$2K^{Q+}$: An Integrated Approach of QoS Compilation and Reconfigurable, Component-Based Run-Time Middleware for the Unified QoS Management Framework*

Duangdao Wichadakul, Klara Nahrstedt, Xiaohui Gu, Dongyan Xu

Department of Computer Science
University of Illinois at Urbana-Champaign
{wichadak, klara, xgu, d-xu}@cs.uiuc.edu

Abstract. Different distributed component-based applications (e.g., distributed multimedia, library information retrieval, secure stock trading applications), running in heterogeneous execution environments, need different quality of service (QoS). The semantics of QoS requirements and their provisions are application-specific, and they vary among different application domains. Furthermore, QoS provisions vary per applications in heterogeneous execution environments due to the varying distributed resource availability. Making these applications QoS-aware during the development phase, and ensuring their QoS guarantees during the execution phase is complex and hard.
In this paper, we present a unified QoS management framework, called $2K^{Q+}$. This framework extends our existing run-time $2K^Q$ middleware system [1] by including our uniform QoS programming environment and our automated QoS compilation system (*Q-Compiler*). The uniform QoS programming and its corresponding QoS compilation allow and assist the application developer to build different component-based domain applications in QoS-aware fashion. Furthermore, this novel programming and compilation environment enables the applications to be instantiated, managed, and controlled by the same reconfigurable, component-based run-time middleware, such as $2K^Q$, in heterogeneous environments.
Our experimental results show that different QoS-aware applications, using the $2K^{Q+}$ framework, get configured and setup fast and efficiently.

1 Introduction

The rapid growth of distributed component-based environments and coexistence of different application domains, such as multimedia and electronic commerce, present significant challenges to the provision of different applications' Quality

* This work was supported by the National Science Foundation under contract numbers NSF EIA 9870736, NSF CCR-9988199, the Air Force Grant under contract number F30602-97-2-0121, NSF CISE Infrastructure grant under contract number NSF EIA 99-72884, and NASA grant under contract number NASA NAG 2-1250.

of Service (QoS). First, different domain applications have specific semantics for QoS requirements and provisions. For example, multimedia and library information retrieval applications are concerned about the service qualities of audio, video streaming (e.g., frame rate, frame size, sampling rate, end-to-end delay) and messaging (e.g., priority, response time, reliability), respectively. Developing an application to be QoS-aware during the application development cycle, and ensuring its QoS provisions during the application's run-time are non-trivial tasks for a specific application and even harder for different applications. Second, with the pervasive computing, these applications are expected to be executed in heterogeneous computing and communication environments with different capacities of processing power (e.g., high performance PC, PDAs), battery power, and network bandwidth (e.g., wired, wireless networks). The big challenging problem, as shown in Fig. 1, is:

"How should a unified QoS management framework look like to allow different distributed component-based applications to use it and achieve required guarantees in heterogeneous, dynamic computing and communication environments ?"

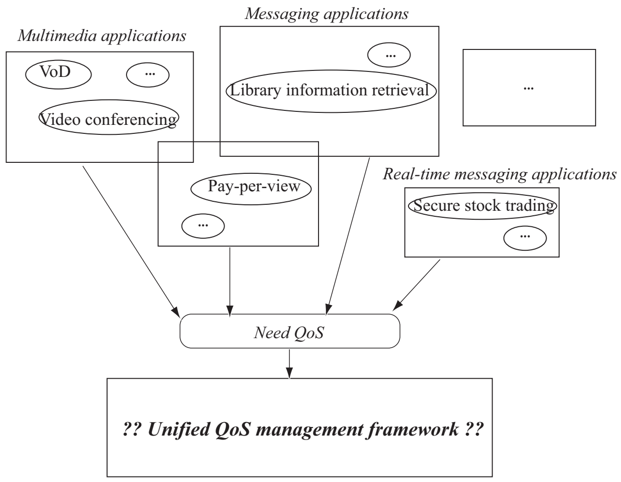


Fig. 1. Challenging Problem

Several QoS architectures and approaches [2, 3, 4, 5, 6, 7] already exist. However, all of them are designed only for one domain of applications, particularly for multimedia applications, or designed to handle particular aspect of QoS provisions such as the QoS-aware resource management [8]. If one wants to implement and run a different domain of applications, the existing tailored architectures won't be applicable or reusable. In the distributed object computing (DOC) middleware such as CORBA, some QoS support [9, 10, 11, 12] has been proposed. In the CORBA case, an application will be QoS-aware if the application developer deploys QoS-related interfaces of requested QoS services. Hence, an application developer has to learn (a) different IDL interfaces for different types of QoS, (b) the semantics of these interfaces, and (c) how to translate his/her application QoS requirements into these interfaces and their parameters appropriately. TAO project [13] developed the optimized CORBA ORB which supports the real-time messaging. QuO project [14] allows an application developer to develop distributed applications that can adapt to the changing quality of service in CORBA environment. The QuO considers the quality of service in the broader domain including (1) quality of object interaction such as reliability, (2) real-time method invocation, and (3) security. QoS in distributed object computing middleware is also tailored towards specific domain of applications or particular aspect of QoS provisions. Hence, no work fully solves the above challenging problem.

In this paper, we present the $2K^{Q+}$ framework, shown in Fig. 2), a unified QoS management framework, which allows and assists the application developer to develop different component-based domain applications in QoS-aware fashions. There different domain applications are able to be developed in the *same* programming and compilation environment, and to be instantiated, managed and controlled uniformly by the *same* reconfigurable, run-time middleware. This means that the unification of the QoS management framework happens at two levels: (1) the application developer can develop different domain applications via the *same* set of QoS specifications, and (2) different domain applications can run via the *same* run-time middleware in heterogeneous environments.

The key components in the $2K^{Q+}$ framework are (1) the *uniform QoS programming environment*; (2) the *Q-Compiler*; and (3) a reconfigurable, component-based run-time middleware such as the *run-time* $2K^Q$ *middleware.*

The *uniform QoS programming environment* provides a *uniform set* of customizable QoS specifications that can be used to develop different domains of QoS-aware applications. These QoS specifications are considered as the input *"source code"* of the Q-Compiler.

The *Q-Compiler*, the automated QoS compilation system, is the core of the unified QoS management. It compiles the corresponding QoS specifications into a well-defined set of QoS-enabled meta information, called QoS-aware Component-based Application Specification (QoSCASpec), considered as the Q-Compiler's *"object code"*, and used uniformly during the application execution phase.

QoSCASpec includes (1) possible delivery forms (configurations) of the QoS-aware application for running in heterogeneous execution environment with different resource availability; and (2) the association of reusable middleware service components and system resources for each configuration with specific semantics of QoS requirements.

The *run-time* $2K^Q$ middleware is the component-based, and dynamically reconfigurable run-time system. It is running in the distributed machines and it is the core of application execution. It assists the Q-Compiler to perform the distributed system resource translation during the application development phase. Furthermore, it instantiates, manages and controls a QoS-aware application during the application execution phase, based on the (1) Q-Compiler result, the application's QoSCASpec, (2) incoming user QoS request, and (3) dynamic run-time constraints such as resource availability, execution environment, and mobility.

