QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments

Klara Nahrstedt, Dongyan Xu, Duangdao Wichadakul
Department of Computer Science
University of Illinois at Urbana-Champaign
{*klara,d-xu,wichadak*}*@cs.uiuc.edu*

Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
bli@eecg.toronto.edu

Abstract

Middleware systems have emerged in recent years to support applications in heterogenous and ubiquitous computing environments. Specifically, future middleware platforms are expected to provide *Quality-of-Service* (QoS) support, which is required by a new generation of QoS-sensitive applications such as media streaming and e-commerce. This article presents four key aspects of a QoS-aware middleware system. First, *QoS specification* to allow description of application behavior and QoS parameters; Second, *QoS translation and compilation* to translate specified application behavior into candidate application configurations for different resource conditions; Third, *QoS setup* to appropriately select and instantiate a particular configuration; Finally, *QoS adaptation* to adapt to runtime resource fluctuations. We also provide a comparison of existing QoS-aware middleware systems in these four aspects.

Keywords: Quality of Service (QoS), Middleware, Ubiquitous Computing

I. Introduction

A new generation of distributed applications, such as telemedicine and e-commerce applications, are being deployed in heterogenous and ubiquitous computing environments. These applications are expected to deliver adaptive and satisfactory Quality-of-Service (QoS), in order to be accepted by general users. This poses a challenge in the support of QoS specification, setup, and enforcement for these applications.

In the past decade, various architectures, protocols, and algorithms have been proposed to address these challenging issues. For example, solutions have been proposed for setting up and enforcing QoS in IP or ATM networks, in operating system (OS) kernels, and in applications themselves. While network and OS level solutions provide native and generic QoS support, they may not be easily and rapidly deployed on a large scale and for all new applications. On the other hand, application level solutions, such as adaptive or layered video coding, may be applicable only to a certain application domain.

More recently, various solutions at the middleware layer have also been presented, which reside between applications and OS kernels. In comparison, middleware solutions provide more flexibility when assisting new applications in ubiquitous computing environments. In this article, we propose our solution to QoS specification, setup, and enforcement at the middleware layer. Our middleware easily cooperates with existing solutions at OS, network, and application levels. Furthermore, even when OS or networks are best effort rather than QoS-enabled, the middleware system can still assist applications with QoS adaptations. Our solution spans from QoS specification and translation in the development phase of an application, to QoS setup and adaptation at runtime. We believe that these capabilities are essential to any QoS-aware middleware system.

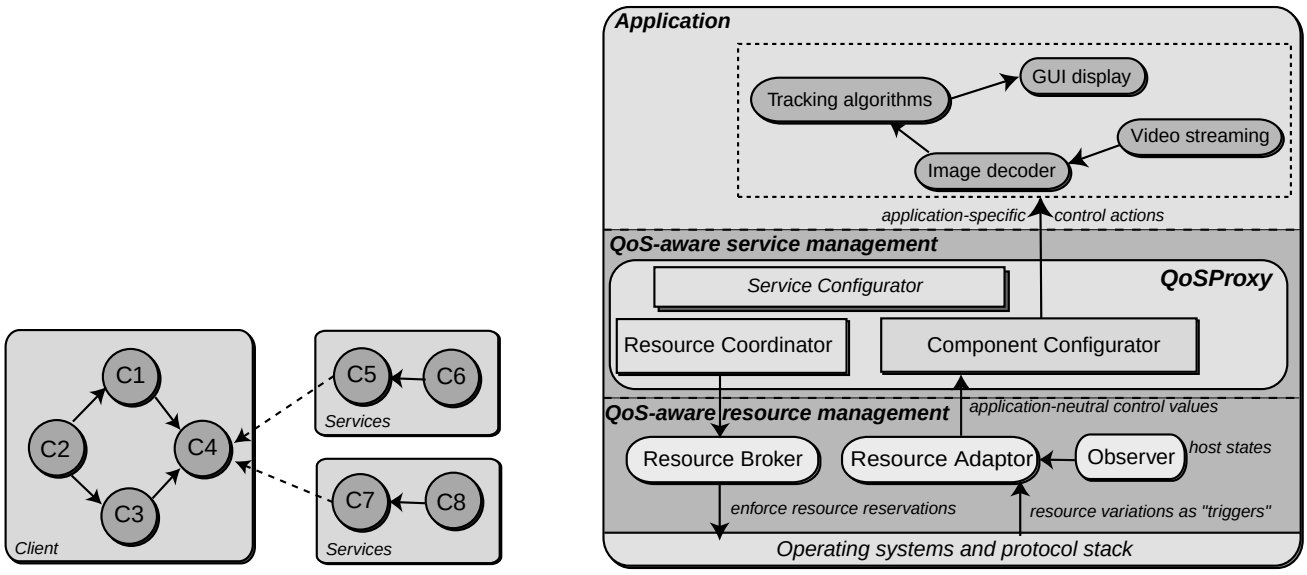
The remainder of this article is organized as follows: Section II presents an architectural overview of our QoS-aware middleware. Section III discusses QoS specification and compilation issues. Sections IV and V present QoS setup and adaptation approaches. In Section VI, we compare existing middleware solutions in related work. We conclude with lessons learned in Section VII.

