A Translation System for Enabling Flexible and Efficient Deployment of QoS-Aware Applications in Ubiquitous Environments*

Duangdao Wichadakul and Klara Nahrstedt

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

Abstract. Ubiquitous Quality of Service(QoS)-aware applications, such as e-business or multimedia delivery are becoming available anywhere anytime. In the past decade, also QoS-oriented middleware services, assisting QoS-aware applications with different aspects of QoS provisions, have been proposed. Assuming the availability of these middleware services, in this paper, we present the *application to middleware service translation* system. This system helps an application developer to develop a QoS-aware application which can be deployed flexibly and efficiently in ubiquitous environments with different available middleware services. We introduce the middleware abstraction layer (MAL) between the application view of middleware and the specific middleware implementations. The translation system assists the QoS-aware application in two phases: (1) environment-independent translation, and (2) environment-dependent translation. The first phase maps the QoS-aware application to configurations of middleware services without indication of specific implementations, to satisfy the qualitative QoS requirements. Its result is the portable MAL representation. The second phase helps the application developer to customize the MAL representation within a specific deployment environment. It deals with (a) mapping of MAL representation into configurations of specific middleware implementations, and (b) mapping of application quantitative QoS requirements into specific middleware implementation's expected parameters. Our translation system facilitates the rapid growth of QoS-aware applications in the ubiquitous environments.

1 Introduction

The QoS-aware applications such as e-business, audio/video streaming, world wide web, and health care system are becoming ubiquitous. Users can instantiate and access these applications anytime, anywhere and use any computing devices.

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

J. Bishop (Ed.): CD 2002, LNCS 2370, pp. 210–221, 2002.
© Springer-Verlag Berlin Heidelberg 2002

In the past decade, several QoS-oriented middleware services dealing with system resource management [1,2,3], different types of adaptations [4,5], or different types of communications [6,7,8], have been proposed to assist in QoS provision for different applications. Assuming the availability of these QoS-oriented middleware services in different ubiquitous computing environments, the challenging question is: "*How to develop a QoS-aware application which can be deployed flexibly and efficiently in different environments, with different available middleware services, and satisfy acceptable quality of service(QoS)?*"

Developing a QoS-aware application like this is not straight forward due to the following limitations. First, available middleware services are platform-specific, and implemented in different languages, with specific semantics and expected parameters. Second, they are designed only to run for specific classes of applications (e.g., multimedia), or to handle particular aspects of QoS provisions (e.g., real-time streaming, real-time messaging, reliable messaging, adaptability, security), with different levels of QoS provisions (e.g, hard, soft, best effort).

Due to these limitations, the problems of developing and deploying such a QoS-aware application are: (1) the application developer can only *statically* decide and deploy a specific configuration of middleware implementations; (2) the application developer needs to well understand characteristics of each selected underlying middleware implementation, and knows how to appropriately map application QoS requirements into its specific semantics, and expected parameters; and (3) the QoS-aware application is bound to deployment environment where the selected middleware implementation exists, for which the application was developed.

Our approach to the problems is to investigate an *application to middleware service translation system* which will map and bind application service components in a component-based application to appropriate middleware services in a configurable and portable fashion. We introduce *middleware abstraction layer (MAL)* between application view of middleware and the specific middleware implementations. MAL abstracts from individual middleware implementations, and represents a high-level functional view to the application. The translation system performs the mappings between application and middleware services in three main steps (see Fig. 1): (1) mapping of application service components to configurations of generic middleware services (e.g. CPU scheduling service), resulting in MAL representation; (2) mapping of each configuration of generic middleware services in MAL representation to possible configurations of specific middleware implementations (e.g., DSRT [2] for CPU scheduling service) available in specific deployment environment; and (3) mapping of application service components' quantitative QoS parameters (e.g., frame rate, frame size) to specific semantics, and expected parameters of specific middleware implementation (e.g., expected cycle time, and computation time of DSRT). The application to middleware service translation system is a core subsystem of our overall QoS compilation framework (Q-Compiler)[9].