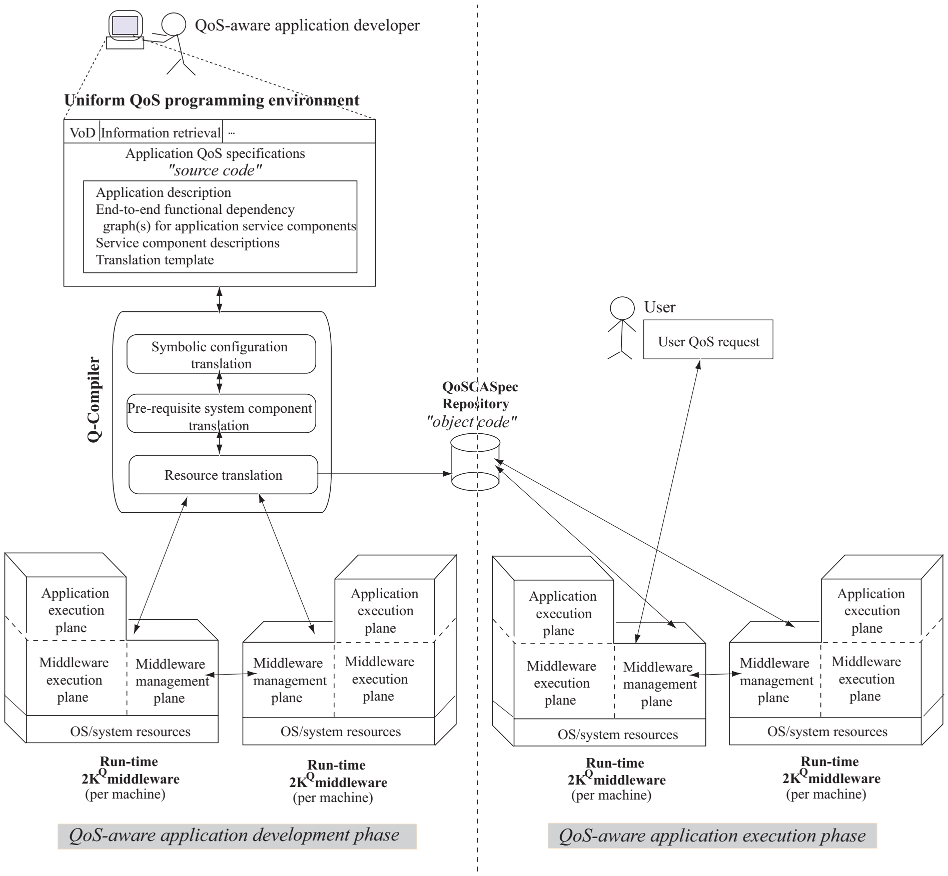


Fig. 2. $2K^{Q+}$: A Unified QoS Management Framework

The rest of the paper is organized as follows. In Sect. 2, we describe the uniform QoS programming environment, and how to develop a QoS-aware application via the uniform set of QoS specifications. In Sect. 3, we give the overview of the Q-Compiler architecture, and its core, the multi-aspect QoS translations. In Sect. 4, we describe the detailed architecture of the run-time $2K^Q$ middleware, how it assists the Q-Compiler to perform the distributed system resource translation, and how it instantiates, manages and controls a QoS-aware application during the application execution phase. In Sect. 5, we describe the implementations and our experimental results. In Sect. 6, we discuss the related work. Finally, we draw conclusions in Sect. 7.

2 Uniform QoS Programming Environment: Application Developer's View

The goal of our uniform QoS programming environment is to allow the application developer develop QoS-aware applications in different domains easily and uniformly via the same set of QoS specifications. To achieve this goal, it provides the following set of QoS specifications: (a) the application description, (b) the end-to-end functional dependency graph(s) of application service components, (c) the service component descriptions, and (d) the user-to-application-specific translation template.

The application description allows an application developer to specify the general information about the application, such as name, category, and accessibility. The Q-Compiler associates this information with the compiled results of the application. The run-time $2K^Q$ middleware uses this information as an index to the appropriate application's configuration(s). Figure 3 shows the application description for two applications: (a) the library information retrieval application, and (b) the video on demand application.

Attributes	Values
Application name	QLibAccess
Application category	Information retrieval
Accessibility	Public

(a) Library Information Retrieval Application

Attributes	Values
Application name	MonetVOD
Application category	Video on Demand
Accessibility	Public

(b) Video on Demand Application

Fig. 3. Application Description for Two Different Applications (Example)

The end-to-end functional dependency graph of application service components allows the application developer to implement an application via the composition of service components flexibly[1]. In a functional graph, we differentiate among service components as follows: each node represents a service component which can be a *specific service component*[2] (e.g., QLibServer, MPEG-II renderer, MPEG-II decoder), a *composite service component* (e.g., VoD Client consisting of MPEG-II decoder, and MPEG-II renderer), or a *generic service*

[1] We assume that the interfaces between two connected components are clearly defined, and the programming environment can check the compatibility between two connected components in the functional dependency graph.

[2] A specific service component represents a specific implementation of a functional unit. We assume that specific service components have been already built, and there exists service component repository storing these components implemented for different devices, resource demands, and with different quality levels in mind.

component[3] (e.g., Information Retrieval Service (IRS), Encoder Service (ES), Decoder Service (DS)). The dotted-rounded rectangle in Fig. 4 represents a machine, and a solid line from service component A to service component B represents a data flow from component A to component B. A functional graph is *fully-defined* if all service components are *substituted* with specific service components or sets of specific service components (in case of composite service component), and it is *partially-defined* if a service component is specified with a generic service component or a composite service component with some generic service components. Figure 4 shows the application functional graph for two applications: (a) the library information retrieval application, and (b) the video on demand application.

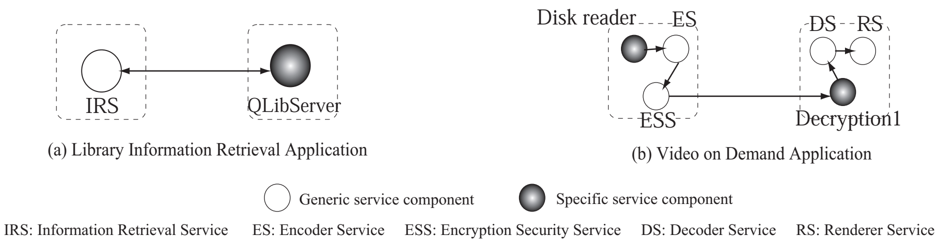


Fig. 4. Application Functional Graph for Two Different Applications (Example)

The service component description allows an application developer to specify individual service component's information which is required by the Q-Compiler. Figure 5 shows the service component descriptions for two applications: (a) the library information retrieval application, and (b) the video on demand application. The "*" represents undefined value which will be determined by the Q-Compiler. The "**" represents the undefined value which will be determined by the run-time $2K^Q$ during the application execution time.

The user-to-application-specific translation template (UtoA template) (as shown in Fig. 6) allows an application developer to define the mapping between different user QoS levels (e.g., UserIrQoS:High, UserPVQoS:High) and corresponding application-specific QoS categories (e.g., Information retrieval, Performance:Video) and QoS dimensions (e.g., request priority, format, resolution)[4]. User QoS levels are determined by the application developer as available

[3] A generic service component represents the requirement of specific service without specific implementation. The application developer can develop an application via the functional dependency graph even though he/she doesn't have all built specific service components. A generic service component will be substituted with possible specific service component(s) during the QoS compilation time.