II. QoS-aware Middleware Architecture

Our QoS-aware middleware architecture favors applications modeled by a generic *application component model*. In this model, we view a collection of interconnected *application components* on a single host as a set of tasks, with input-output dependencies. Beyond a single end-host, we group the entire distributed application into *clients* and *services*. The collection of clients and services form another directed graph representing the service provider-consumer relations. This graph is called an *application functional graph*, as illustrated in Figure 1(a).

In fact, our QoS-aware middleware is a component-base system itself. Its architecture is shown in Figure 1(b), with components at both QoS-aware resource management and QoS-aware service management levels. An instance of this architecture is running in every end-host in the environment.

- *QoS-aware resource management* consists of *Resource Brokers*, *Resource Adaptors* and *Observers*. They are responsible for resource reservation, enforcement, adaptation, and monitoring. QoS-aware resource management is built on top of individual OS and network resource management functions, such as the reservation and scheduling of CPU, disk, and network bandwidth.
- *QoS-aware service management* is represented by a collection of middleware components, collectively referred to as the *QoSProxy*. The decisions and actions of a QoSProxy are driven by resource conditions reported by the underlying resource management components. The definitions of these decisions and actions are initially injected via QoS specification and compilation (to be described in Section III), and they reflect the middleware's capabilities of service discovery, application configuration selection/re-selection/instantiation, and coordinated multi-resource allocation. It is worth noting that QoSProxies operate *only* on the con-



(a) An Application Functional Graph

(b) Components in a QoS-aware Middleware Architecture

Fig. 1. QoS-aware Middleware Architecture: An Overview

trol/management plane of an application, not on its data plane. Therefore, they do not hinder the processing and transmission of the application data.

For an application, the QoS-aware middleware provides support spanning from its development phase to its runtime phase.

- During the development phase (Section III), the application developer specifies QoS parameters, possible configurations and applicable environments of an application. The specifications are then translated by the *QoS compiler*, a companion development tool of the middleware (not shown in Figure 1(b)), into internal representations, which will be injected into the middleware and used at runtime.
- During the runtime phase, the QoS-aware middleware performs QoS setup and adaptation for the application. QoS setup (Section IV) takes place right before the execution of the application, while QoS adaptation (Section V) is triggered during the application execution by resource fluctuation, user mobility, and change of user preference.

III. Application Development Phase

A. QoS Specification

During application development phase, an application developer provides *QoS specification* about the target application. The format of QoS specification varies in different QoS-aware middleware systems. For example, in QoSME [1], QoS is described via a Quality of Service Assurance Language (QuAL); in Agilos [2], QoS is defined via rules and membership functions; while in Q-RAM [3], QoS is represented by resource utility functions. However, QoS specifications share the following characteristics: (1) they are application-specific; (2) their formats are tailored for the targeted application domains, and (3) they need translations from the original application-level notations into the system-level QoS parameters and representations.

For QoS specification of applications in ubiquitous environments, we adopt a representation which includes: (1) an *application description* detailing the set of participating application components, the application QoS parameters and levels, and the mapping function from user-perceived QoS levels to the application QoS levels, (2) *application adaptation policies* indicating when and how the application should adapt to changing environments and resource conditions (to be detailed in Section V), and (3) an *application state template* defining the necessary state information with which the application execution can properly pause and resume. For example, the *application state template* of a media streaming application may be specified as its current video and audio *frame numbers*. Our middleware supports a companion *QoS programming environment* which helps developers conform to such a QoS specification format.

