QoS-Aware Meta-Data Compiler for Ubiquitous Multimedia Applications

Duangdao Wichadakul[1], Klara Nahrstedt[2]
[1]National Electronics and Computer Technology Center
112 Paholyotin Rd., Klong 1, Klong Luang, Pathumthani 12120, Thailand
[2]Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL, 61801, USA
duangdao@hpcc.nectec.or.th, klara@cs.uiuc.edu

June 7, 2004

Abstract

The reusability of available multimedia and middleware services brings new challenges for enabling flexible and efficient development and deployment of distributed end-to-end multimedia applications with specific Quality-of-Service(QoS) in ubiquitous environments. The main challenges in reusing available components include understanding and utilizing of domain-specific components and middlewares with various semantics, and enabling their QoS-aware interoperability in ubiquitous environments that resource fluctuations, device and service changes are a common phenomenon. This paper presents a *QoS-aware meta-data compiler framework* that provides a solution for the challenges. The framework extends standard component construction and composition with QoS-related meta-data. It defines a set of QoS-aware models and meta-data translation models that are essential for modelling QoS consistency. The framework also enables QoS-aware semantics and interfaces for interoperability among connected components forming a QoS-aware multimedia application. Besides the defined models, the framework introduces an architecture that integrates the models with a set of high-level specifications, a meta-data compiler protocol, and a run-time support to form a programming environment, called Q-Compiler. The Q-Compiler helps to automate the development and deployment of a component-based, QoS-aware application, deployable in ubiquitous environments. To validate the viability of the Q-Compiler, we use it to develop a mobile Video-on-Demand application in an active space project. The experimental results show that the introduction of a translator code between connected components does not degrade the overall service quality of the components. Although the main contributions of the framework are validated via multimedia domain,

we anticipate that fundamental concepts and design will be applicable to other application domains.

Index Terms: reusable components, ubiquitous computing, QoS-aware meta-data programming and compilation, QoS-aware models, QoS-aware semantic and interface interoperability, QoS-aware meta-data translations.

1 Introduction

A ubiquitous computing promotes the proliferation of various stationary, embedded and mobile devices interconnected by heterogeneous networks (e.g., wired, wireless, infrared). It promises a dynamic, distributed computing environment, that seamlessly and pervasively delivers applications to the user, despite changes of resources, devices, and locations. In a ubiquitous environment, component technology has become widely-accepted for enabling the rapid construction of applications. Available component models and reusable multimedia components allow fast and flexible packaging, distribution and deployment of component-based multimedia applications. Only a few of these applications, however, are Quality-of-Service(QoS)-aware. One of the main reasons is that it is complex and difficult to build QoS into these applications in such ubiquitous environments. The complexity and difficulty lie in the following facts.

First, reusable components in different layers (e.g., multimedia components in the application layer, and QoS-enabling services in the middleware and RM management layer) that can be reused to compose a QoS-aware application have their own QoS notations and semantics. Also, different components in the same layer are domain-specific. For example, in the application layer, video-streaming applications deal with frame size, frame rate, and color depth. Video conferencing applications deal with lip synchronization skew, and visual tracking applications deal with tracking precision. In middleware and resource management layers, a CPU scheduling service deals with period and cycle time to ensure the required processing time for a running component. A bandwidth broker service deals with bandwidth reservation for transmitting the data and/or control of the component. These specific QoS semantics add to the complexity of interoperability, translations, and mappings between connected components, both in the same layer and across layers and among distributed end systems (See Figure 1).

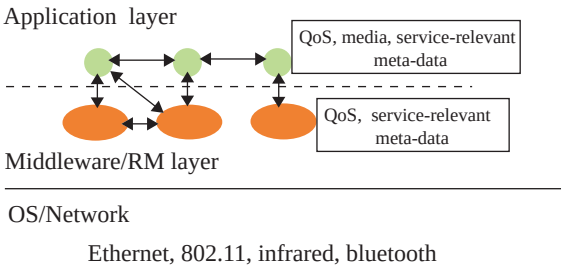


Figure 1: Horizontal and Vertical QoS-Aware Interoperability Among Service Components in Different Layers in Distributed End Systems

Second, while a ubiquitous environment is dynamic due to changing availability of devices, software components and resources, a traditionally developed QoS-aware application is tightly-coupled with specific application service components and QoS-enabling services. Hence, the developed application might not be adaptable to a dynamic environment and deployable in different environments, due to unavailability of required software components, devices, and resources.

Most of the past efforts on enabling QoS for multimedia applications have focused on QoS architectures[1, 2, 3, 4, 5], resource management algorithms and systems[6, 7, 8, 9, 10, 11, 12, 13], QoS mappings and translations[1, 14, 15, 16, 17, 18, 19], and measurement-based resource translations[20, 21, 22]. Agilos[23] is the adaptation-based QoS middleware, based on fuzzy logics and control theory. In the Distributed Object Computing (DOC) community, standard middleware such as CORBA defines control of audio/video streaming specification[24]. MULTE-ORB[25] is a reflective multimedia object request broker that mainly considers the reconfigurable bindings and explicit streaming bindings to support real-time requirements in ORB. GOPI[26] and OpenORB[27] at Lancaster provide a set of middleware modules representing a generic middleware platform, which can be extended to support different types of interactions (e.g., streaming, messaging) with specific semantics and QoS specifications. DotQoS[28] extends the .NET Remoting, which is the middleware of Microsoft's .NET framework, with QoS specification, the definition of QoS-enabled component, the handling of QoS contracts, and the composition of QoS mechanisms in the message path. In pervasive computing and communication, BASE[29] is proposed to provide an extensible middleware via easy-to-use abstractions to access remote services and device-specific capabilities. It provides the dynamic extensibility supporting the range of devices from sensors to full-fledged computers. PCOM[30] offers application programmers high-level programming abstractions that

capture the dependencies between components using contracts. It supports automatic adaptation in cases where the execution environment changes to the better or to the worse.

These research results could be considered as reusable QoS-enabling services for building different aspects of QoS provisions into multimedia applications executable in ubiquitous environments. As addressed previously, specific QoS semantics of services add to the complexity of their reusability. An application developer will need a good understanding of QoS characteristics of available services in order to select among them appropriately. Moreover, the application developer needs to know how to translate or map between different semantics and interfaces of deployed QoS-enabling services and multimedia services in order to form a QoS-aware multimedia application. Hence, the effective development of a QoS-aware application depends mainly on the expertise and experience of the developer. Moreover, the developed application that is tightly-coupled with specific QoS-enabling services results in the inflexibility of its deployment.

In this paper, we introduce a QoS-aware *meta-data compiler* that addresses fundamental requirements and presents a solution for enabling the flexible reuse of multimedia components and QoS-enabling services for composing a multimedia application with specific QoS requirements. The goals of the compiler are to (1) reduce the complexity of developing a component-based, QoS-aware multimedia application, and (2) improve the flexibility and efficiency of its deployment in ubiquitous environments. To achieve these goals, the framework defines a component-based QoS-aware application as the composition of application and QoS-enabling services where each service is described by a meta-data (See Figure 2). The framework consists of a set of QoS-aware models, and meta-data translations and compilations that help to (1) automate the QoS consistency check and interoperability of the composed application, (2) enable QoS provision for the application based on QoS-enabling services association, (3) optimize the association results, and (4) generate QoS-aware deployment descriptors of the application, executable in ubiquitous environments.

The rest of this paper is organized as follows. Section 2 introduces the overall concepts and design of the meta-data compiler. We justify why the a meta-data compiler is needed. Section 3 presents QoS-aware models and component categorizations designed for the meta-data compiler. Section 4 presents meta-data translation models for enabling QoS-aware semantic and interface interoperability. Section 5 describes the meta-data compiler architecture that puts together QoS-

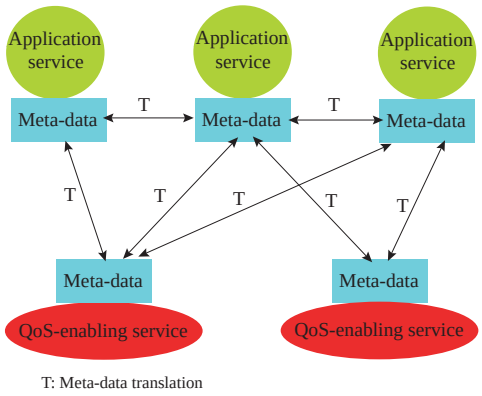


Figure 2: A Generic Component-Based QoS-Aware Application and Its Related Meta-Data Translations

aware models and meta-data translation models into a QoS-aware programming and compilation environment workable by a meta-data compiler protocol and a run-time support. Section 6 illustrates the usage of the meta-data compiler to develop and deploy a multimedia application. Section 7 presents a set of experimental results that validate performance and usability of the framework. Section 8 discusses related work. Finally, Section 9 concludes the paper.

2 Overall Framework

We propose the meta-data compiler to fulfil the lack of a framework that enables easy and flexible reuse of available multimedia components and QoS-enabling services to compose a QoS-aware multimedia application. In a meta-data compiler, we need (1) *QoS-aware models* that can be used to describe QoS-aware component constructions[1] and domain-specific compositions[2] of application; (2) *translation models* that include the translation specifications and their compilations for enabling and automating different types of QoS-aware semantic interoperability among connected components forming a QoS-aware application; (3) *high-level specifications* that can hide the complexity of QoS-aware programming, customizable by an application developer; and (4) a *meta-data compiler protocol* and a *run-time support* that put together available models and specifications into a QoS-aware programming environment usable by an application developer to develop and deploy a

[1]The component construction extracts functionality from a reusable component and describes it with QoS, media, and service-relevant meta-data and rules.

[2]The domain-specific application composition concatenates components to form functional dependency graphs of the application, and constructs of meta-data and rules for the application.

QoS-aware multimedia application into ubiquitous environments, with less complexity and higher flexibility.

These entities are general principles for building any domain-specific meta-data compilers for component-based QoS-aware applications. They are analogous to basic entities such as language specification, lexical analyzer, parser, code generator, and their compilation protocol, comprising a typical programming language compiler. However, a meta-data compiler is domain-specific; there is no generic meta-data compiler although general principles are the same for each compiler. For instance, the meta-data compiler for multimedia domain needs QoS-aware models that validate compatibility and consistency of multimedia-specific QoS parameters along the end-to-end composition of the application.