[4] *QoS category* and *QoS dimension* are part of *QoS specifications* to characterize non-functional properties of a specific application [15, 16]. A *QoS dimension* defines a qualitative or quantitative attribute for a *QoS category*, and it is used to characterize

Attributes	Values
Service component type	Generic
Service component name	*
Service component category	IRS
Service component repository	*
Target node	**
Hardware requirements	*
Supporting QoS categories	Information retrieval:
	request priority *
	request reliability *
	valid response time *

(I) Information Retrieval Service's description

Attributes	Values
Service component type	Specific
Service component name	QLibServer
Service component category	ISS
Service component repository	ComponentRepository
Target node	boston.cs.uiuc.edu
Hardware requirements	<(CPU:SUN Ultra 60), (MEM:5MB)>
Supporting QoS categories	Information retrieval:
	request priority {Low, Medium, High}
	request reliability {Low, Medium, High}
	valid response time [5,20]

(II) QLibServer's description

(a) Library Information Retrieval Application

Attributes	Values
Service component type	Specific
Service component name	Disk reader
Service component category	DRS
Service component repository	ComponentRepository
Target node	paris.cs.uiuc.edu
Hardware requirements	<(CPU:SUN Ultra 60), (MEM:20MB)>
Supporting QoS categories	Performance:Video:
	format any
	resolution {740x480, 480x360,360x240}
	frame-rate [10,30]
	color depth {8}

(I) Disk reader's description

...

Attributes	Values
Service component type	Generic
Service component name	*
Service component category	RS
Service component repository	*
Target node	**
Hardware requirements	*
Supporting QoS categories	Performance:Video:
	format *
	resolution *
	frame-rate *
	color depth *

(VI) Renderer Service's description

(b) Video on Demand Application

Fig. 5. Service Component Descriptions for Two Different Applications (Example)

application-category-specific QoS levels for the user during the application request time. Application-specific QoS categories and dimensions are determined by the application developer as the input QoS parameters to the Q-Compiler, corresponding to different user QoS levels. *QoS levels* can be generated by the Q-Compiler, based on all combinations of user QoS levels in different application-specific QoS categories. However, without loss of generality, we assume in this paper that the Q-Compiler will generate *QoS levels* for an application, based on combinations of only the same user QoS levels in all different application-specific QoS categories. Figure 6 shows the UtoA template for two different applications: (a) the library information retrieval application, and (b) the video on demand application.

The uniform QoS programming environment enables the first level of unification, mentioned in the Introduction. Once the above QoS specifications are in place, the Q-Compiler starts to compile the QoS specifications into the QoS-enabled meta information as described in the following section.

3 Q-Compiler: Automated QoS Compilation System

The primary goal of the Q-Compiler[5], which is the key part of the $2K^{Q+}$ unified QoS management, is to compile QoS specifications of different domain applications into QoS-enabled meta information that can be used uniformly by the

a particular category [16]. For example, QoS dimension "resolution" is an attribute of QoS category "Performance:Video". QoS dimension can be interchangably used with QoS parameter.

[5] The details of the Q-Compiler framework can be found in [17].

QoS levels

	High	Medium	Low
	UserIrQoS:High	**UserIrQoS:Medium**	**UserIrQoS:Low**
Information retrieval			
request priority	High	Medium	Low
request reliability	High	Medium	Low
valid response time (min)	15	10	5

UserIrQoS: User's QoS Information retrieval category

(a) Library Information Retrieval Application

QoS levels

	High	Medium	Low
	UserPVQoS:High	**UserPVQoS:Medium**	**UserPVQoS:Low**
Performance:Video			
format	{mjpeg, mpeg-II}	{mjpeg, mpeg-II}	{mjpeg, mpeg-II}
resolution (pixelxpixel)	740x480	480x360	360x240
frame-rate (fps)	25	20	10
	UserPAQoS:High	**UserPAQoS:Medium**	**UserPAQoS:Low**
Performance:Audio			
format	{wav, mp3}	{wav, mp3}	{wav, mp3}
delay (sec)	50	75	100
sampling rate (kHz)	24	16	8
sample bit (bit)	24	16	8
	UserSecQoS:High	**UserSecQoS:Medium**	**UserSecQoS:Low**
Security			
format	des	des	des
key length encryption (bit)	128	80	64

UserPVQoS: User's QoS Performance (video data type) category
UserPAQoS: User's QoS Performance (audio data type) category
UserSecQoS: User's QoS Security category

(b) Video on Demand Application

Fig. 6. User to Application Specific Translation Template for Two Different Applications (Example)

distributed run-time middleware during the application execution in heterogeneous environments.

In the Q-Compiler, we consider the following resulting output information as QoS-relevant to ensure the QoS provisions for different domain applications during their execution:

– Possible delivery forms of an application, associated with different quality levels, for heterogeneous computing and communication environments, and for adaptations;
– Association of each delivery form with suitable set of middleware service components and their appropriate QoS parameters;
– System resource requirements of each delivery form with its associated middleware service components to be used (1) for resource reservation in case it exists, (2) for selection of the most suitable delivery form, and (3) for global resource optimization.

To achieve the primary goal, the Q-Compiler (as shown in Fig. 7) consists of three aspects of translations: (1) *symbolic configuration translation*, (2) *prerequisite system component translation*, and (3) *resource translation*.

The **Symbolic Configuration Translation** compiles the input QoS specifications of an application into possible delivery forms (configurations). This aspect of translation is based on the *generic service component's substitution* and *application functional graph transformation* with the *consistency check* of QoS requirements and QoS provisions between two consecutive specific service components in the graph. The pre-defined possible configurations for different domain applications are provided as an internal information for the translation.

