements. From the comparison table, almost all QoS middleware architectures only provide the interfaces or mechanisms to specify the application QoS parameters or the system QoS parameters directly. How to specify these QoS parameters especially in the system level appropriately, corresponding to different QoS requirements; however, depends on the application developer's decision. The multi-phase translations in our $2K^Q$ QoS Service are proposed to help the application developer in this aspect. It translates the application QoS parameters into the association of system components, their suitable QoS parameters, and the overall resource requirements systematically.

Range of supporting applications represents which application domains a QoS middleware architecture is targeting on. From the comparison table, almost all QoS middleware architectures are tailored toward a specific application domain. In Q-RAM [11] and QoSME [5], even though they do not specify a specific application domain, they are mainly dealing with the management of underlying resources. This is not enough for supporting a broader range of QoS-aware applications which may have application QoS parameters which can not be translated directly to the underlying system resource requirements. In $2K^Q$ QoS Service , we support a broader range of applications' QoS provisions based on the customizable QoS specifications and multi-phase translations.

QoS enforcement demonstrates how a QoS middleware architecture enforces QoS provision. In $2K^Q$ QoS Service, QoS enforcement is based on minimum resource reservation model if it is applicable. Note that the resource reservation model in $2K^Q$ will be applied after the multi-phase translations which handle the QoS provisions of application QoS parameters which can not be translated directly into the system resource requirements. From the comparison table, QoS services in CORBA do not have any concrete QoS enforcement except the real-time CORBA.

QoS adaptation represents how a QoS middleware architecture adapts corresponding to different incoming events such as QoS violation, changing of execution environment, changing of user preference, and user mobility. In this aspect, none of QoS services in CORBA except the audio/video streaming service [7] provides a formal QoS adaptation mechanism. In Agilos [10], QoS adaptation is based on control-based theory. In $2K^Q$ QoS Service, QoS adaptation is based on data adaptation in the QoS-aware resource management layer, and based on functional adaptation in the QoS-aware service management layer.

9 INTEGRATION AND IMPLEMENTATION NEEDED TO BE DONE

1. *Integration*: Modify all codes related to user presence detection to use the infrastructure of Gaia user presence detection, based on the event service. The main code needed to be modified is in the mobile ID application. CORBArize the mobile ID , the voice mail and the remote control applications based on the CORBA component template defined in the UOB.

2. *Integration*: Study the DOS (Data Object Service, by Christophser Hess) and see how it could be used or integrated with our applications. The DOS is considered as the file system in the smart space. The DOS at least provides different containers which handle different types of input data. For example, a container transcodes the MPEG-1 stream to the bitmap stream. Because we have had some transcoders and may implement some more, the implementation should follow the specification of the data object service.

3. *Implementation*: Implement a QoSCSpec repository as a CORBA service. This repository should at least allows the distributed QoS compiler to register a new QoSCSpec, allows the run-time instantiation to retrieve a QoSCSpec with some constraints, and allows a web browser to retrieve the available QoSCSpec (all, or some) to display on the browser. The current constraint may include only the application name, smart space id, and user id. In the next phase, the resource constraints and the other execution environment constraints will be included and the constraint satisfaction engine may be deployed.

4. *Implementation*: Implement the user to application-specific translation template, and extend the distributed QoS compiler with the fully implemented first and second phases of translations, including the generator of QoSCSpec

5. *Implementation*: Implement a resource monitoring service which includes distributed CPU resource, and network bandwidth among two nodes. (Tomonori had done with the DSRT reporting of the current CPU to the trader service.) The resource monitoring service allows the retrieving information of available CPU in a distributed node, and the available network bandwidth among two specified nodes.

6. *Implementation*: Implement the code instrumentation GUI as part of the QoS Editor which allows the application developer to insert the DSRT APIs into a service component source code, to compile and link the instrumented source code and to install this instrumented object code into a location known by the QoS compiler. The QoS compiler uses this code during the distributed resource translation. Note that the source code of a service component can be on distributed nodes.

References

[1] 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*, 7, 1998.

[2] H. Chu and K. Nahrstedt. CPU Service Classes for Multimedia Applications. *Proceedings of IEEE International Conference on Multimedia Computing and Systems (IEEE ICMCS '99)*, June 1999.

[3] J. Zinky, D. Bakken, and R. Schantz. Architecture Support for Quality of Service for CORBA Objects . *Theory and Practice of Object Systems*, January 1997.

[4] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson. QoS Aspect Languages and Their Runtime Integration. *Lecture Notes in Computer Science, Springer-Verlag. Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, 1511, May 1998.

[5] Patricia Gomes Soares Florissi. QoSME: QoS Management Environment. *Ph.D. Thesis, Columbia University.*

[6] BEA Systems Inc., Expersoft Corporation, Imprise Corporation, International Business Machine Corporation, International Computers Ltd., IONA Technologies Plc., Northern Telecom Corpoaration, Novell Inc., Oracle Corporation, Peerlogic Inc., and TIBCO Inc. CORBA Messaging. *online documentation at http://www.omg.org/cgi-bin/doc?orbos/98-05-05.*, May 1998.