The rest of the paper is organized as follows. In Sect. 2, we give the overview of our QoS compilation system (Q-Compiler) to place in context the application

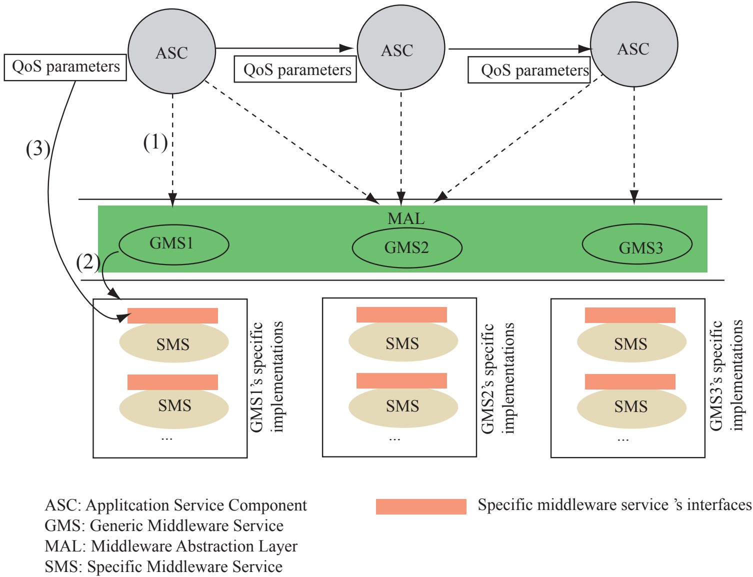

Fig. 1. Mapping Steps in Application to Middleware Service Translation System

to middleware service translation system. In Sect. 3, we present the architecture of the application to middleware service translation, its main entities which enable the flexible and efficient deployment of QoS-aware application in different deployment environments. In Sect. 4, we discuss the related work. Finally, we draw conclusions in Sect. 5.

2 Q-Compiler Overview

Q-Compiler is a meta-level compilation system, which takes QoS specifications of a distributed component-based QoS-aware application as input. Then, assisted by a run-time middleware such as $2K^Q$[1], it translates QoS specifications in multiple phases into end-to-end system/resource configurations. We will briefly discuss the QoS specifications as well as individual compilation phases to provide sufficient background information for discussing our application-to-middleware service translation.

2.1 QoS Specifications

QoS specifications (Fig. 2) are the *"source code"* of the Q-Compiler. An application developer implements a QoS-aware application by customizing these

[1] Run-time $2K^Q$ middleware [9] comprises of a group of QoS-related management services forming the execution environment for instantiating and managing a distributed component-based QoS-aware application.