3 QoS-Aware Models and Service Categorizations

Even though component models have been proposed in standard component architectures[31, 32, 33], none of them considers QoS as a fundamental concept. The lack of standard QoS-aware models for both component construction and composition makes the programming models of these architectures less applicable for enabling the development of QoS-aware applications. No standard is defined for reusing, packaging and deploying such applications.

To enable the practical development and deployment of QoS-aware applications based on reusable service components in different layers (e.g., application, middleware, resource management), we define service categorization models, QoS-aware component model, QoS-aware application model, and QoS-enabling service model, as fundamental models for the meta-data compiler.

Service categorizations deal with the classifications of application-level services and QoS-enabling services (e.g., middleware, resource management). Both classifications divide available services into categories[3] according to specific service domain and functionality. *Common ontology*[4] is defined for each service category, as *meta-data*, representing QoS as requirement and provision of quality in *data semantics* and/or *service semantics*. Service categories and their defined common ontologies reduce the complexity and promote the generalization of translation models between two connected

[3]While these categorizations are not exhaustive, they are extensible with a new service category and its reusable service components in the future.

[4]In our framework, an ontology of a service category represents common QoS parameters of the service category.

components, which are usually encoded with their own semantics. They enable different types of QoS-aware semantic and interface interoperability for service composition, and help to separate responsibility between a service component developer and a QoS-aware application developer.

The *QoS-aware component model* represents an atomic unit in the application layer for composing a service or an end-to-end application. It is labelled with QoS meta-data derived from the categorization of application services. The *QoS-aware application model* defines how to compose a service or an application. Furthermore, it defines (1) QoS consistency check between two connected components, and (2) labelling of the service or application with an aggregated QoS meta-data, derived from the composed components. The *QoS-enabling service model* represents an atomic unit in the middleware or resource management layers for enabling QoS management and enforcement. It is labelled with QoS meta-data derived from the categorization of QoS-enabling services. We describe service categorizations and QoS-aware models in details in the following sections.

3.1 Service Categorization

3.1.1 Application Service Categorization

To describe **service semantics**, we classify application services into a hierarchy as shown in Figure 3(a), according to domain and functionality of services. The first level of the hierarchy (e.g., AC_1, AC_2) represents different service domains. Each domain is further divided into subcategories corresponding to the specific functionality of services. A leaf of the hierarchy (e.g., AC_{11}, AC_{12}) represents a specific category that is described with common and configurable parameters representing service semantics. We use these parameters to customize and/or control service quality of services in the category. A service in a category inherits common parameters from the category. Figure 3(b) represents a concrete service hierarchy for multimedia domain. Each leaf of the hierarchy represents multimedia services categorized into categories (e.g., encoding service, playback service, encryption service) according to their specific functionality. The encoding service, for instance, consists of three configurable parameters: "quantization factor," "DCT coefficient," and "scaling factor." A specific encoder (e.g., MPEG encoder) will inherit these meta-data from the encoding service.

To describe **data semantics** of an application service, we categorize possible input and out

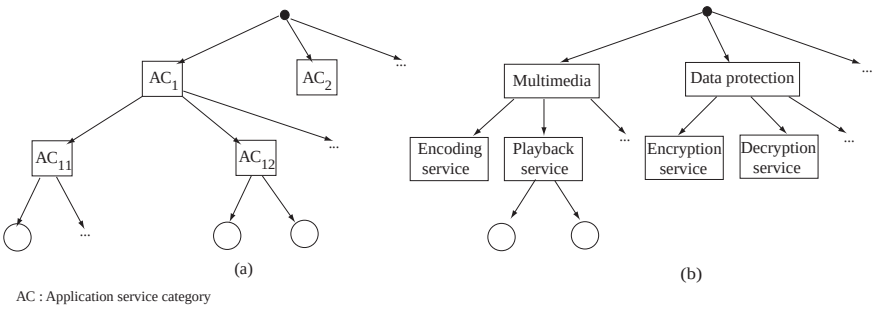


Figure 3: Categorization of Reusable Application Services

data into a hierarchy, as shown in Figure 4, according to data types and formats. The first level of the hierarchy represents common data types. Each internal nodes that are not in the first level represents a sub data type which will inherit parameters from its parent and might introduce some new parameters. A leaf node represents a specific data format which will inherit all parameters from its associated data type and might also introduce some new parameters. In Figure 4, we classify data types into video, audio, raw data, text, and message. Each data type is then further refined into sub data types or formats. As shown in Figure 4, the video data type category is defined by four QoS parameters: format, sample size, sample rate, and sample bit. An MPEG video format category, then, refines sample size and sample rate for each frame type (I, P, B), and introduces QoS parameters related to the group of picture (GOP).

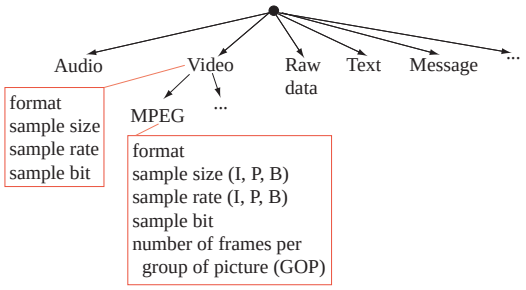


Figure 4: Categorization of Data Types and Formats

3.1.2 QoS-Enabling Service Categorization

To describe **service semantics** of QoS-enabling services, we classify reusable QoS-enabling services into a hierarchy, according to their aspects of QoS provisions, as shown in Figure 5(a). The overall hierarchy is similar to the hierarchy of application service categorization. The first level

of the hierarchy includes categories offering a specific aspect of QoS provision (e.g., SC_1, SC_2). Each category is further divided into subcategories corresponding to different approaches for QoS provisions. Leaves (e.g., SC_{11}, SC_{12}) of the hierarchy are specific categories where each category is described by a set of common and configurable parameters representing service semantics. These parameters are used to customize and/or control service quality of the QoS-enabling service in the category. A QoS-enabling service in a category will inherit parameters from the category.

Figure 5(b) represents a concrete service hierarchy of QoS-enabling services usable by multimedia domain. A leaf of the hierarchy, CPU scheduling service, is described by a set of configurable service-quality parameters (e.g., period, cycle time).

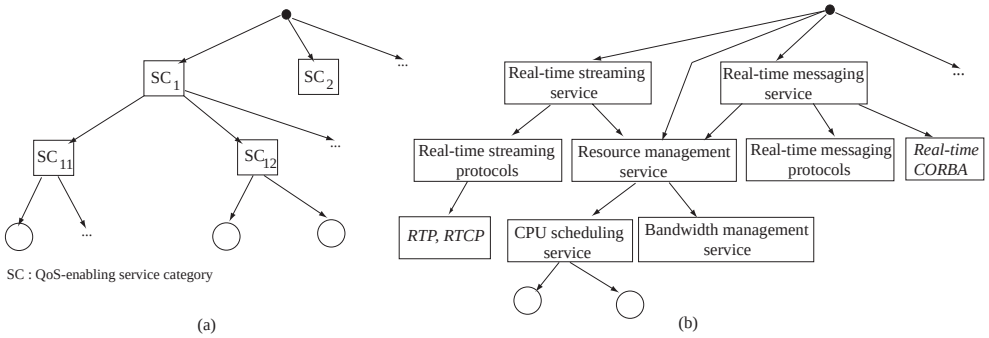


Figure 5: Categorization of Reusable QoS-Enabling Services

3.2 QoS-Aware Models

3.2.1 QoS-Aware Component Model

Abstract Model of a QoS-aware component extends a standard component model with QoS-related meta-data information. The **abstract model** utilizes the task model by Liu et al.[34], as shown in Figure 6(a). The *configurable service parameters* (S), the *input data quality* (Q_{in}), and the *output data quality* (Q_{out}) represent service and data semantics of the service component. Q_{in} and Q_{out} specify specific data types and formats that the component expects as its processing input and provides as its processing output, respectively. The *reward profile* (R_k) represents a mapping from Q_{in}, system resource requirements (r), and configurable service parameters (S) to Q_{out}.

Concrete Model constructs and extends a generic CORBA component[5], shown in Figure 6(b)[6], with QoS-related information, as shown in Figure 6(c). A *QoS-aware component* consists of (1) application service component code, (2) a QoS-related meta-data, and (3) an adaptation handler. The *QoS-related meta-data* represents properties of the component that are necessary for enabling QoS-consistent service composition and efficient instantiation. The meta-data consists of the overall information (e.g., name, category, interfaces, hardware and software constrains, required libraries) about the component, plus the description of service semantics, data semantics, and reward profile, presented in the abstract model. The *adaptation handler* is a set of interfaces that are introduced for controlling the component to adapt according to changing environment. It connects to the event sinks and sources in the model and extends the component with a set of interfaces such as *tuneServiceParams(params_vector)* and *tuneDataParams(params_vector)*.

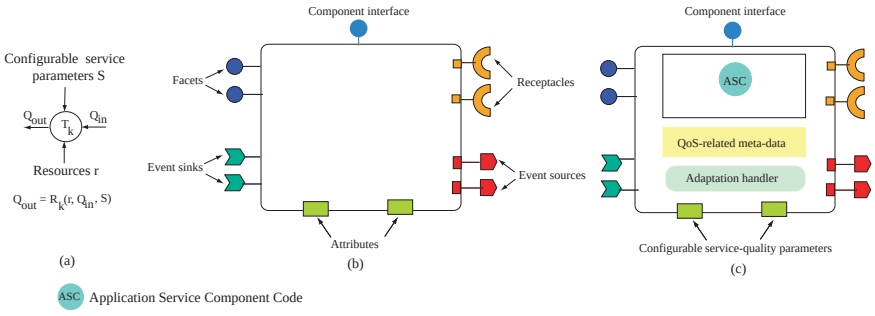


Figure 6: QoS-Aware Component Models: (a) Abstract Model, (b) CORBA Component, (c) Concrete Model

3.2.2 QoS-Aware Application Model

Abstract Model of a QoS-aware application defines how to compose an *end-to-end* QoS-aware application. As shown in Figure 7(a), a composition is represented as functional graphs of QoS-aware service components. The model deploys a task-flow model and considers each QoS-aware service component, y_i, as a task with input data quality (Q_{in}) and output data quality (Q_{out}), driven by available resources. The model defines how QoS-aware service components should connect

[5]We describe our model as an extension of a CORBA component because the CORBA component has been defined with features that are necessary for any component architectures.