[7] IONA Technologies Plc., Lucent Technologies Inc., and AG Siemens-Nixdorf. Control and Management of Audio/Video Streams OMG RFP Submission. *online documentation at http://www.omg.org/docs/telecom/98-10-5.doc*, May 1998.

[8] Ericsson, Eternal Systems Inc., HighComm, Inprise Corporation, IONA Technologies Plc., Lockheed Martin Corporation, Lucent Technologies, Objective Interface Systems Inc., Oracle Corporation, and Sun Microsystems Inc. Fault Tolerant CORBA, Joint Revised Submission. *online documentation at http://www.omg.org/techprocess/meetings/schedule/Fault_Tolerance_RFP.html*, December 1999.

[9] Alcatel, Hewlett-Packard Company, Highlander Communications L.C., Inprise Corporation, IONA Technologies, Lockheed Martin Federal systems Inc., Lucent Technologies Inc., Nortel Networks, Objective Interface Systems Inc., Object-Oriented Concepts Inc., Sun Microsystems Inc., and Tri-Pacific Software Inc. Real-Time CORBA, Joint Revised Submission. *online documentation at http://www.omg.org/cgi-bin/doc?orbos/99-02-12*, March 1999.

[10] B. Li and K. Nahrstedt. A Control-based Middleware Framework for Quality of Service Adaptations. *IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms*, September 1999.

[11] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. *In Proceedings of the IEEE Real-Time Systems Symposium*, 1997.

[12] X. Gu and K. Nahrstedt. Visual Quality of Service Programming for Distributed Heterogeneous Systems. *Technical Report UIUCDCS-R-2000-2190 UILU-ENG-2000-1746, Department of Computer Science, University of Illinois at Urbana-Champaign, Nov.2000(submitted for ACM SIGPLAN 2001).*

[13] Dongyan Xu and Klara Nahrstedt. Supporting Multimedia Service Polymorphism in Dynamic and Heterogeneous Environments. *Technical Report UIUCDCS-R-2000-2159, Department of Computer Science, University of Illinois at Urbana-Champaign, (submitted for journal publication)*, October 2000.

[14] D. Xu, K. Nahrstedt, and D. Wichadakul. MeGaDiP: A Wide-Area Media Gateway Discovery Protocol. *19th IEEE International Performance, Computing, and Communications Conference (IPCCC 2000)*, February 2000.

[15] D. Xu, D. Wichadakul, and K. Nahrstedt. A Resource-Aware Middleware for Active and Configurable Distributed Services. *to appear in Proceedings of the 2th NSF Workshop on Active Middleware Services*, August 2000.

[16] D. Wichadakul and K. Nahrstedt. Distributed QoS Compiler.

[17] http://cairo.cs.uiuc.edu/software/DSRT 2/scheduler.html. DSRT 2.0. November 2000.

QoS Middleware Architecture	QoS Specification	QoS Translation	Range of Supporting Applications	QoS Enforcement	QoS Adaptation
$2K^Q$ QoS Service	QoS specifications comprising the QoS−aware application programming environment	Multi−Phase Translations	Different domains of Applications via customizable QoS specifications and Multi−Phase translations	Based on minimum resource reservation model	Data adaptation and Fucntional adaptation
QoS in CORBA					
Audio/Video Streaming Service	Pre−defined IDL interfaces extended from standard CORBA IDL interfaces	Internal translation	Audio/Video streaming applications	Based on available transport protocols	Modified QoS of transport protocol
QoS Messaging Service	Pre−defined IDL interfaces extended from standard CORBA IDL interfaces	N/A	Messaging applications	Queue order in a Router process. This is applied only with the specific router type.	N/A
Real−time CORBA	Pre−defined IDL interfaces extended from standard CORBA IDL interfaces	N/A	Real−time applications	Based on the real−time extension in standard operating system	N/A
Fault Tolerance CORBA	Pre−defined IDL interfaces extended from standard CORBA IDL interfaces	N/A	Fault tolerance applications	Number of replications needed to be maintained from the implementation point of view.	N/A
TAO	Pre−defined IDL interfaces	N/A	Real−time messaging applications	Based on priority queues in ORB.	N/A
QuO	A set of Quality Description Language (QDLs)	N/A	Messaging applications	Depending on individual application−specific implementation and specification via QDLs	Depending on individual application−specific implementation and specification via QDLs
QoSME	QuAL (Quality Assurance Language)	N/A	Any applications needing QoS in transport protocols and QoS in OS	Based on available transport protocols and available OS compiled with POSIX standard	Via a set of operators for QoS re−negotiation during the run−time
Hafid's and Bochmann's QoS management in distributed multimefdia applications	N/A	N/A	Distributed multimedia applications	Based on QoS nego−tiation protocol and resource allocation	Via a renegotiation to find another suitable configuration
Q−RAM	Utility functions	N/A Assume the availability of resource consumption functions	Any applications running in the sharing resource environment and needing QoS provision	Based on resource allocation model	Via the repeat of resource allocation algorithm
Agilos	Rule−based and membership functions	Internal analytical translation with the help of QualProbe during the run−time	Any applications which is complied with control based QoS adaptation	Based on best−effort with the control−based application−specific adaptations	Via a control−based adaptation

Figure 8: Comparisons of QoS Middleware Architectures