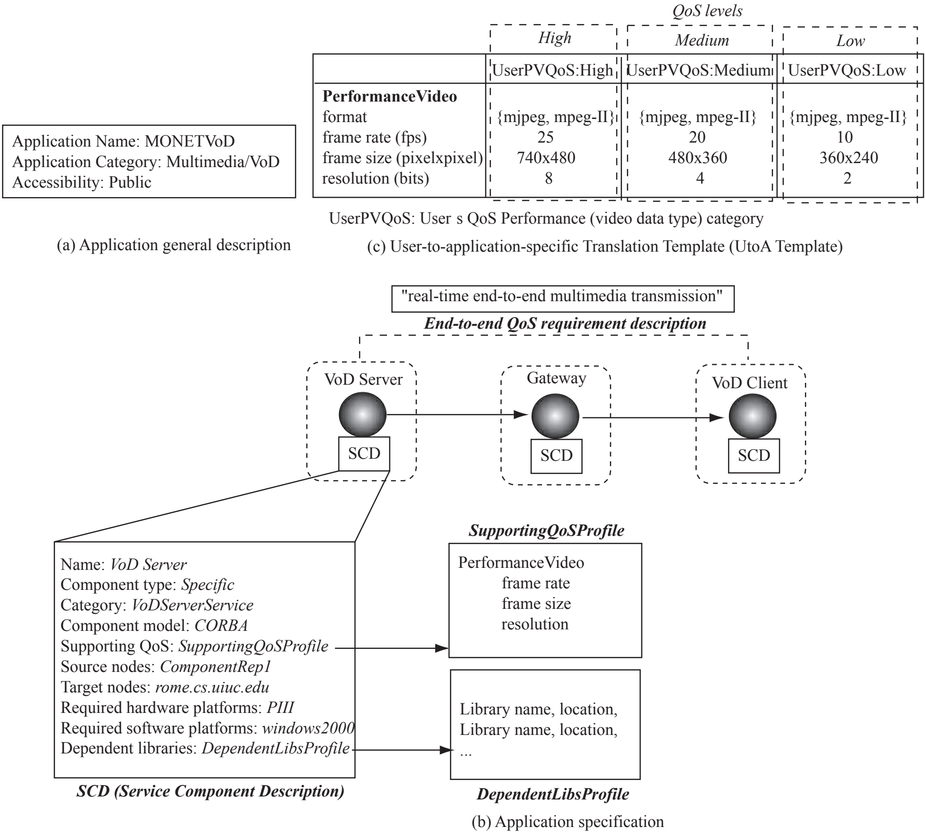

Fig. 2. QoS Specifications for a Video-on-Demand Application (Example)

specifications. QoS specifications include: (a) *application general description*, (b) *application specification*, and (c) *user-to-application-specific translation template (UtoA template).*

Application general description allows the application developer to specify the general information about the application, such as application name, application category, and its accessibility.

Application specification includes the *application functional dependency graph*, which is labeled with an *end-to-end QoS-requirement description*, and composes of different *application service components (ASCs).* The functional dependency graph allows the application developer to express an application via the composition of specific, generic, or composite application service components, flexibly. The graph is *fully-defined* if all its application service components are specific. Otherwise, it is *partially-defined.* The *end-to-end QoS-requirement description* allows the application developer to label the application functional dependency graph or sub-graphs with different specific *end-to-end qualitative* QoS require-

ments. Each *application service component* is associated with a *service component description* which allows the application developer to specify detailed component information (e.g., name, hardware/system software requirements, resource requirements). Each service component description is associated with the *supporting QoS profile*, and the *dependent libraries profile*. The supporting QoS profile consists of QoS categories and their *quantitative* QoS dimensions the application service component supports. The dependent libraries profile consists of a list of application service component's dependent libraries, and their locations or pointers to their locations.

User-to-application-specific translation template (UtoA template) defines the mapping between different user QoS levels and corresponding application specific QoS categories and their dimensions[2].

2.2 Q-Compiler Model

Q-Compiler (see Fig. 3) consists of three phases: (Phase I) *symbolic configuration translation*; (Phase II) *application to middleware service translation*; and (Phase III) *distributed multi-resource translation*.

(Phase I) takes QoS specifications and compiles the partially-defined application functional dependency graph into **symbolic QoS configurations** where each configuration is a fully-defined functional graph of application components with consistent end-to-end QoS. This phase, modeled as a constraint satisfaction problem [11], generates possible delivery forms for the application to be executable in ubiquitous environments. (Phase II) consists of two sub phases: (Phase II.a) compiles symbolic QoS configurations into **MAL representation**. MAL representation represents the associations between each symbolic QoS configuration and possible configurations of generic middleware services. (Phase II.b) compiles each association in MAL representation into **application-middleware association**, representing associations between each symbolic QoS configuration and possible specific middleware implementations available in the deployment environment. (Phase III) maps each association in the application-middleware association into distributed multi-resources requirements representing a **system QoS configuration**.

Phase I and Phase II.a are *environment-independent* because they perform translations logically without concerning about physical environment constraints. Phase II.b and Phase III are *environment-dependent* because they perform translations corresponding to physically available middleware implementations in specific deployment environment. The environment-independent translations help developing a generic, portable QoS-aware application. The environment-depend-

[2] *QoS category* and *QoS dimension* are part of *QoS specifications* to characterize non-functional properties of an application. A *QoS dimension* defines a qualitative or quantitative attribute for a *QoS category* [10]. For example, QoS dimensions "format", "frame rate", "frame size" and "resolution" are attributes of QoS category "PerformanceVideo". Note that in our compilation framework, we limit a QoS dimension to a *quantitative* attribute.