[6]In a CORBA component, the attributes represent configuration properties. The facets represent the offered operation interfaces. The receptacles represent the required operation interfaces. The event sources and sinks represent the produced events and consumed events, respectively.

together to form a consistent QoS-aware multimedia application. It defines a functional consistency check and a QoS consistency check as described in[35].

Concrete Model is composed of a set of service graphs or configurations, as shown in Figure 7(b). Each graph is composed of QoS-aware service components. A *QoS-aware application* is described by (1) QoS-related meta-data and (2) an adaptation handler. The *QoS-related meta-data* describes (i) the structure of the application, (ii) its data semantics, and (iii) its adaptation control policy. The meta-data of the application structure includes (a) general information of the application such as name, category and accessibility, (b) functional dependency of services in form of service graphs, and (c) description for each component in the service graphs. The Q_{in}^{source} is the input data semantics of a source node in a service graph. The adaptation control policy describes how the application should adapt according to the availability of resource, device, and services. The *adaptation handler* is introduced to control the adaptation of the application. It cooperates with the adaptation handlers of service components comprising the application. The handler assumes the availability of an underlying event service. It subscribes to the event service for specific events, and performs corresponding actions, defined in the adaptation control policy.

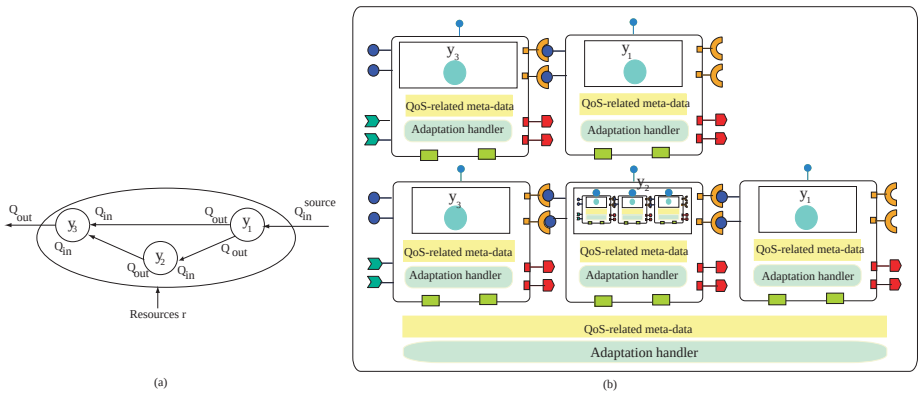


Figure 7: QoS-Aware Application Models: (a) Abstract Model, (b) Concrete Model

3.2.3 QoS-Enabling Service Model

Abstract Model of QoS-enabling service describes a reusable QoS-enabling middleware or resource management service with QoS-related information. The model in Figure 8(a) describes the QoS-enabling service through its service semantics that include a requested service quality (Q_{req}),

a provided service quality (Q_{prov}), an accessing attribute, and a reward profile (R_k). Q_{req} allows a QoS-aware service component that wants to use the QoS-enabling service to specify its service requirement. In general, a specific translation from Q_{in}, Q_{out}, and S of the QoS-aware service component in Figure 6(a) into Q_{req} is needed. Q_{prov} returns a level of service quality that the QoS-enabling service can provide. The *accessing attribute* determines if the QoS-enabling service is exclusive or can be shared among service components requesting the service. The *reward profile* (R_k) represents the mapping from Q_{req} and system resources (r) to Q_{prov}.

Concrete Model is constructed as a reusable QoS-enabling middleware or resource management service, attached with a *QoS-related meta-data*, and wrapped with an *intermediate representation (IR)*, as shown in Figure 8(b). The *QoS-related meta-data* represents attributes of the QoS-enabling service that are necessary for enabling a *consistent integration* between itself and a service component. The overall description is very similar to those QoS-related meta-data of the QoS-aware component model, with additional IR, defined for each QoS-enabling service category. The IR is a set of common interfaces and parameters that enable service abstraction for QoS enforcement and provision. Common interfaces are generalized from the functionality and pre-defined interfaces, provided by specific services in the same category. Common parameters are service semantics defined by the category. Services in the same category will share the same intermediate representation. For example, instances of a CPU real-time scheduling service, such as Dynamic Soft-Real-Time Scheduler (DSRT) and RT-Mach scheduler, will share the same common interfaces and parameters for CPU scheduling and CPU bandwidth reservation. The introduction of IR allows the late binding between a service component and its associated QoS-enabling service. During the instantiation, the application bound with an intermediate representation can dynamically bind with any specific QoS-enabling service, implementing the IR.

4 Meta-Data Translation Models for Interoperable Services

As different application service components and QoS-enabling services have their own pre-defined service and data semantics with specific interfaces, it is difficult to efficiently reuse these components to compose a QoS-aware application, without having suitable mechanisms enabling their interop-

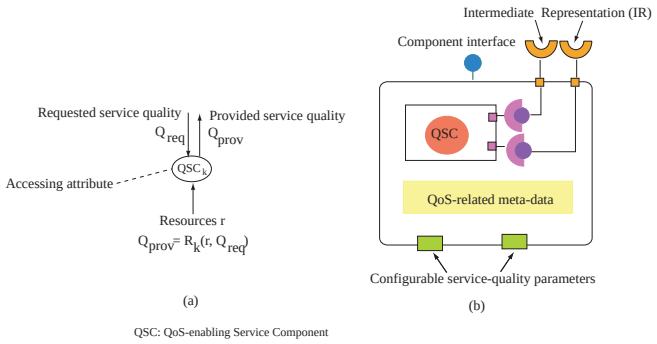


Figure 8: QoS-Aware QoS-Enabling Service Models: (a) Abstract Model, (b) Concrete Model

erability. While past research results have solved aspects of interoperability problems (e.g., [36]), none of them focuses on QoS-aware semantic interoperability. In addition, even though several QoS translations and mappings [37, 14, 15, 17, 16, 38, 39, 22] result in interoperability of QoS-aware semantics, they mainly focus on mappings from specific set of QoS parameters to resource requirements, and do not consider interoperability from programming or software reuse point of view.

In this section, we present a set of generalized meta-data translation models that are essential for enabling the flexible and consistent construction and composition of a component-based, QoS-aware application. These translation models (a) define meta-data specifications for different translations, (b) determine appropriate operations to translate from a specification into another, and (c) compile a specification into real codes. Based on service categorizations and pre-defined common ontologies via service and data semantics, we can generalize the meta-data translations into three models, as shown in Figure 9(a) and (b) for abstract and concrete models, respectively:

- (A) Common-actual ontology translation that handles QoS-aware semantic mappings between the service and data semantics defined for a service category and the actual service and data semantics of a specific service in the category;

- (B) Common-common ontology translation that handles QoS-aware semantic mappings between two common service and data semantics of two service categories;

- (C) Intermediate representation translation that handles QoS-aware interface mappings between common interfaces (an intermediate representation) of a service category, and the actual interfaces, given by the implementation of a specific QoS-enabling service in the category.

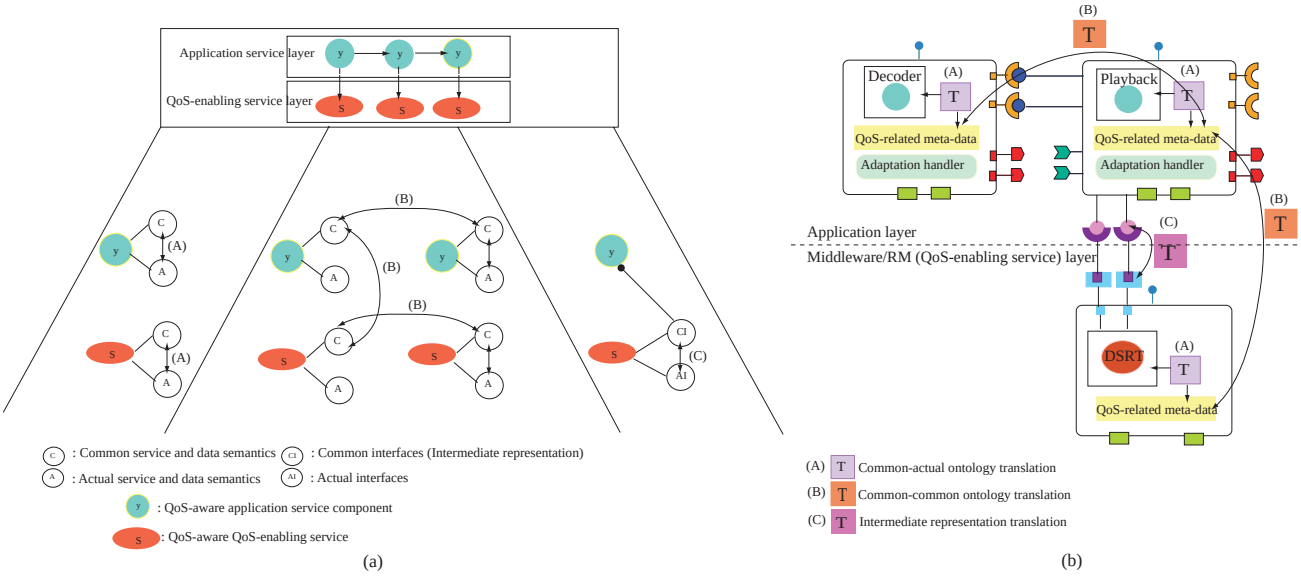


Figure 9: Meta-Data Translation Models: (a) Abstract Model, and (b) Concrete Model

These translation models provide formal mechanisms and allow a separation of responsibility in handling QoS-aware mappings according to different roles of developers. The compilation results of the translation specifications are reusable. They enable the automation for composing a QoS-aware application, hence, make the development of the application more practical and effective. We describe each translation model in following sections.

4.1 Common-Actual Ontology Translation

The common-actual ontology translation utilizes service categorization and pre-defined common semantics discussed in Section 3. As available application services and QoS-enabling services are developed with their own QoS semantics, it is not our goal to build all translators between sets of actual parameters of services and their corresponding common parameters. Instead, this translation is proposed as a mapping placeholder for an application service or a QoS-enabling service developer to specify a proper translation between the actual parameters of a service and the pre-defined common parameters. A translation consists of two main parts, the specification and the compilation.