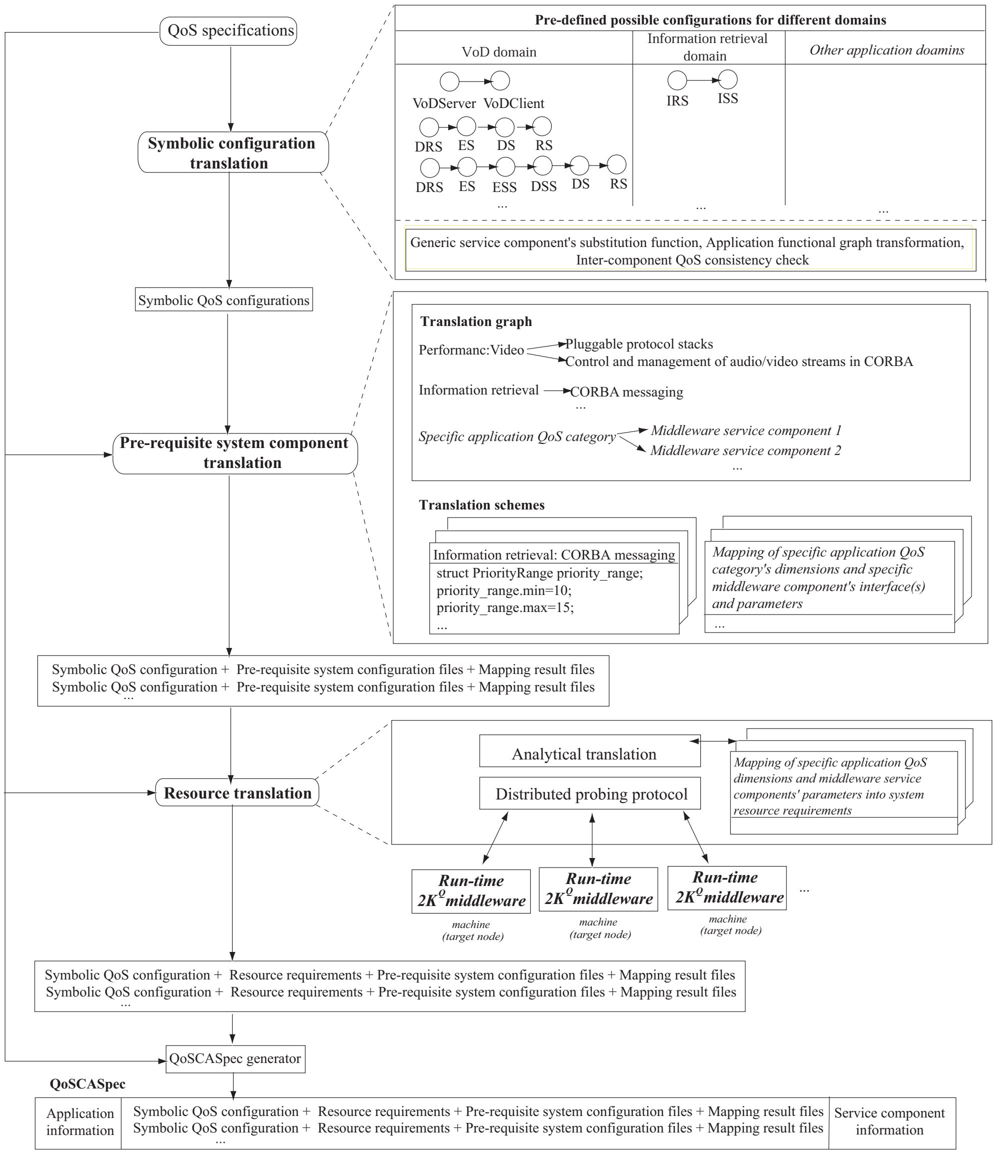


Fig. 7. Multi-aspect QoS Translations

The symbolic configuration translation can be considered as the *horizontal end-to-end configuration translation* in the application layer. The results of the symbolic configuration translation are the *symbolic QoS configurations*, represented by fully-defined functional graphs with associated *QoS levels*. Note that the number of possible configurations for an application is limited by some factors such as the failure of consistency check, the number of generated QoS levels which is usually small, and the limited number of application's possible configurations pre-defined as a part of the service component substitution and graph transformation. Hence, the considered number of feasible configurations is bounded.

The **Pre-requisite System Component Translation** determines the appropriate set of *middleware service components*, including their interfaces and QoS parameters. For each symbolic QoS configuration, the included middleware service components can be classified into two groups: (1) QoS-specific components, such as CORBA messaging, control and management of audio/video streams in CORBA, which provide service quality for specific QoS requirements, and (2) QoS-generic components, such as observer and resource adaptor in Agilos [18], DSRT [19], which perform generic functionality as adaptations, system resource management and monitoring.

This aspect of translation is based on an extensible *translation graph* which represents the association between specific application QoS categories and their suitable middleware service components, and *translation schemes* which represent the mappings between specific QoS categories' dimensions and middleware service components' interfaces and parameters. The input application QoS categories and their dimensions are retrieved from the specific service components in each symbolic QoS configuration. The pre-requisite system component translation can be considered as the refinement of each symbolic QoS configuration with the *vertical end-system configuration translation* from specific service components' QoS requirements in application layer into service components in middleware layer, and the *horizontal end-to-end configuration translation* among the determined middleware service components. The results of the pre-requisite system component translation are (1) the *pre-requisite system configuration files*, containing the associated middleware service component(s) for the symbolic QoS configuration per target node, and (2) the *mapping result files*, associated with individual pre-requisite system configuration files, containing the mapping result between specific service components' QoS category's dimensions and middleware service components' interfaces and parameters.

Note that how these two results will be used by whom depends on the involving middleware service components. In this paper, we consider two types of the middleware service components: (type I) components (e.g., CORBA messaging, DSRT) which need the instrumentation of application specific service components' source codes to include their pre-defined APIs, and (type II) components (e.g., pluggable protocol) which do not need any modification of the application specific service components' source codes. For type I, the compiled results will be interactively returned as hints to the application developer. There hints will indicate modification places in the specific service components' source codes to include the middleware service components' APIs, corresponding to the mapping information in the mapping result files. For type II, the compiled results will be used directly by the run-time middleware to dynamically link the required middleware service component(s) with their suitable parameters into the application execution environment.

This phase of compilation enables that different QoS-aware applications are executed uniformly in the *same* reconfigurable, component-based run-time middleware. It means that this phase represents the second level of QoS management unification as discussed in the Introduction.

The **Resource Translation** translates each symbolic QoS configuration and its associated middleware service component(s) into distributed system resource requirements. It is dealing with coordination among distributed resource brokers, resource negotiation, and resource translation. This aspect of translation is based on the *analytical translations*, and the *distributed probing and profiling techniques*. The analytical translation consists of the mapping or translation functions from specific QoS dimensions in the application layer and/or middleware layer into system resource requirements. The analytical translation will be used if there exists a suitable function. The distributed probing protocol, assisted by the run-time middleware, (1) instantiates the specific application service components and their associated middleware service components into the distributed target nodes, (2) coordinates with distributed resource brokers to perform the system resource probing, (3) collects the probing results, and (4) associates them with the previously compiled results. The resource translation can be considered as the *vertical resource translation* from QoS requirements in the associated upper layers into the system resource requirements at the end-system, and the *horizontal end-to-end resource translation* among the distributed resource brokers. The results of the resource translation are the minimum end-to-end system multi-resource requirements for each symbolic QoS configuration, associated with specific middleware service components, and specific QoS requirements.

The application's compiled results are represented as the *QoS-aware Component-based Application Specification (QoSCASpec)*, which is the Q-Compiler's *"object code"*. QoSCASpec includes (1) application description; (2) set of *QoS configurations*; and (3) service component description for all specific service components in the symbolic QoS configurations. QoSCASpec is installed in the QoSCASpec repository as a "ready-to-use" configuration information for end-to-end QoS setup and adaptation of an application.

Figure 8 demonstrates the Q-Compiler's multi-aspect QoS translations and the QoSCASpecs for two different applications: (a) the library information retrieval application, and (b) the video on demand application. Note that in our implementation, the QoSCASpecs are represented by the XML-based description. When the applications' QoSCASpecs are available, the applications are ready for execution by the distributed run-time middleware.