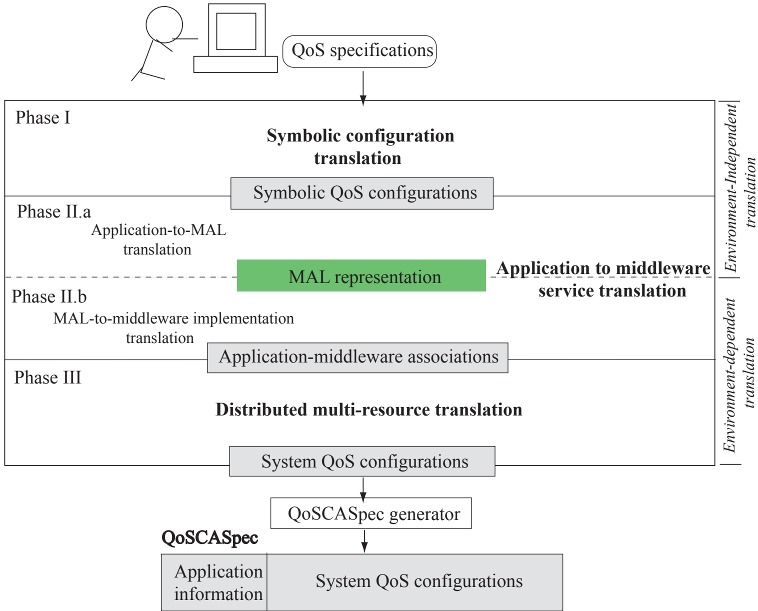

Fig. 3. Q-Compiler Model

ent translations help customizing the QoS-aware application with specific deployment environment.

The compiled result of the Q-Compiler, in a specific deployment environment, is *QoS-aware Component-based Application Specification (QoSCASpec)*. QoSCASpec includes (1) application description, and (2) a set of system QoS configurations. QoSCASpec is QoS-enabled meta information which can be utilized flexibly by a run-time middleware such as $2K^Q$, during the application instantiation, and adaptation (reconfiguration) in a specific deployment environment. As discussed in Sect. 1, in this paper, we mainly focus on the mapping problems in the Q-Compiler Phase II.

3 Application to Middleware Service Translation

Application-to-middleware service translation deals with the following problems: (1) "how to develop a QoS-aware application which is independent from specific deployment environment and specific middleware implementations?", (2) "how to flexibly and efficiently deploy this QoS-aware application into different environments with different available specific middleware implementations and satisfy acceptable quality of service (QoS)?" To solve these problems we present the architecture of the application-to-middleware service translation (see Fig. 4), which consists of two main processes: (1) *application-to-MAL translation*, and (2) *MAL-to-middleware implementation translation*. We describe these two processes in details, in the following section.