The *translation specification*, as shown in Figure 10, includes: (1) header, (2) substitution mappings from common to actual parameters, and (3) substitution mappings from actual to common parameters[7]. The header includes the involved common and actual ontologies and helper functions.

[7]Sometimes, the actual-common ontology translation is needed for calling back from the middleware or resource

The substitution mappings can belong to one of the patterns: 1-1, 1-M, and M-1, shown in Figure 10(b). The 1-1 pattern represents the direct mapping between a common parameter and an actual parameter. The 1-M pattern conveys the mapping from a parameter in the source ontology to a set of parameters in the target ontology. The M-1 pattern demonstrates the mapping from multiple parameters in the source ontology to a single parameter in the target ontology.

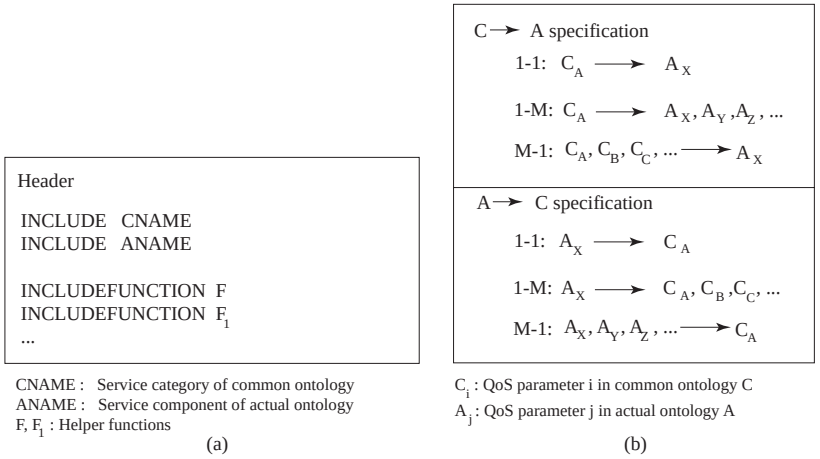


Figure 10: Specification of Common-Actual Ontology Translation: (a) Header, (b) Detailed Specification

The *translator* compiles the translation specification into a *semantics translator*, as an object code in a specific programming language (e.g., C++ or java). The 1-1 and M-1 patterns will be translated into an individual assignment statement. The 1-M mapping pattern will be compiled into multiple assignment statements, where each corresponds to a parameter in the target ontology in the specification. The *semantics translator* is a software code, which acts as a "glue code" for solving a QoS-aware semantic interoperability between a pair of common and actual semantics. Figure 11 shows an example of the common-actual ontology translation for the Dynamic Soft-Real-Time Scheduling (DSRT) service. The *CToATranslation* and *AToCTranslation* elements consist of the translation specification from common to actual and from actual to common semantics, respectively.

management to the service component for QoS adaptation. Hence, there is a placeholder for it in the translation specification.

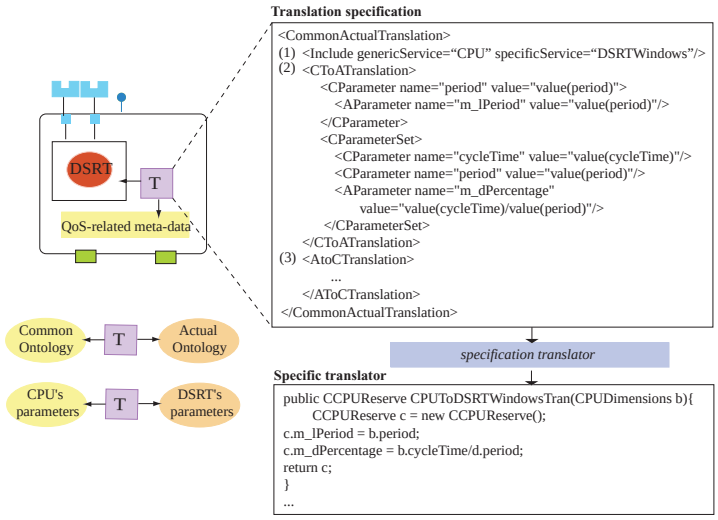


Figure 11: Common-Actual Ontology Translation (Example)

4.2 Common-Common Ontology Translation

While several QoS translations and mappings between two sets of QoS parameters (e.g., video streaming parameters and resource-specific parameters) are proposed, there are no formal specifications and compilation mechanisms, which enable (a) reusability of these QoS translations/mappings, and (b) fast and QoS consistent composition of a QoS-aware application. The common-common ontology translation is proposed as a mapping placeholder for a developer, who understands how composing components with different common semantics can communicate, to specify the proper translation between two common semantics. The common-common ontology translation consists of two main parts that have a very similar structure and functionality as the common-actual ontology translation, with the addition of the one-to-one-special (1-1-special) pattern. The added pattern is similar to the one-one pattern except that the target parameter is described not only by a source parameter but also by some parameters in the target ontology. It is different from the many-to-one pattern, where all parameters affecting the target parameter are in the source ontology. The pattern will be translated into an assignment statement.

Figure 12 shows an example of the common-common ontology translation between a video playback service and the scheduling service. The *C2ToC1Translation* and *C1ToC2Translation* elements consist of the translation specifications from common 1 to common 2 and from common 2 to common 1 semantics, respectively. Each is translated into a function call consisting of assignment

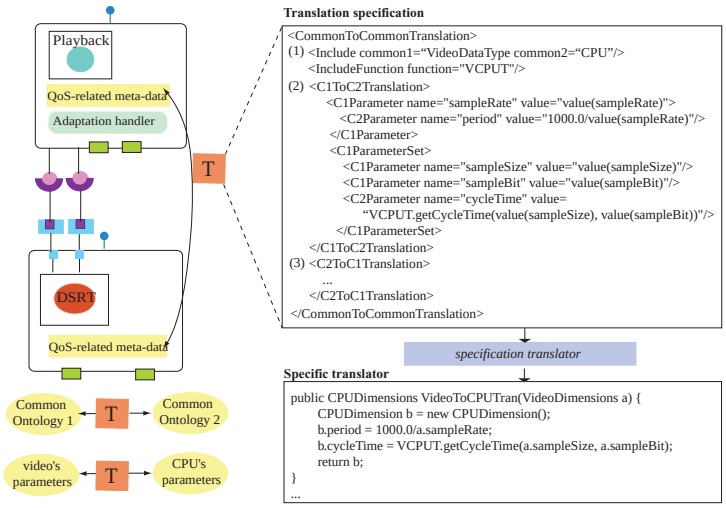


Figure 12: Common-to-Common Ontology Translation (Example)

statements derived from the specified mapping patterns. As shown in Figure 12, *VideoDataType-ToCPUTran()* function is translated from *C1ToC2Translation* element and its child elements.

4.3 Intermediate Representation Translation

Reusable QoS-enabling middleware or resource management services are implemented with their own interfaces. The intermediate representation translation is introduced to enable a mapping between interfaces of an implemented service and general interfaces, defined by the intermediate representation of the service category that the QoS-enabling service belongs to. This translation allows a QoS-enabling service developer to specify a proper translation between the intermediate representation (wrapper) of a service category and the actual interfaces of a QoS-enabling service in the category. This translation consists of two main parts that have the same structure as the previous two translations. However, its mapping patterns are different from the previous patterns. This translation deals with direct and indirect interface mapping patterns. The direct interface mapping pattern represents a mapping from return type, name, and a list of parameters of a common interface into return type and name of an actual interface, assuming that the actual interface has the same set of parameters as of the common interface. The indirect interface mapping pattern is used if the actual interface expects a set of parameters different from the parameters of the common interface. In this case, it is helped by a common-actual semantics translator. Each of direct and indirect mapping patterns will be compiled into a function with the return type,

name, and parameters defined as attributes of the common interface. This function will call the corresponding actual interfaces defined in its mapping detailed specification.

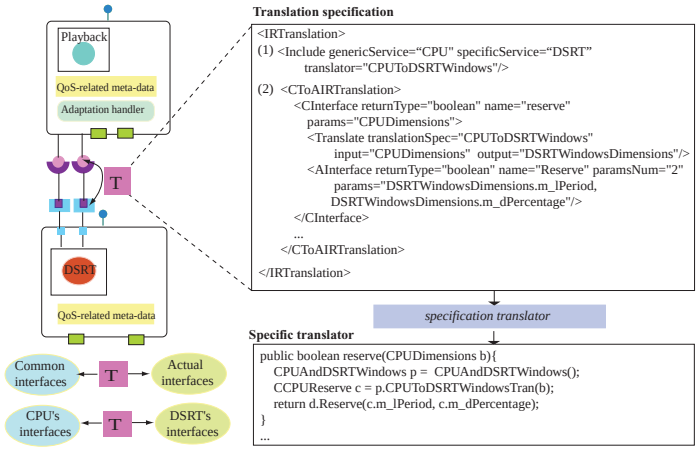


Figure 13: Intermediate Representation Translation (Example)

Figure 13 shows an example of the intermediate representation (IR) translation from the IR of CPU scheduling service into the actual interfaces of DSRT. Each *CInterface* element consists of translation specification from a common interface of the CPU scheduling service to an actual interface of the DSRT. As shown in Figure 13, the *reserve()* function of CPU scheduling service is mapped to the *Reserve()* function of DSRT.

5 Meta-Data Compiler Architecture

The meta-data compiler architecture, called the Q-Compiler, integrates QoS-aware service models and meta-data translation models into a working QoS-aware development and deployment environment via the meta-data compiler protocol and the run-time meta-data execution, described in the following sections.

5.1 Meta-Data Compiler Protocol

The meta-data compiler protocol, as shown in Figure 14(a), enables a flexible and effective development and deployment of a component-based QoS-aware application in dynamic environments. Its concept is to separate application model from its specific implementation and generate alternative service configurations (compositions) that can be instantiated in different environment accord-

ing to resource and service availability. The meta-data compiler protocol consists of two phases: environment-independent and environment-dependent compilations.