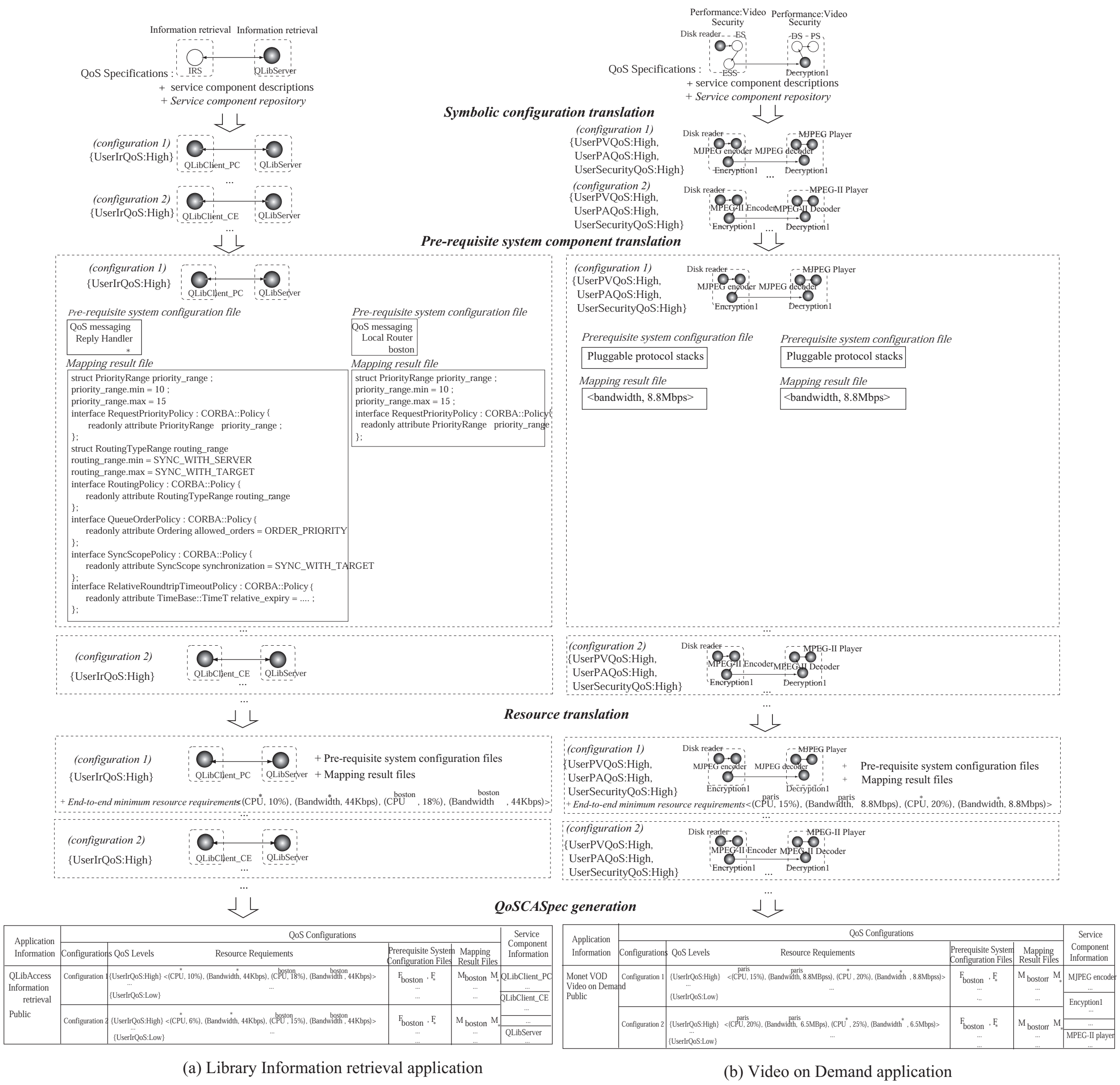


Fig. 8. Multi-aspect QoS Translations of Two Different Applications (Example)

4 Run-Time $2K^Q$ Middleware

In this section, we present the component-based and reconfigurable run-time $2K^Q$ middleware, which plays major role during application execution.

4.1 Architecture

The run-time $2K^Q$ middleware (shown in Fig. 9) consists of three planes: the *application execution plane*, the *middleware execution plane*, and the *middleware management plane*.

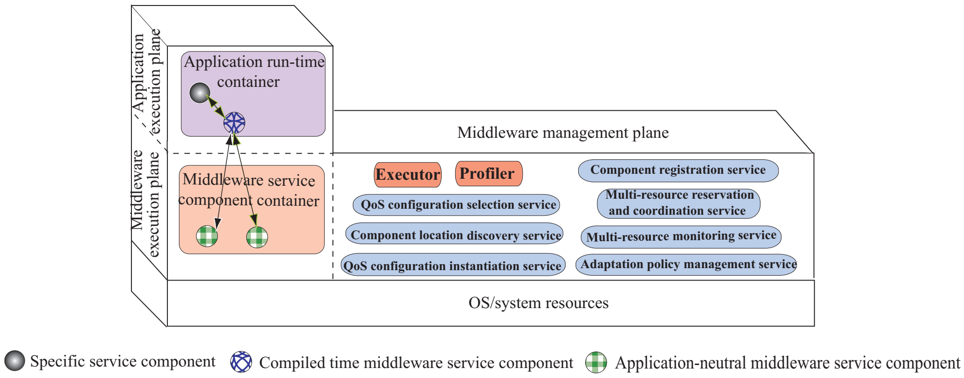


Fig. 9. Run-time $2K^Q$ Middleware Architecture

In the **application execution plane**, the *application run-time container* is the place where the application service components, compiled with required middleware service components' APIs and their necessary libraries, are instantiated and running. There exists one application run-time container per application.

In the **middleware execution plane**, the *middleware service component container* is the place where the application-neutral middleware service components (e.g., resource brokers [19], observer and resource adaptor [18]), are instantiated and running. There exists one middleware service component container per machine and is shared among different applications. Both containers are dynamically created, managed and controlled by the middleware management plane.

The **middleware management plane** provides a set of middleware management services which help to instantiate, manage, and control an application with the quality of service provision. Specifically, it needs to provide support for (1) *QoS setup* of the application corresponding to user QoS request[6], and (2)

[6] For example, an end-user provides a user request as follows: [<application name = "MONETVoD">, <application category = "Video on Demand">, <QoS level

QoS adaptations of the application in case of changes in user preference, and dynamic constraints such as resource availability, execution environment and mobility. The goal is to maintain the QoS provision or gracefully degrade QoS during the application run-time if changes occur.

The *profiler* and *executor* are the front-end agents of the run-time $2K^Q$ middleware waiting for probing service requests from the Q-Compiler's resource translation, and for user QoS requests from end-users for the application execution, respectively. They handle the requests using the provided services in the middleware management plane.

The middleware management plane composes of the following services: (1) *QoS configuration selection service*; (2) *component location discovery service*; (3) *QoS configuration instantiation service*; (4) *component registration service* ;(5) *multi-resource reservation service*, (6) *multi-resource monitoring service*, and (7) *adaptation policy management service.*

The *QoS configuration selection service* consults the *QoSCASpec repository* to get all possible QoS configurations corresponding to the user QoS request for an application. The selection service chooses the best configuration among the returned QoS configurations based on the current available resources, and execution environment (e.g., it performs the match between the current available resources and the configurations' system resource requirements). If the chosen configuration consists of a component with undefined location (target node), the *component location discovery service* will be activated to discover (i.e., contacting the public domain of running components) the component's best location.