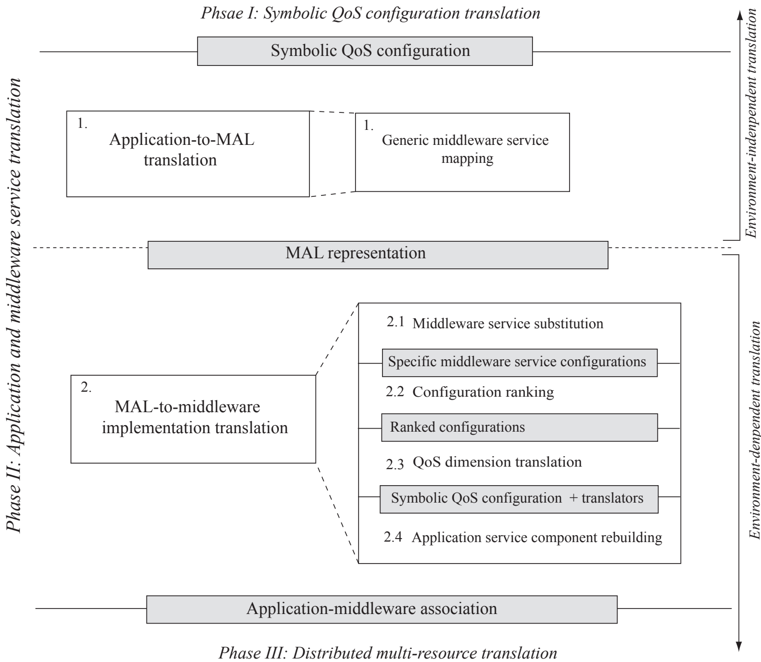

Fig. 4. Application to Middleware Service Translation Architecture

3.1 Application-to-MAL Translation

Application-to-MAL translation, corresponding to mapping (1) in Fig. 1, helps an application developer to associate a QoS-aware application with generic middleware services, independent of specific deployment environment and specific middleware implementations. The core of the application-to-MAL translation is the labeling decision engine. The *Labeling decision engine* is based on *mapping rules*, as shown in Fig. 5. The mapping rules are pre-defined and relate end-to-end QoS requirements to generic middleware services. QoS requirements are predicates, and operations are suggestions of possible configuration(s) of generic middleware services, their placements, and their association to application service components in the symbolic QoS configuration.

The result of application-to-MAL translation is MAL representation, as shown in Fig. 6, for the Video-on-Demand (VoD) application (shown in Fig. 2). MAL representation consists of (i) symbolic QoS configurations and their associations with generic middleware services[3], (ii) application service components' descriptions, and (iii) ranking policies.

[3] In our current system, we classify the available middleware implementations into categories (e.g., CPU scheduling service, bandwidth broker service, fuzzy-control

if subgraph is labeled with "real-time end-to-end multimedia transmission"
then (1) all application service components in the subgraph need local CPU broker service;
(2) all application service components dealing with transmission among distributed machines need bandwidth broker service.
...

Fig. 5. Generic Middleware Service Mapping Rules

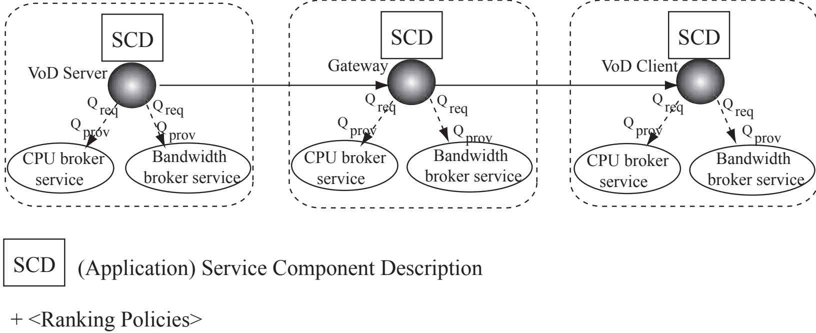

Fig. 6. MAL representation for VoD's Symbolic QoS Configuration (Example)

3.2 MAL-to-Middleware Implementation Translation

MAL-to-middleware implementation translation helps an application developer to customize and deploy a QoS-aware application, represented as MAL presentation, into different deployment environments, flexibly and efficiently. The translation is based on the following steps. **Middleware Service Substitution**, corresponding to mapping (2) in Fig. 1, determines all possible configurations of specific middleware implementations[4], in the considered deployment environment, for each generic middleware service configuration in MAL representation. The substitution process is modeled as a constraint satisfaction problem[11], described as follows. Considering MAL representation in Fig. 6, each generic middleware service is considered as a symbolic variable. Each variable will be substituted by a set of specific middleware implementations satisfying specific requirements (e.g., hardware/system software requirements). An edge between an application service component and a generic middleware service in the graph represents the compatibility constraint between their QoS requirements (Q_{req}) and QoS provisions (Q_{prov}). Q_{req} is the vector of application-specific QoS categories and their dimensions. Q_{prov} is the vector of service quality that the

adaptation-based middleware service, real-time messaging service) corresponding to their functionality. The generic middleware service represents a specific category of the implementations.