B. QoS Compilation

After accepting the QoS specification of an application, the *QoS compiler* translates the specification into a *QoS profile*. The QoS profile serves as both a 'contract' and a 'script' to be followed by the QoS-aware middleware at runtime. QoS compilation is analogous to program compilation: the application source code is translated into an object code by the language compiler, so that at runtime, the object code will be executed by the runtime support system. Similarly, QoS profile — the 'object code' generated

by the QoS compiler, will be 'executed' by the QoS-aware middleware system for the setup, delivery, and adaptation of application QoS.

With QoS specification as the 'source code', QoS compilation proceeds as follows (more details can be found in [4]):

- **Step 1:** QoS compiler translates the QoS specification into a set of *application functional graphs*. Each graph contains a different set of application components, representing a possible *configuration* of the application. In ubiquitous and heterogeneous environments, it is desirable for an application to have multiple configurations, each suited for a different QoS requirement, resource condition, or physical environment of users.
- **Step 2:** QoS compiler associates each application functional graph with appropriate *system service components* — components which perform domain-specific but application-independent functions, such as CPU monitors, buffers, or RTP-based senders and receivers. Step 2 can be seen as a refinement of the application functional graphs generated in Step 1.
- **Step 3:** QoS compiler derives the end-to-end resource assignment to application components in each application functional graph, i.e. each application configuration. This is done by either analytical resource calculation or experimental resource probing. Our QoS compiler only determines the *minimum* end-to-end resource assignment for each configuration of the application.

The resultant QoS profile consists of three parts: (1) candidate *application configurations* and their resource assignments, (2) *application adaptation policies*, and (3) *application state template*.

IV. Runtime Phase: QoS Setup

After QoS compilation, the application is ready for deployment: the application components will be installed in the servers or clients of this application; the QoS-aware middleware runs in each host in the environment; and the result of QoS compilation — QoS profile, will first be stored in the QoSProxy of the application server. At runtime, parts of the QoS profile will be downloaded to the QoSProxy of each client, as will be described shortly.

In this section, we present the runtime QoS setup performed by the QoS-aware middleware. QoS setup begins when a user starts an application with a certain QoS requirement, and ends when the application begins to execute. The entities involved in this phase are shown in Figure 2. During QoS setup, the middleware customizes the application by selecting one of the pre-specified configurations. The selection is driven by the user's QoS requirement and current end-to-end resource condition.

A. QoS Setup Protocol

Major steps in QoS setup include service discovery, application configuration selection, and resource allocation. In addition, if the user is mobile, QoS setup also performs *application-level handoff* when the user's location or physical environment changes.