The *QoS configuration instantiation service* is responsible to instantiate specific service components and their associated middleware service components defined in the pre-requisite system configuration files[7] into the distributed locations (target nodes). The instantiation services in the distributed locations coordinate among themselves to (1) dynamically create the application run-time containers and the middleware service component containers in the distributed locations,(2) dynamically download the service component(s) from a service component repository if the required specific or middleware service component(s) are not located in the specified locations, and (3) instantiate the service components in these containers. These steps will be performed only if no instance of the required service component is running on a particular target node. Note that when a service component is instantiated, it may advertise itself to a public domain of running components via the *component registration service.*

The *multi-resource reservation and coordination service*, based on the end-to-end run-time multi-resource negotiation, is activated after the QoS configuration instantiation service if the involving distributed locations support the reserva-

= [High:UserPVQoSHigh, UserPAQoS:High, UserSecQoS:High]>, <accessibility = "Public">].

[7] The configuration files are accompanied with corresponding mapping result files (sec Sect. 3) and the mapping result files will be used by the run-time middleware only if the associated middleware service components do not need any instrumentation of the specific service components' source codes.

tion of resources. The end-to-end resource negotiation is based on the minimum resource requirement information compiled during the application development phase.

The *multi-resource monitoring service* is responsible to measure and gather, via the resource brokers, current available resources and service components' resource requirements, at the end-system and in the distributed locations. The returned result from the resource monitoring service can be used as hint for selecting the most suitable configuration during the QoS configuration selection service, and the most suitable QoS adaptation during the application run-time. During the application development phase, the Q-Compiler's resource translation uses this service via the profiler to measure the minimum end-to-end multi-resource requirements for each specific configuration.

The *adaptation policy management service* performs the setup of adaptation policies which include the possible QoS configurations and their transitions, corresponding to user preferences encoded in the user QoS request, and application developer's choices specified during application development phase. The setup of the adaptation policies can be considered as part of the QoS setup. It helps the run-time middleware to manage the adaptations of an application appropriately during the application run-time.

While we present the middleware management plane as the composition of seven necessary middleware management services to manage and control the application execution plane, the middleware management plane is configurable and extensible by additional services. Moreover, the middleware management plane on different machines with varied capacities (e.g., high performance PCs, handheld PDAs) and environments (e.g., reservation-enabled, or best-effort) can be dynamically customized. For example, the middleware management plane on the handheld PDAs can be configured to include only partial services or even no services, and to rely on a gateway with fully-support middleware management plane.

In the following subsections, we describe how the run-time $2K^Q$ middleware assists the Q-Compiler's resource translation during the application development phase, and how it manages and controls the execution of an application during the application execution phase.

4.2 Run-Time $2K^Q$ Middleware and Q-Compiler's Resource Translation

The run-time $2K^Q$ middleware performs the following steps (see Fig. 10) to assist the Q-Compiler's resource translation measuring the distributed multi-resource requirements for a specific configuration[8]. Step 1: the profiler gets the resource translation's distributed probing request. Step 2: the profiler activates the QoS configuration instantiation service to collaboratively create the application run-time containers and the middleware service component containers

[8] Note that the individual steps in this paragraph correspond to the steps in Fig. 10.

in the distributed locations, corresponding to the specific configuration associated with the distributed probing request. Step 3: the instantiation services in the distributed locations, then, dynamically download the required service components from the service component repository, and instantiate them into the created containers. Step 4: the profiler activates the resource monitoring service to collaboratively gather the distributed multi-resource requirements for the configuration. Step 5: the profiler returns the minimum end-to-end multi-resource requirements of the configuration to the Q-Compiler's resource translation.

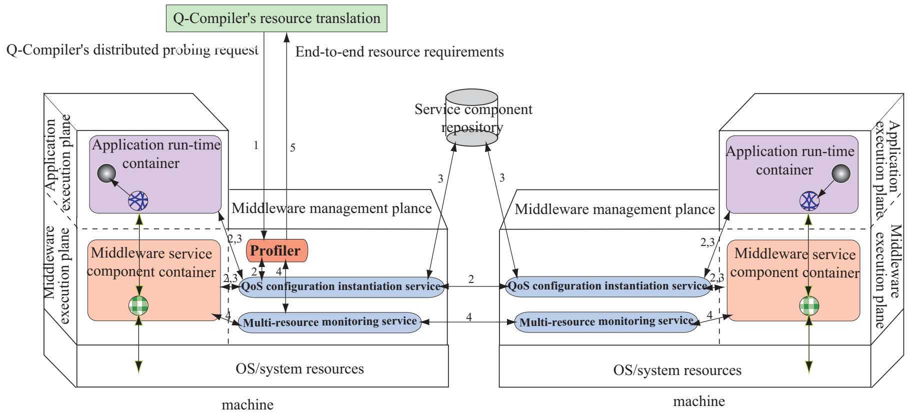


Fig. 10. Run-time $2K^Q$ Middleware Assisting Q-Compiler's Resource Translation

4.3 Run-Time $2K^Q$ Middleware and QoS-Aware Application Execution

In this section, we describe how the run-time $2K^Q$ middleware instantiates, manages, and controls an application execution corresponding to a user QoS request, the application's QoSCASpec, and the dynamic run-time constraints such as current resource availability, and execution environment.

The run-time $2K^Q$ middleware performs the following steps[9] (see Fig. 11). Step 1: the executor gets a user QoS request. Step 2: the executor activates the QoS configuration selection service to find the most suitable configuration, which will be determined based on (1) the application's compiled result getting from the QoSCASpec repository (see Step 2.1 in Fig. 11), and (2) the current available resources getting from the multi-resource monitoring service (see Step 2.2 in Fig. 11). The adaptation policy management service performs the setup of the adaptation policies among different possible configurations getting from the QoSCASpec repository (see Step 2.3 in Fig. 11). The component location

[9] Note that the individual steps in this paragraph correspond to the steps in Fig. 11.

discovery service will be activated in Step 3 if there exists an undefined location in the selected configuration from Step 2. Step 4: the QoS configuration selection service returns the most suitable configuration to the executor. Step 5: the executor activates the QoS configuration instantiation service to collaboratively create the application run-time containers in the distributed locations, according to the specified information in the returned configuration. Step 6: the instantiation services in the distributed locations dynamically download the required service components from the service component repository, and instantiate them into the created containers. Note that in Step 6, when a service component is instantiated, it may also register itself to a public domain of running components via the component registration service. In Fig. 11, we assume that the middleware service component containers are already available in the distributed locations. Step 7: the executor activates the multi-resource reservation and coordination service to coordinate the reservation of resources in the distributed locations for the end-to-end configuration. The resource reservation service will be activated only if the involving distributed locations support the reservation model. Step 8: the executor returns the execution result to the end-user.