The *environment-independent compilation* processes the input high-level, QoS-aware application specification, utilizing QoS-aware models and their QoS-related meta-data. It (a) checks the QoS consistency of the application configuration, (b) associates each QoS-consistent configuration with *generic* QoS-enabling services, (c) allows the application developer to perform the instrumentation of intermediate representations of generic QoS-enabling services into an application component's code if necessary, (d) compiles the adaptation rules in the meta-data specification into a XML representation, used as the adaptation control policy for customizing the adaptation handler for the application, and (e) combines the compilation results in previous steps into a XML document, called the *QoS-aware application deployment descriptor*[8], as shown in Figure 14(b). The descriptor represents a portable and environment-independent QoS-aware application meta code, with QoS-enabled and QoS-consistent information.

The *environment-dependent compilation* helps an application developer to deploy a generic QoS-aware application represented by the *QoS-aware application deployment descriptor* into a specific deployment environment with satisfactory QoS. Based on the availability of semantics translators, resulted from the meta-data translations, and the help of the underlying run-time meta-data execution[40], the environment-dependent compilation (a) compiles the descriptor into alternative QoS-consistent configurations with different possible service quality provisions, (b) optimizes the compilation results with cost models for enabling the effective setup and adaptation for the application, and (c) combines all results into a XML document, called the *QoS-aware Component-based Application Specification (QoSCASpec)*, as shown in Figure 14(c). The QoSCASpec is the environment-dependent deployment descriptor.

5.2 Run-Time Meta-Data Execution

The run-time meta-data execution[40] is a component-based, reconfigurable middleware. It consists of a set of management services that comprise a distributed run-time system for the meta-data compiler (e.g., for probing, interacting with underlying services in a specific deployment environment),

[8]The descriptor is comparable to the EJB's deployment descriptor [31] and CCM's descriptors [32] with the extension of QoS-enabling information.

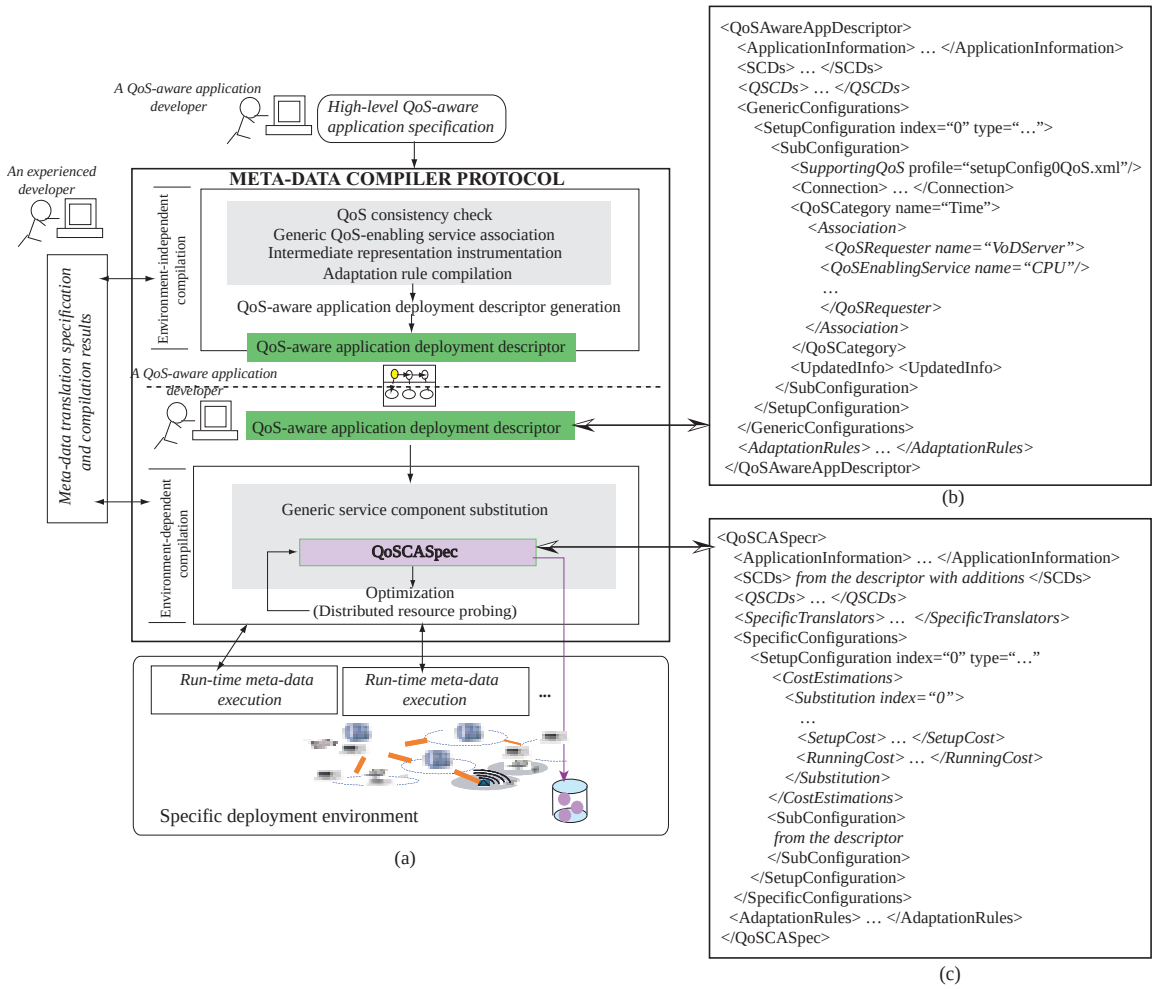


Figure 14: Meta-Data Compiler Architecture: (a) Meta-Data Compiler Protocol and Run-Time Meta-Data Execution, (b) Environment-Independent Deployment Descriptor, (c) Environment-Dependent Deployment Descriptor

and uses the result of the meta-data compiler in running an application. The management services are: (1) configuration selection service; (2) component location discovery service; (3) configuration instantiation service; (4) component registration service; (5) distributed multi-resource monitoring and probing service, and (6) adaptation management service.

The *configuration selection service* consults the QoSCASpec repository to get all possible QoS-aware configurations satisfying QoS requirements of the user for application instantiation. It chooses the best configuration among the returned results, based on available resources and services in the execution environment. If the chosen configuration consists of a component with an undefined location, the *component location discovery service* will be used to discover the best location for the

component. Note that the configuration selection service could be considered a mapping problem from a service graph into the best composition of distributed services. In a large ubiquitous environment or a large peer-to-peer infrastructure, this service could be very complex. Research results such as [41, 42] are proposed to deal with this mapping problem in a large, distributed environment.

The *configuration instantiation service* helps to instantiate application service components and their associated QoS-enabling services into distributed locations. The instantiation services in distributed locations coordinate among themselves to (1) dynamically create the containers in the distributed locations, (2) dynamically download service components from a service component repository into the locations, and (3) instantiate the components in the containers. These steps will be performed only if no instance of the required service component is running on a particular target node. An instantiated component will advertise itself to a public domain of running components via the *component registration service*.

The *distributed multi-resource monitoring service* measures and gathers available resources in the distributed locations. Its result is considered as a factor in selecting the most suitable configuration for application instantiation and adaptation. The *distributed multi-resource probing service* measures the resource requirements of an individual component or a distributed application for different QoS provisions. The meta-data translations use this service to measure the minimum resource requirements for different levels of service quality that can be offered, by an individual application service component or a QoS-enabling service. The *adaptation management service* controls and manages adaptations for QoS-aware applications, according to adaptation rules of the application, specified as a part of the input high-level specification.

While the run-time middleware is presented as the combination of these management services, it can be dynamically customized to be executable on different machines with various capacities (e.g., high performance PCs, PDAs) and environments (e.g., reservation-enabled, or best-effort).

6 QoS-Aware Application Programming and Execution

This section illustrates how a developer uses the Q-Compiler to develop a component-based QoS-aware application and how an end user executes the developed application via the Q-Compiler.

6.1 Development Process

To develop a component-based QoS-aware application, first, a developer uses the *application description GUI* to specify (1) the overall construction of the application, (2) the description of individual service components with QoS, and (3) the setup configurations. Figure 15 shows the input of the developer in the application description GUI for a mobile Video-on-Demand application.

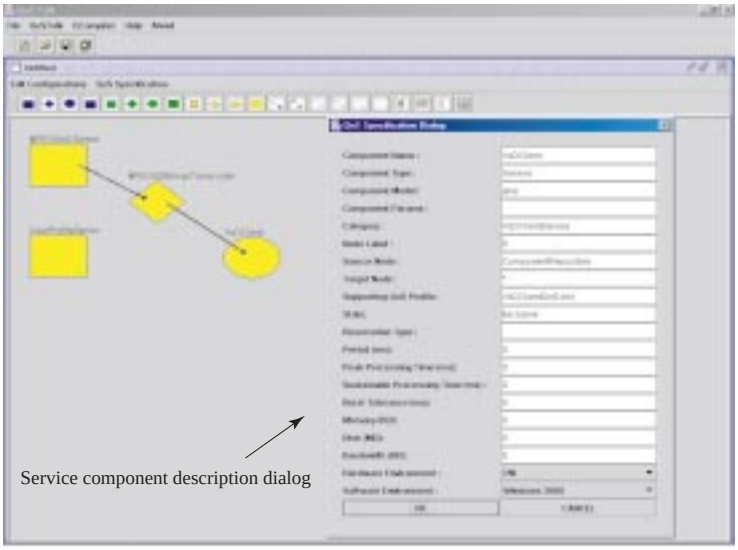
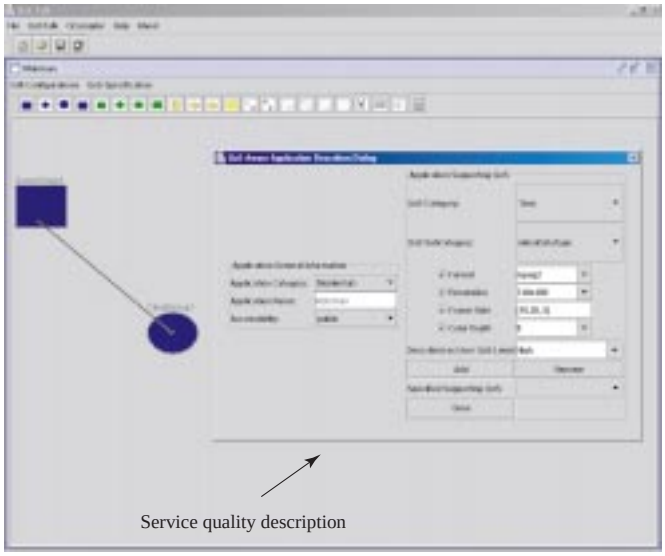


Figure 15: Application Description for a Mobile Video-on-Demand Application (Example)

Second, the developer uses the *service quality description GUI*, as shown in Figure 16(a), to specify QoS requirements expected to be provided by the application. The developer could specify service quality of the application via QoS categories, QoS subcategories, and their corresponding parameters. The meta-data compiler automatically transforms these service quality inputs into a XML representation, as shown in Figure 16(b), for the mobile Video-on-Demand application.