[4] We assume that a specific middleware implementation advertises itself with detailed descriptions, in a directory service, available in specific deployment environment.

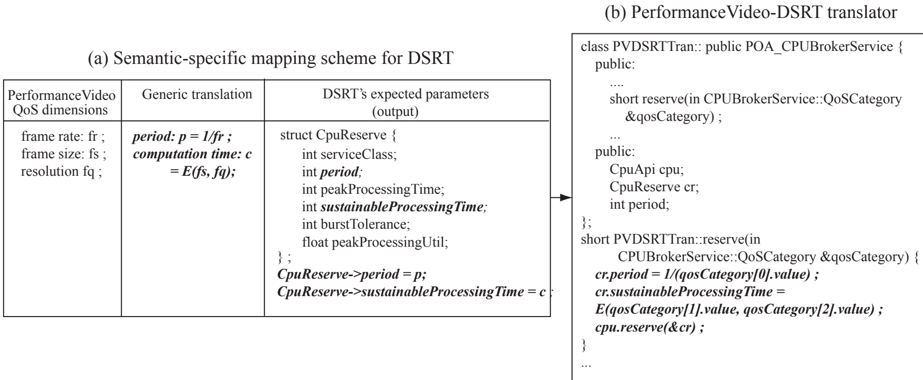

Fig. 7. Semantic-Specific Mapping Scheme for DSRT and PerformanceVideo-to-DSRT Translator

specific middleware implementation can provide. Q_{req} and Q_{prov} are compatible if Q_{prov} satisfies Q_{req}. Note that the satisfiability check needs the translation between Q_{req}'s and Q_{prov}'s semantics. This translation deploys a semantic-specific mapping scheme, as shown in Figure 7.(a).

Configuration Ranking ranks among specific configurations for each generic middleware service configuration. The ranking algorithm weights each specific middleware implementation in a configuration using constraints[5] such as its requirement of system resources (e.g., memory footprint, CPU processing time), its supporting levels of QoS provisions (e.g., soft, hard, best effort), and its instantiation overhead (e.g., dynamic downloading) if unavailable in a specific location. Note that we can vary weights among constraints, corresponding to different major concerns, and generate multiple ranked specific configurations for each generic middleware service configuration.

QoS Dimension Translation, corresponding to mapping (3) in Fig. 1, bridges the gap between application-specific *quantitative QoS requirements* (specified via QoS category and its dimensions), and a specific middleware implementation's semantics, and expected parameters. QoS dimension translation determines a proper translator for each pair of application-specific QoS dimensions and specific middleware implementation. A proper translator can be built by a tool of the Q-Compiler, or manually built by an application developer. A specific translator (see Fig. 7.(b)) wraps up a specific middleware implementation's interfaces with a *semantic-specific translation scheme* (see Fig. 7.(a)). A semantic-specific mapping scheme represents an analytical translation or a mapping from application-specific QoS dimensions into specific middleware implementation's semantics and expected parameters.

[5] We assume that these information can be queried from the directory service available in the deployment environment.

Application Service Component Rebuilding process helps instrumenting and rebuilding application service component's code with specific translator's interfaces. The rebuilding process performs only if the application service component's rebuilt version with the instrumented translators' interfaces is unavailable in the *Rebuilt Application Service Component Repository*[6]. This repository helps to reduce the overhead in the code instrumentation, and rebuilding processes.

The results of the MAL-to-middleware implementation translation are *application-middleware associations* which will be passed to the third phase of the Q-Compiler for distributed resource translation. The application-middleware associations with their resource requirements compose *system QoS configurations* and are represented as the *QoS-aware Component-based Application Specification (QoSCASpec).* During the application instantiation and execution, the underlying run-time middleware such as $2K^Q$, will select the best system QoS configuration from the application's QoSCASpec, for application setup and adaptation (reconfiguration), corresponding to the current deployment environment, and resource availability.