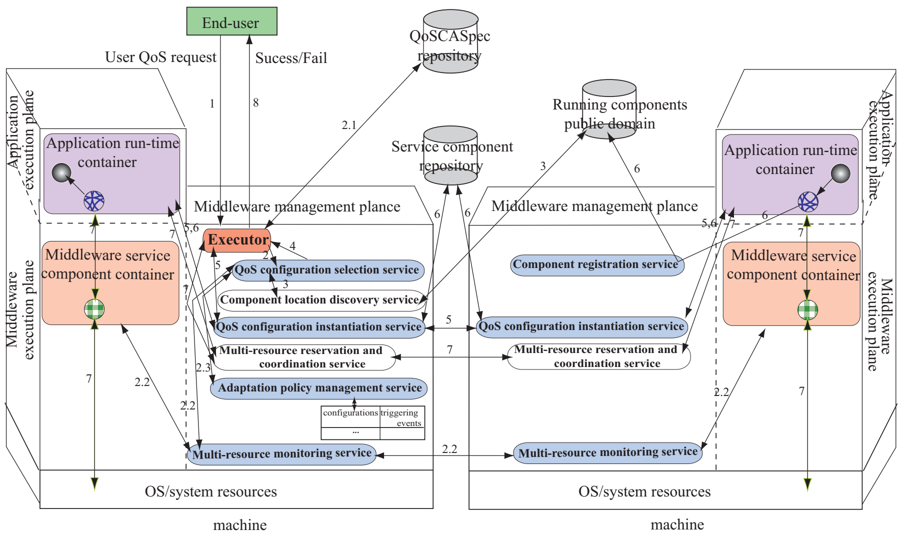


Fig. 11. Run-time $2K^Q$ Middleware Managing QoS-aware Application Execution

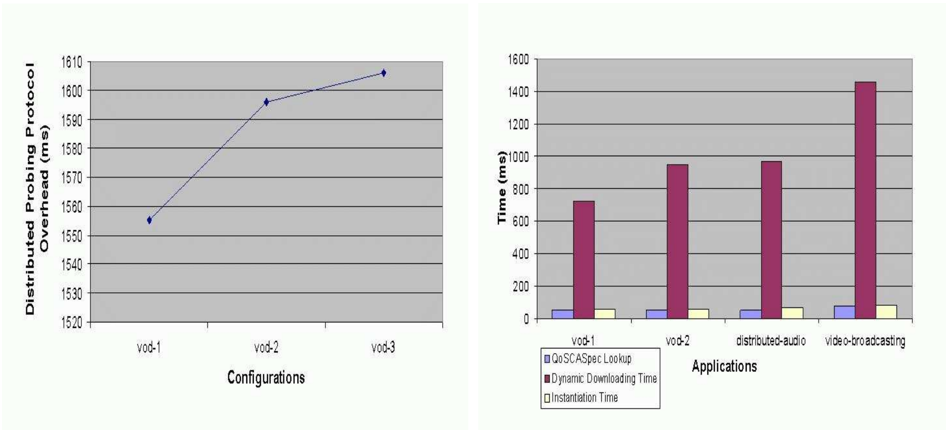
Because the run-time $2K^Q$ middleware is component-based and reconfigurable, and allows for dynamic downloading and linking of service components into distributed target nodes securely, it can execute and ensure the QoS provisions for different types of QoS-aware applications uniformly, based on the compiled meta information in applications' *QoSCASpec*s.

5 Implementation and Experiments

The implementation is divided into two parts: (1) the *uniform QoS programming environment* and the *Q-Compiler's multi-aspects QoS translations* are implemented in Java, and integrated with the visual programming environment [20]; (2) the distributed *run-time* $2K^Q$ *middleware* is implemented as a set of CORBA components, based on the dynamic reconfigurable middleware "dynamicTAO" [21], and the resource reservation model in the QualMan system [22].

The run-time $2K^Q$ nodes are connected via a 100 Mbps Ethernet[10]. The nodes are: (1) two Sun Ultra-60 workstations, each with two 360 MHz processors and 768 MB memory, (2) one Sun Ultra-60 workstation with two 450 MHz processors and 1024 MB memory. All machines are running SunOS 5.7. The file system is NFS. The Q-Compiler runs on Pentium-II PC with one 366 MHz processor and 128 MB memory, and connects to the run-time $2K^Q$ nodes via the 100 Mbps Ethernet.

We show our first results of the unified QoS management framework's performance via the *overhead of the resource translation's* distributed probing protocol (see Fig. 12(a)), and the *overhead of the QoS setup* (see Fig. 12(b)) for different applications.



(a) (b)

Fig. 12. (a) Resource Translation Times for Different Configurations During the Application Development Phase, (b) QoS Setup Times for Different Applications During the Application Execution Phase

In Fig. 12(a), vod-1, vod-2 and vod-3 are three different configurations of a video on demand application: (1) a single server with a single client, (2) a single server with two clients, and (3) a single server with three clients. The graph is an average of fifteen runs for the three configurations. The results show that

[10] We are working towards heterogeneous networks with different devices' capabilities.

the average resource translation times, based on the *distributed probing protocol*, are 1555.1, 1596.1, and 1606.4 ms, respectively. Times vary corresponding to the number of associated service components needed to be instantiated in the configurations, service components' sizes and target locations. Note that the measurements do not include the duration of individual service components' local resource probing.

In Fig. 12(b), we measure the *QoS Setup* time of four applications: vod-1, vod-2, distributed audio (distributed-audio) with three audio players, and video broadcasting (video-broadcasting) with a server, a gateway, and two receivers. The QoS setup time includes QoSCASpec lookup, dynamic downloading of service components and component instantiation. The graph is an average of fifteen runs for the four applications. The results show that the QoS setup times for the four applications vary mainly according to their dynamic downloading times, which are 723.0, 945.5, 964.5, and 1458.7 ms, respectively. Dynamic downloading times vary corresponding to the number of associated service components in the configurations, the service components' sizes and the target locations. The average QoSCASpec lookup times for the four applications are 53.3, 54.7, 48.9, and 72.3 ms, respectively. The QoSCASpec lookup is stable and very small because the current QoSCASpec repository is running on the same machine as the executor getting the user QoS request. Also, in these experiments, the QoSCASpec lookup performs only the configuration matching without considering additional constraints. The average component instantiation times for the four applications are 55.3, 59.3, 66.1, and 82.1 ms respectively. The component instantiation time is the time used in CORBA method invocations to start the distributed service components forming the application. If we assume that service components (some or all) in a configuration are already in places, the QoS setup time for the configuration will be very small.

6 Related Work

Extensive related work exists in different areas of the QoS:

- **QoS specifications and QoS translations**
  QoS specifications, proposed as part of QoS architectures or QoS middleware architectures, depend on the design and objectives of the architectures. For example, in QuO project [14], QoS is specified via a suite of description languages based on aspect-oriented programming [23]. In QoSME [7], QoS is described via a Quality of Service Assurance Language (QuAL). In CORBA, QoS is specified via pre-defined interfaces of different QoS extensions such as control and management of audio/video streams in CORBA [9], CORBA messaging [10], fault tolerant CORBA [11], and real-time CORBA [12]. In Agilos middleware [18], QoS is defined via rules and membership functions. In Q-RAM project [6], QoS is represented via utility functions.
  In the area of *QoS Translation*, QoS translations, based on analytical functions, are proposed. For example, in [2] Nahrstedt et al. propose a translation

from a multimedia application's QoS parameters into transport subsystem's QoS parameters, and in [24] Kim et al. propose a translation from MPEG video parameters into CPU requirements. The proposed analytical translations are application-specific and applied to multimedia domain. Moreover, they deal only with the translation between the application QoS parameters and the system resource requirements.