Third, the developer uses the *adaptation control description GUI*, as shown in Figure 17, to specify how the run-time meta-data execution should manage and control the application according to resource availability, load balancing, and mobility. The adaptation is specified in the *if events then actions* clauses. This description will be compiled into a XML representation during the environment-independent compilation.

Based on inputs from the above three descriptions, including the pre-defined QoS-aware models, and the compilation results of the translation models, the meta-data compiler protocol compiles



(a)

```
<?xml version="1.0"?>
<UtoATemplateProfile>
  <QoSCategory name="Time">
    <QoSSubCategory name="videoDataType">
      <UserQoSLevel level="High">
        <Format type="String"> mpeg-2 </Format>
        <Resolution type="CompoundSet"> {740x480} </Resolution>
        <Framerate type="IntegerRange"> [25,30,2] </Framerate>
      </UserQoSLevel>
      <UserQoSLevel level="Medium">
        <Format type="String"> mpeg-2 </Format>
        <Resolution type="CompoundSet"> {480x360} </Resolution>
        <Framerate type="IntegerRange"> [20, 25, 1] </Framerate>
      </UserQoSLevel>
      <UserQoSLevel level="Low">
        <Format type="String"> mpeg-2 </Format>
        <Resolution type="CompoundSet"> {360x240} </Resolution>
        <Framerate type="IntegerSet"> [15,20,1] </Framerate>
      </UserQoSLevel>
    </QoSSubCategory>
  </QoSCategory>
</UtoATemplateProfile>
```

(b)

Figure 16: Service Quality Description for a Mobile Video-on-Demand Application: (a) Graphical Input, and (b) Generated XML Representation from the Graphical Input

these inputs into environment-independent and environment-dependent deployment descriptors, usable by the run-time meta-data execution during the application instantiation and adaptation.

6.2 Execution Process

To execute a developed QoS-aware application, a user utilizes the execution GUI, as shown in Figure 18(a), to specify the category and QoS requirements of the application. The execution GUI will automatically filter out the setup configurations that do not satisfy the user requirements. When the user clicks "execute," the run-time meta-data execution will consult the QoSCASpec, which is the environment-dependent deployment descriptor of the application, and try to select the best configuration to be instantiated according to the availability of service components, devices, and resources. Figure 18(b) shows an example of the application execution via the execution GUI. Note that other usages of the Q-Compiler such as GUIs for meta-data translations, IR instrumentation, component resource probing, are omitted due to limited space, and can be found in [35].

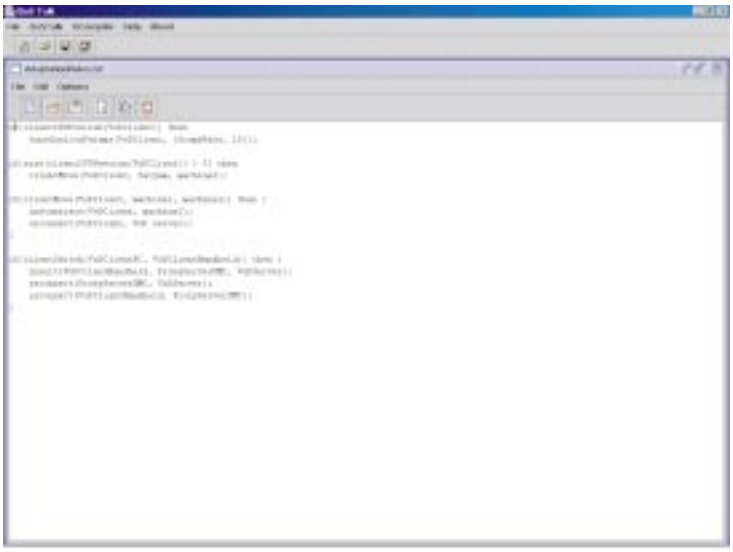


Figure 17: Adaptation Control Description for Mobile Video-on-Demand Application (Example)

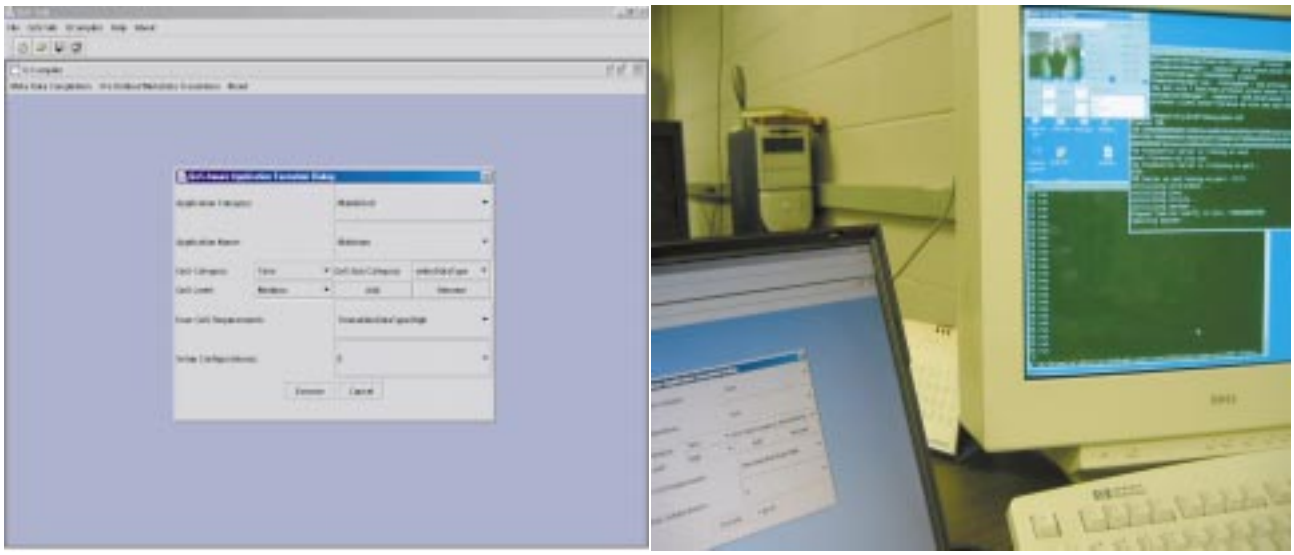
7 Validation

The implementation of the meta-data compiler is divided into two main parts: (1) the core of meta-data compiler including the meta-data translations, the high-level specification, and the meta-data compiler protocol are implemented in Java; and (2) the front-end of the run-time meta-data execution is implemented using idlj[43] and LuaJava[44] to communicate with Gaia QoS services (e.g., instantiation service with resource reservation) and Gaia services[9](e.g., component manager core, space repository, component repository) in the active space project[45].

The application test-bed is the mobile Video-on-Demand application consisting of four application service components: a user profile server, a VoD server, a VoD playback service, and a media transcoder. The application is represented by two setup configurations: (1) {*VoD server, VoD playback service*}, and (2) {*VoD server, media transcoder, VoD playback service*}, and {*user profile server, VoD playback service*}, assuming that the *VoD playback service* is generic. The QoS-enabling service associated with these configurations is DSRT, that is a CPU scheduling service.

The Gaia environment test-bed consists of three PCs: (1) Satyam is a Pentium III machine with a 700 MHz processor and 384 MB RAM,(2) Florence is a Pentium III machine with a 930

[9]Gaia is a distributed operating system for a ubiquitous smart room environment. It brings the functionality of an operating system to physical spaces. The Gaia kernel consists of a set of services: for example, context service, component repository, event manager, component manager core, etc.[45].



(a) (b)

Figure 18: QoS-Aware Application Execution: (a) Input QoS Requirements for Executing a Mobile Video-on-Demand Application, Specified by a User, and (b) Real Execution (Example)

MHz processor and 256 MB RAM, and (3) Casablanca is a Pentium III machine with a 930 MHz processor and 256 MB RAM. These nodes are connected via a 100 Mbps Ethernet. All PCs are running Windows 2000. The run-time meta-data execution is running on all nodes. The meta-data compiler is running on Satyam. We demonstrate concepts, design, and implementation of our meta-data compiler framework via two sets of the experiments. First, we measure performance and overhead of the meta-data compiler protocol. Second, we measure the service quality of the QoS-aware application developed via the meta-data compiler. Additional experiments, such as the overheads of component and configuration instantiations performed by the run-time meta-data execution integrated as a part of the Gaia environment, were reported in [46].

7.1 Overhead of Meta-Data Compiler Protocol

In this section, we measure the overhead of the environment-dependent compilation. In particular, we measure time used for generic service substitution in a specific deployment environment.

As the *VoD playback service* in the application test-bed is "generic," the meta-data compiler protocol needs to substitute this service with compatible services available in the deployment environment during the environment-dependent compilation. We utilize the component repository implemented in the Gaia project as the repository for available QoS-aware services in the deploy-

ment environment. In this experiment, the repository consists of two registered components: the *BitmapVoDPlayback* and the *MPEGIIVoDPlayback*, both of which implement the *PlaybackService* category. We measure the overhead of service substitution for setup configuration 1, and in both configurations of the mobile Video-on-Demand application, as shown in Figures 19(a) and 19(b), respectively. The overhead for the substitution includes (1) querying the component repository for the available service components satisfying the pre-defined constraints, such as the component category of the generic service component, and (2) performing the substitution using the result from the component repository. The average values of ten runs for querying the component repository and performing the substitution for configuration 1 are 40.0 and 72.1 ms, respectively. The average values of ten runs for the same metrics of both configurations are 42.0 and 88.1 ms, respectively.

Evaluation: The overheads for querying the component repository in both experiments are very similar because we use the same constraint in the queries. The overhead for performing the substitution depends on the number of setup configurations consisting of a "generic" service component. Since both configurations consist of the *VoD playback service*, the overhead of the substitution in Figure 19(b) is higher than the overhead of the substitution including only configuration 1 in Figure 19(a).