4 Related Work

Besides the related work in the area of QoS-oriented middleware services discussed in Sect. 1, MAL representation, introduced in this paper, shares the same ideas as of the EJB's deployment descriptor [12], and CCM's descriptors [13](e.g, CORBA software, CORBA component, and component assembly descriptors), and shares the similar idea as of COM+'s attributed-based or declarative programming [14]. In the area of QoS mapping and QoS translation, QoS translations, based on analytical functions, are proposed. For example, in our group, in [15] Nahrstedt et al. propose a translation from a multimedia application's QoS parameters into transport subsystem's QoS parameters, in [16] Kim et al. propose a translation from MPEG video parameters into CPU requirements, and in [3], Viswanathan proposes the analytical translations from MPEG video parameters into CPU requirements required by the transportation task for video transmission, and into network bandwidth requirements. In [17], Dasilva presents a framework for predicting end-to-end QoS at the application layer based on mapping of QoS guarantees across layers in the protocol stack. Our translation system utilizes some of these translations as semantic-specific mapping schemes.

5 Conclusion

Emerging ubiquitous QoS-aware applications call for a new developing tool which can help an application developer to develop a QoS-aware application which can be deployed efficiently and flexibly in the ubiquitous environments with the satisfactory QoS. Assuming the availability of QoS-oriented middleware services,

[6] *Rebuilt Application Service Component Repository* collects the version(s) of the application service components built with specific translators.

in this paper, we present a translation system between the application and middleware services as an enabling tool. Key features of our translation system are: (1) helping an application developer to develop a component-based QoS-aware application with the broader aspects (e.g., real-time, reliability, availability, security, or their combination) of service quality; and (2) dealing mainly with the configurability of different generic middleware services and their specific implementations, which can provide the service quality according to the qualitative and quantitative QoS requirements for the application in different ubiquitous environments. We believe that the availability of the architecture like our translation system will facilitate the rapid growth of QoS-aware applications in ubiquitous environments.

We are prototyping the application-to-middleware service translation, as the enhancement of the implemented Q-Compiler, described in [9]. Experimental preliminary results of the Q-Compiler and the run-time $2K^Q$ middleware can be found in [9]. We plan to evaluate the effectiveness of this approach, as part of the active space project [18].

References

1. 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.
2. 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.
3. 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.
4. 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.
5. R. Vanegas, J.A. Zinky, J.P. Loyall, D.A. Karr, R.E. Schantz, and D.E. Bakken. Quo's runtime support for quality of service in distributed objects. *In Proc. of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 1998)*, pages 207–222, September 1998.
6. 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.
7. Object Management Group Inc. Corba 2.5 - chapter 22 - corba messaging. *online documentation at http://www.omg.org/cgi-bin/doc?formal/01-09-26*, September 2001.
8. Object Management Group Inc. Corba 2.5 - chapter 24 - real-time corba. *online documentation at http://www.omg.org/cgi-bin/doc?formal/01-09-28*, September 2001.
9. 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 Proc. of IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001.

10. 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.
11. E. Tsang. *Foundations of Constraint Satisfaction*, chapter Introduction. Academic Press, 1993.
12. Sun Microsystems. Enterprise javabeans tm specification, version 2.0. *online documentation at http://java.sun.com/Download5*, August 2001.
13. Object Management Group Inc. Corba 3.0 new components chapters. *online documentation at ftp://ftp.omg.org/pub/docs/ptc/01-11-03.pdf*, November 2001.
14. 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/com/wpaper/default.asp*, December 1997.
15. 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.
16. 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, November 1997.
17. L. A. DaSilva. Qos mapping along the protocol stack: Discussion and preliminary results. *In Proceedings of IEEE International Conference on Communications*, pages 713–717, June 2000.
18. Manuel Romn, Christopher K. Hess, Anand Ranganathan, Pradeep Madhavarapu, Bhaskar Borthakur, Prashant Viswanathan, Renato Cerqueira, Roy 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.*