- **QoS Architectures**

  Several reservation-based QoS architectures and approaches already exist. For example, in [2, 3, 4, 5] researchers propose end-to-end QoS management frameworks for multimedia applications. In the best-effort environment, the adaptation-based QoS architectures exist. For example, Agilos [18] proposes a framework assisting in QoS enforcement for a distributed visual tracking application. Q-RAM project [6] proposes a QoS management framework based on the multi-resource allocation model mainly focusing on the global resource optimization. QoSME [7] with the Quality of Service Assurance Language (QuAL) provides the abstractions for QoS management to the underlying network management. These QoS architectures and approaches are designed only for one type of applications, or to handle particular aspect of QoS provisions such as the resource management. If one wants to run a different type of applications, the existing QoS architectures do not scale, i.e., might not be applicable, or reusable.

- **QoS in Distributed Object Computing (DOC) Middleware**

  In *distributed object computing (DOC) middleware* such as CORBA, control and management of audio/video streams in CORBA, CORBA messaging, fault tolerant CORBA and real-time CORBA [9, 10, 11, 12] are proposed to provide quality of service for different types of applications. In this case, an application will be QoS-aware if the application developer deploys these QoS-related interfaces. Hence, an application developer has to learn different IDL interfaces for different types of provisions. In addition, the application developer has to know the semantics of these interfaces, and how to translate his/her application QoS requirements into these interfaces and their parameters appropriately. For example, in real-time CORBA, an interface allows the application developer to specify the protocol configuration such as protocol type, ORB protocol property, and transport protocol property. What is not clear is how an application's QoS specifications should be translated appropriately into these IDL interfaces and parameters. Other QoS efforts in DOC middleware are the optimizations of ORBs, such as TAO project [13] at Washington university to support the real-time messaging. BBN's Quality Object (QuO) project [14] allows an application developer to develop distributed applications that can adapt to the changing quality of service in CORBA environment. Lancaster's multimedia component architecture [25] is extended beyond CORBA or DCOM and takes application QoS parameters into account. Adapt project [26] allows explicit bindings in CORBA via open bindings. Qualities of services in DOC middleware are also tailored toward specific type of applications or particular aspect of QoS provisions.

7 Conclusions

Different domains of distributed component-based applications, running in heterogeneous execution environments, need different quality of service semantics. It is hard to provide quality of service for individual applications, and even harder to handle them uniformly in a QoS management framework. The difficulty is in both the development phase, and the execution phase to integrate the QoS as part of the application.

In this paper, we describe the architecture and the philosophy of a unified QoS management framework, $2K^{Q+}$, based on the integrated approach of the QoS compilation and the component-based and reconfigurable run-time middleware. The framework provides uniform and systematic mechanisms for developing a QoS-aware application during the application development phase, and for ensuring QoS provisions based on the compiled information and dynamic run-time constraints, during the application execution time.

While the component-based middleware concept by itself is not novel, (1) the introduction of an automated QoS compilation concept, which helps the application developer to decide the possible configurations and the appropriate set of middleware components running in heterogeneous execution environments, and (2) the integration of these two concepts, forming the unified QoS management framework, are novel.

We believe that $2K^{Q+}$ is a practical solution for a unified QoS management framework, which includes not only QoS setup and QoS provision during the application run-time, but also QoS programming and compilation for different applications during the application development.

References

[1] K. Nahrstedt, D. Wichadakul, and D. Xu. Distributed qos compilation and run-time instantiation. *In Proceedings of the Eighth IEEE/IFIP International Workshop on Quality of Service*, pages 198–207, June 2000.

[2] K. Nahrstedt and J. Smith. Design, implementation and experiences with the omega end-point architecture. *IEEE Journal on Selected Areas in Communication*, 14(7):1263–1279, September 1996.

[3] A. Campbell, G. Coulson, and D. Hutchison. A quality of service architecture. *Computer Communication Review*, 24(2):6–27, April 1994.

[4] L. C. Wolf. *Resource Management for Distributed Multimedia Systems*. Kluwer, Boston, Dordrecht, London, 1996.

[5] A. Hafid and G. Bochmann. An approach to qos management in distributed multimedia applications: Design and an implementation. *Multimedia Tools and Applications*, 9(2), 1999.

[6] 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*, pages 298–307, December 1997.

[7] P. G. S. Florissi. *QoSME: QoS Management Environment*. PhD thesis, Columbia University, Department of Computer Science, 1996.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] D. Schmidt, D.Levine, and C. Cleeland. *Advances in Computers, Marvin Zelkowitz (editor)*, chapter Architectures and Patterns for High-performance, Real-time ORB Endsystems. Academic Press, 1999.

[14] J. Zinky, D. Bakken, and R. Schantz. Architecture support for quality of service for corba objects. *Theory and Practice of Object Systems*, 3(1):55–73, January 1997.

[15] G. Bochmann, B. Kerherve, and M. Mohamed-Salem. Quality of service management issues in electronic commerce applications. *to be published as a chapter in a book.*

[16] S. Frolund and J. Koistinen. Quality of service specification in distributed object systems design. *In Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems*, pages 1–18, 1998.

[17] D. Wichadakul and K. Nahrstedt. Distributed qos compiler. *Technical Report UIUCDCS-R-2001-2201 UILU-ENG-2001-1705, Department of Computer Science, University of Illinois at Urbana-Champaign, (submitted for journal publication)*, February 2001.

[18] 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*, 17(9):1632–1650, September 1999.

[19] H. Chu and K. Nahrstedt. Cpu service classes for multimedia applications. *In Proceedings IEEE International Conference on Multimedia Computing and Systems*, pages 296–301, June 1999.

[20] X. Gu, D. Wichadakul, and K. Nahrstedt. Visual qos programming environment for ubiquitous multimedia services. *to appear in Proceedings of IEEE International Conference on Multimedia and Expo*, August 2001.

[21] M. Roman, F. Kon, and R.H. Campbell. Design and implementation of runtime reflection in communication middleware: the dynamictao case. *In Proceedings.*

19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware, pages 122–127, June 1999.

[22] 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(3-4):229–257, 1998.

[23] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson. Qos aspect languages and their runtime integration. *In Lecture Notes in Computer Science, Springer-Verlag of the Fourth International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 1511:303–318, May 1998.

[24] K. Kim and K. Nahrstedt. *Building QoS into Distributed Systems, Andrew Campbell, Klara Nahrstedt (editors)*, chapter QoS Translation and Admission Control for MPEG Video, pages 359–362. Chapman and Hall, 1997.

[25] D. G. Waddington and G. Coulson. A distributed multimedia component architecture. *In Proceedings of the First International Enterprise Distributed Object Computing Workshop*, pages 337–345, October 1997.

[26] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. Supporting adaptive multimedia applications through open bindings. *In Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 128–135, May 1998.