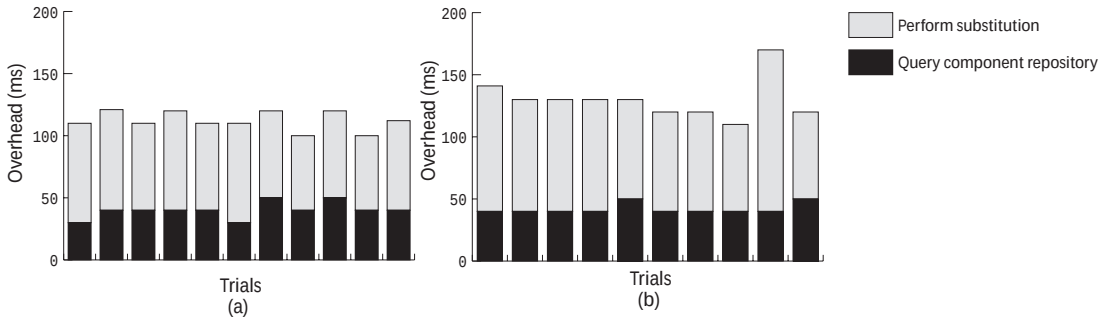


Figure 19: Overhead of Generic Service Component Substitution for (a) Setup Configuration 1, and (b) Both Setup Configurations, of Mobile Video-on-Demand Application

7.2 Service Quality of QoS-Aware Component with IR Translator

In this set of experiments, we respectively measure the effect of an IR translator on the service quality of a service component, and the overhead of the IR translator as additional layer of component communication.

7.2.1 Effect of IR Translator on Service Quality of MPEGIIVoDPlayback

In this experiment, we measure the service quality of the *MPEGIIVoDPlayback* in three scenarios: (1) *MPEGIIVoDPlayback* without QoS-enabling service (DSRT), (2) *MPEGIIVoDPlayback* with DSRT monolithically integrated, and (3) *MPEGIIVoDPlayback* composed with DSRT via an IR translator. The quality metric of the playback service is the intermediate delay time between two consecutive video frames. For all scenarios, during time interval A, an additional background process is introduced to heavily consume CPU of the client node.

Evaluation: As shown in Figure 20(a), the normal execution of the *MPEGIIVoDPlayback* without DSRT in the CPU-loaded environment cannot maintain its service quality. The intermediate delay is longer as shown in the interval A due to the loaded CPU. The intermediate delay is much shorter as shown in the interval B immediately after the additional process is terminated. Figure 20(b) shows that service quality of the *MPEGIIVoDPlayback* can be maintained with the help of DSRT even though the CPU is loaded during time interval A. Figure 20(c) shows that the introduction of IR translator does not effect the overall service quality, maintained by DSRT, of the *MPEGIIVoDPlayback*.

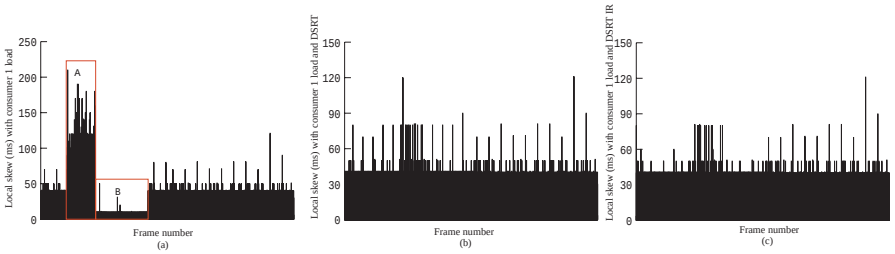


Figure 20: Service Quality of the MPEGIIVoDPlayback: (a) Normal Execution with Loaded CPU in Interval A, (b) Normal Execution with Monolithically Integrated DSRT and Loaded CPU in Interval A, (c) Normal Execution with DSRT, via an IR Translator, and Loaded CPU in Interval A

7.2.2 Intermediate Representation (IR) Overhead

Within this experiment, we measure the overhead of calling a function in a QoS-enabling service by the *MPEGIIVoDPlayback* in two scenarios: (a) the *MPEGIIVoDPlayback* directly calls an actual function provided by DSRT, and (b) the *MPEGIIVoDPlayback* calls the actual function of the DSRT via the *IR translator*. The overhead of calling each actual function in DSRT without passing

through the translator is less than 1 ms. The overheads of calling functions: *startRTRun()* and *YieldCPU()* through the *IR translator* of the DSRT are shown in Figure 21. The average values of ten runs are 33.1 and 49.4 ms, respectively. Note that each run that measures the overhead of calling the *YieldCPU()* function consists of 1748 iterations.

Evaluation: Even though the instrumentation of the *IR translator* of DSRT into the code of *MPEGIIVoDPlayback* costs tens of milliseconds overhead in calling the corresponding actual functions, it does not degrade the service quality of the component.

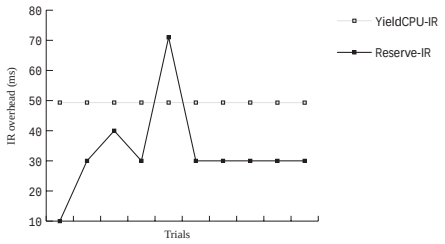


Figure 21: Overhead of Introducing IR Translator Between MPEGIIVoDPlayback and DSRT

8 Related Work

In this section, we discuss related work in four areas: component architecture, OMG MDA, tools for building distributed multimedia applications, and MPEG-21.

8.1 Component Architecture

Standard component architectures such as Enterprise JavaBeans[31], CORBA Component Model[32], COM/COM+[47] have been proposed for enabling fast development and deployment of distributed component-based applications. None of them, however, defines a model for the applications with QoS-aware descriptions as their first-class information. Our QoS-aware service models and categorizations can be used to extend these component architectures with QoS-related meta-data. It considers QoS as a fundamental objective in the development and deployment of an application.

8.2 OMG MDA

Model Driven Architecture(MDA)[48] provides models such as PIM (Platform Independent Models) and PSM (Platform Specific Models) that can be used for modelling an application from different

views and abstractions. It enables the separated responsibility among groups of people who design, develop, and deploy the application. For instance, an architect would focus on creating the architectural and platform independent model for the application. The middleware designer then uses a UML profile for a specific middleware to model specific aspects of the abstract system. A programmer, then, uses the model as well as the generated specific aspects to add codes to complete the value-added business logic for the application. Explicit mapping relations across models are available to enable potential automation of PSM generation as well as to ease the integration.

While both OMG MDA and our framework propose models and mappings for application development and deployment, the overall concepts, contexts and objectives are different. First, while OMG MDA mainly focuses on models and their relations for enabling the clear responsibility among workers in different phases of software development and deployment, we propose different sets of models that mainly enable QoS-aware interoperability among connected components. Second, while a UML profile for specific middleware is analogous to a translation specification in our framework, it does not focus on QoS mappings and semantics interoperability. Third, while the refinement of PIM to PSM in OMG MDA seems similar to our environment-independent and environment-dependent compilations, our protocol mainly focuses on automating the generation of QoS-aware configurations and enabling the instantiation of QoS-aware applications into distributed environments with the help of run-time meta-data execution.

Recent research results such as [49, 50, 51] have been proposed to introduce QoS into specific middlewares, using the OMG MDA approach. For instance, [49] adds QoS into the .NET Remoting, which is a middleware of the Microsoft's .NET framework. These research results provide configurable QoS-aware middlewares that could be considered reusable QoS-enabling services in our framework.

8.3 Tools for Building Distributed Multimedia Applications

Different development tools have been developed for building multimedia applications. For example, the Continuous Media Toolkit (CMT)[52], developed at UC Berkeley, provides a programming environment for enabling fast development of continuous media applications. VuSystem[53] provides a programming environment that separates codes of control and user interfaces from media

manipulation. Ooi et al develops a multimedia software library, called Dali [54], that includes a set of intermediate level abstractions between C and conventional libraries. The StreamIt [55] project provides a special-purpose language to improve programmer productivity and program robustness within the streaming domain. These research results provide libraries or specific programming languages that enable specific aspects of multimedia programming.

From the object-oriented or component-based side, software toolkits [56, 57, 58] have been proposed to help the application developer to develop a distributed multimedia applications flexibly and more easily. For example, DAVE [56] provides a plug-and-play programming paradigm, which allows the application developer to connect the distributed objects or devices forming the distributed application. SCOOT [57] provides the reliable multimedia collaboration, based on the object-oriented approach. In mash toolkit[58], Mccanne et al propose a common infrastructure, that allows an application developer to utilize different media and protocol objects from different research groups to develop a distributed multimedia application flexibly. Comparing to [58], our work distinguishes itself by focusing on defining QoS-aware service models, meta-data translation models, and meta-data compilations that enable the automation of QoS-aware semantics and interface interoperability among connected components forming a QoS-aware application.

8.4 MPEG-21

MPEG-21[59] aims to provide open framework for multimedia delivery and consumption. Two fundamental entities of MPEG-21 are digital item that concerns about unit of distribution and transaction and the interaction of users with digital items. Users should be able to access, exchange, consume and manipulate digital items in transparent and interoperable manner. To fulfil the objective, MPEG-21 defines several related entities such as Data Item Declaration that defines elements useful for modelling a data item, Data Item Identification that considers how to uniquely identify data items, their types and interconnection, Intellectual Property Management and Protection that defines interoperable framework for protecting and managing intellectual property, and Rights Expression Language that declares rights and permissions for digital contents based on Rights Data Dictionary. Besides these entities, MPEG-21 specifies Data Item Adaptation that emphasizes universal multimedia access and enables transparent and interoperable content creation

and sharing with quality guaranteed. The concepts and design of the Q-Compiler complement aspects of MPEG-21 objective. Especially, semantics translators resulted from the compilations of the meta-data translations enable semantics interoperability for universal multimedia access. Also, the meta-data compiler protocol and the run-time meta-data execution enable the ubiquitous and dynamic deployment of multimedia contents and services.

9 Conclusion

The availability of reusable multimedia components and QoS-enabling services bring new challenges for enabling the flexible and efficient development and deployment of component-based multimedia applications, deployable in ubiquitous environments with QoS guarantees. In this paper, we present a novel meta-data compiler for multimedia domain. Key contributions of the framework include:

- *a set of QoS-aware models* that supports the flexible integrated reuse of multimedia and QoS-enabling services for composing a distributed, multimedia application in ubiquitous deployment environments;
- *a set of extensible meta-data translation models* and their compilations that enable semantic interoperability among application-specific QoS requirements, QoS-enabling service provisions, and resource requirements;
- *a set of high-level specifications* that hides the complexity of QoS-aware programming, and allows an application developer to develop a multimedia application with QoS by customizing its specifications;
- *a meta-data compiler protocol* with environment-independent and environment-dependent meta-data compilations that enable QoS consistency of a composed application, and promotes configurability and flexibility of its deployment;
- *a run-time meta-data execution* that provides the run-time support for QoS compilation (e.g., for probing, interacting with underlying services in a specific deployment environment) and the usage of meta-data compilation results for running an application.

Although the framework is presented via multimedia domain, we anticipate that the fundamental concepts and design will be applicable in other application-specific domains.

References

[1] 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.

[2] M. Shankar, M. DeMiguel, and J. Liu. An end-to-end qos management architecture. *In Proceedings of IEEE RTAS'99*, June 1999.

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

[4] 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.

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

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

[8] 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.

[9] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, pages 73–82, October 1990.

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

[11] B. Ravindran. Engineering dynamic real-time distributed systems: architecture, system description language, and middleware. *IEEE Transactions on Software Engineering*, 28(1):30–57, January 2002.

[12] ATM Forum. Atm user-network interface specification v3.1. 1993.

[13] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp) – version 1 functional specification. *RFC 2205, Proposed Standard.*, September 1997.

[14] K. Kim and K. Nahrstedt. *QoS Translation and Admission Control for MPEG Video*, pages 359–362. Chapman and Hall, November 1997.

[15] A. K. Viswanathan. *Design and Evaluation of a CPU-aware Communication Broker for RSVP-based Networks.* Master thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 2000.

[16] K. Fukuda, N. Wakamiya, M. Murata, and H. Miyahara. Qos mapping between user's preference and bandwidth control for video transport. *In Proceedings of the 5th IFIP International Workshop on Quality of Service (IWQoS 97)*, pages 291–302, May 1997.

[17] T. Yamazaki and J. Matsuda. On qos mapping in adaptive qos management for distributed multimedia applications. *In Proceedings of the ITC-CSCC'99*, pages 1342–1345, July 1999.

[18] H. Knoche and H. de Meer. Quantitative qos-mapping: A unifying approach. *In Proceedings of the 5th IFIP International Workshop on Quality of Service (IWQOS'97)*, pages 347–358, May 1997.

[19] L. A. DaSilva. Qos mapping along the protocol stack: Discussion and preliminary results. *In Proceedings of the IEEE International Conference on Communications*, pages 713–717, June 2000.

[20] B. Li and K. Nahrstedt. Qualprobes: Middleware qos profiling services for configuring adaptive applications. *In Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms*, pages 256–272, April 2000.

[21] T. F. Abdelzaher. An automated profiling subsystem for qos-aware services. *In Proceedings of the 6th IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, pages 208–217, 2000.

[22] J.-F. Huard and A.A. Lazar. On end-to-end qos mapping. *In Proceedings of the 5th IFIP International Workshop on Quality of Service (IWQOS'97)*, May 1997.

[23] 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.

[24] Object Management Group Inc. Audio/video stream specification. *Online documentation at http://www.omg.org/cgi-bin/apps/doc?formal/00-01-03.pdf*, January 2000.

[25] T. Plagemann, F. Eliassen, B. Hafskjold, T. Kristensen, R. H. Macdonald, and H. O. Rafaelsen. Flexible and extensible qos management for adaptable middleware. *In Proceedings of the International Workshop on Protocols for Multimedia Systems (PROMS 2000)*, October 2000.

[26] G. Coulson and M. W. Clarke. A distributed object platform infrastructure for multimedia applications. *Computer Communications*, 21(9):802–818, July 1998.

[27] N. Parlavantzas, G. Coulson, M. Clarke, and G. Blair. Towards a reflective component based middleware architecture. *in Workshop on Reflection and Metalevel Architectures*, June 2000.

[28] A. Ulbrich, T. Weis, K. Geihs, and C. Becker. DotQoS – a QoS extension for .NET remoting. In *Proceedings of International Workshop on Quality of Service*, pages 363–380, June 2003.

[29] C. Becker, G. Schiele, H. Gubbels, and K. Rothemel. Base - micro-broker based middleware for pervasive computing. *In Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications*, pages 443–451, March 2003.

[30] C. Bekcer, M. Handte, G. Schiele, and K. Rothemel. Pcom - a component system for pervasive computing. *In Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications*, pages 67–76, March 2004.

[31] Sun Microsystems. Enterprise javabeans specification, version 2.1, final release. *Online documentation at http://java.sun.com/products/ejb/docs.html*, November 2003.

[32] Object Management Group Inc. Corba component model, v3.0. *Online documentation at http://www.omg.org/cgi-bin/apps/doc?formal/02-06-65.pdf.*

[33] Mary Kirtland. The com+ programming model makes it easy to write components in any language. *Microsoft System Journals, Online documentation at http://www.microsoft.com/msj/1297/complus2/complus2.aspx*, December 1997.

[34] J. W.S. Liu, K. Nahrstedt, D. Hull, S. Chen, and B. Li. Epiq qos characterization, draft version. July 1997.

[35] D. Wichadakul. *Q-Compiler: Meta-Data QoS-Aware Programming and Compilation Framework*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 2003.

[36] J. M. Purtilo and J. M. Atlee. Module reuse by interface adaptation. *In SoftwarePractice and Experience*, (6):539–556, June 1991.

[37] K. Nahrstedt and J. Smith. The qos broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[38] A. Richards, G. Rogers, M. Antoniades, and V. Witana. Mapping user level qos from a single parameter. *In Proceedings of the International Conference on Multimedia Networks and Services (MMNS'98)*, November 1998.

[39] N.Nishio and H.Tokuda. Simplified method for session coordination using multi-level qos specification and translation. *In Proceedings of the 5th IFIP International Workshop on Quality of Service (IWQOS'97)*, pages 335–346, May 1997.

[40] D. Wichadakul, K. Nahrstedt, X. Gu, and D. Xu. 2kq+: An integrated approach of qos compilation and component-based, run-time middleware for the unified qos management framework. *In Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms*, pages 373–394, November 2001.

[41] X. Gu, K. Nahrstedt, R. N. Chang, and C. Ward. Qos-assured service composition in managed service overlay networks. *In Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 194–203, May 2003.

[42] J. Jin and K. Nahrstedt. Large-scale service overlay networking with distance-based clustering. *In Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, pages 394–413, June 2003.

[43] Inc. Sun Microsystems. Java idl technology documentation. *Online documantation at http://java.sun.com/j2se/1.4.2/docs/guide/idl/.*

[44] C. Cassino, R. Ierusalimschy, and N. Rodriguez. Luajava - a scripting tool for java. *PUC-RioInf.MCC02/99*, February 1999.

[45] M. Romn, C. K. Hess, A. Ranganathan, P. Madhavarapu, B. Borthakur, P. Viswanathan, R. Cerqueira, R. H. Campbell, and M. Dennis Mickunas. Gaiaos: An infrastructure for active spaces. *Technical Report UIUCDCS-R-2001-2224 UILU-ENG-2001-1731, Universiy of Illinois at Urbana-Champaign*, 2001.

[46] D. Wichadakul, X. Gu, and K. Nahrstedt. A programming framework for quality-aware ubiquitous multimedia applications. *In Proceedings of ACM Multimedia*, pages 631–640, December 2002.

[47] Microsoft corporation. The com specification. *Online documentation at http://www.microsoft.com/com/resources/comdocs.asp.*

[48] J. Miller and J. Mukerji (Editors). Mda guide version 1.0.1. *Online documentation at http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf*, June 2003.

[49] T. Weis, A. Ulbrich, K. Geihs, and C. Becker. Quality of service in middleware and applications: A model-driven approach. In *Proceedings of the 8th International Enterprise Distributed Object Computing Conference.*

[50] A. Gokhale, D. Schmidt, T. Lu, B. Natarajan, and N. Wang. Cosmic: An mda generative tool for distributed real-time and embedded applications. *In Proceedings of Workshop on Model-driven Approaches to Middleware Applications Development*, June 2003.

[51] D. Simmonds, S. Ghosh, and R. B. France. An mda framework for middleware transparent software development and quality of service. *In Proceedings of Workshop on QoS in Component-Based Software Engineering*, June 2003.

[52] K. Mayer-Patel and L. A. Rowe. Design and performance of the berkeley continuous media toolkit. *in Multimedia Computing and Networking, Proc. SPIE 3020*, pages 194–206, 1997.

[53] C.J. Lindblad and D.L. Tennenhouse. The vusystem: A programming system for compute-intensive multimedia. *IEEE Journal of Selected Areas in Communication*, pages 1298–1313, September 1996.

[54] W.T. Ooi, B. Smith, S. Mukhopadhyay, H. H. Chan, S. Weiss, and M. Chiu. Dali : A Multimedia Software Library. *SPIE Multimedia Computing and Networking*, January 1999.

[55] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. *In Proceedings of the International Conference on Compiler Construction*, 2002.

[56] R. F. Mines, J. A. Friesen, and C. L. Yang. Dave: A plug-and-play model for distributed multimedia application development. *In Proceedings of the ACM Multimedia Conference*, pages 59–66, 1994.

[57] E. Craighill, M. Fong, K. Skinner, R. Lang, and K. Gruenefeldt. Scoot: An object-oriented toolkit for multimedia collaboration. *In Proceedings of the ACM Multimedia Conference*, pages 41–49, 1994.

[58] S. McCanne, E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T. Tung, D. Wu, and B. Smith. Toward a common infrastructure for multimedia-networking middleware. *In Proceeding of the 7th International Workshop on Networking and Operating System Support for Digital Audio and Video*, pages 39–49, May 1997.

[59] J. Bormans and K. Hill (Editors). Mpeg-21 overview v.5. *Online documentation at http://www.chiariglione.org/mpeg/standards/mpeg-21/mpeg-21.htm*, October 2002.