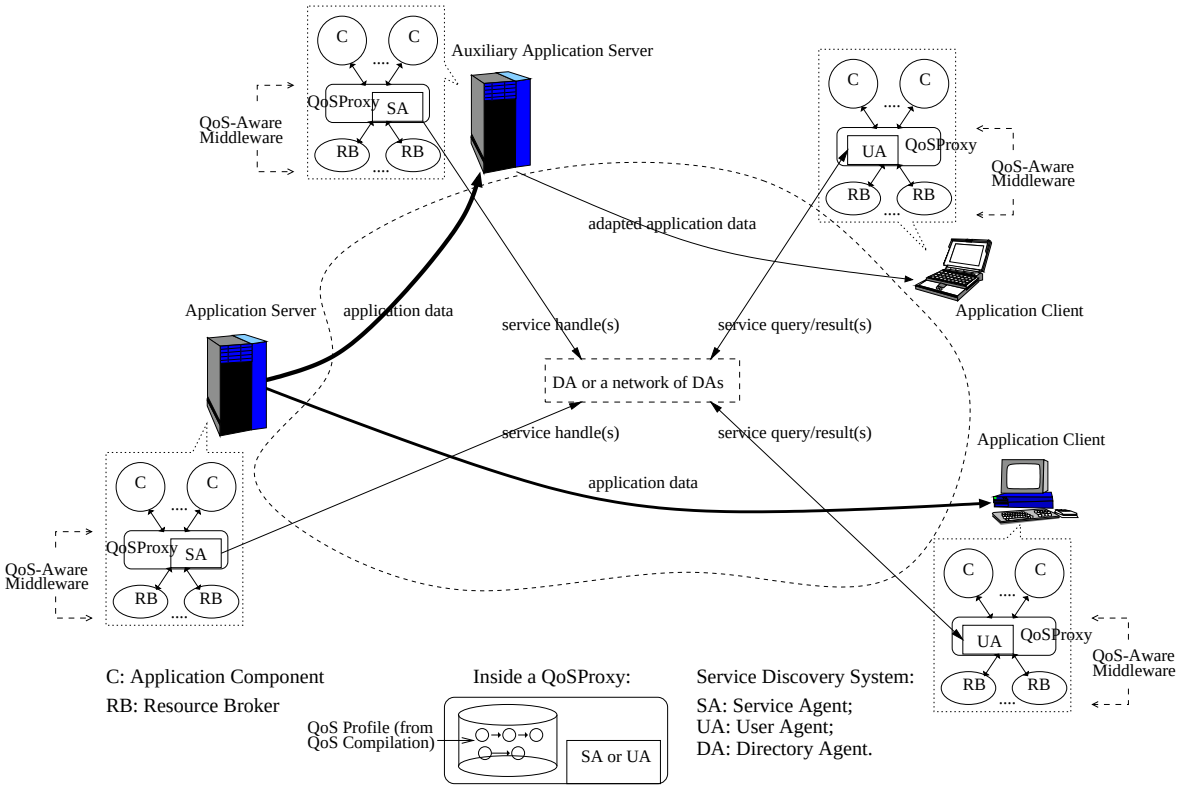


Fig. 2. Entities Involved in the Runtime QoS Setup

- **Step 1 (Service Discovery):** The user's request does not have to designate the location of the corresponding application server. Instead, the user specifies a descriptive *service query*, including the QoS requirement. The query will then be submitted to a *Service Discovery System*, which is a public infrastructural service (like the DNS) responsible for discovering the server of an application. From a user's point of view, the Service Discovery System accepts a service query, and returns *service handles* of a set of qualified servers. Looking into the Service Discovery System, it consists of three types of entities: *User Agent (UA)*, *Directory Agent (DA)*, and *Service Agent (SA)*[1]. As part of the QoSProxies, the UAs and SAs run in clients and in servers, respectively. The DAs are logically independent brokers between UAs and SAs. A UA intercepts a user's service query and submits it to the DA. Meanwhile, an SA sends a service handle on behalf of the server to the DA. Upon receiving a service query from a UA, the DA pulls out every qualified service handle that satisfies the query. If there are multiple qualified service handles, either the DA or the UA will make a choice among them. Examples of Service Discovery Systems include the IETF Service Location Protocol (SLP) [5], Jini by Sun Microsystems [6], and Berkeley's Service Discovery Service (SDS) [7].
QoS-awareness is also an important requirement for a Service Discovery System. It requires that a discovered server be able to deliver satisfactory QoS to the querying client. To make a Service Discovery System QoS-aware, there exist both server-based and client-based approaches: the former involves the reporting of current server performance status by the SA to the DA; while the latter leverages the client feedbacks about recently perceived application QoS from the UA back to the DA [8]. The DA will then use the server report or QoS feedback to make QoS-aware server selections for upcoming service queries.
- **Step 2 (Application Configuration Selection):** After the application server has been discovered, the next step is to customize, or to *configure* the application. In ubiquitous environments, the end-to-end resource conditions observed by clients are highly heterogeneous. For example, the server load or end-to-end network bandwidth may fluctuate, and different clients may have different processing capabilities. Therefore, it is desirable that a ubiquitous application does not execute in a single form. Instead, different configurations will be selected dynamically under different end-to-end resource conditions.
Application configuration selection is based on both the user QoS requirement and the QoS profile generated during the QoS compilation. First, the current end-to-end resource condition is collected by querying the Resource Brokers (RBs) in the client and in the server. Second, the server-side QoSProxy compares the current resource condition with the resource assignment of each candidate application configuration in the QoS profile. The configuration is then selected as the one whose resource assignment is satisfied by the current resource condition, and whose resultant end-to-end QoS is equal to or better than the QoS requirement specified by the user. If no candidate configuration is able to deliver the required QoS, the user may be notified, and the configuration that delivers the best possible end-to-end QoS under the current resource condition may be selected.
In a selected application configuration, there may be application components that run on some auxiliary application servers. These components perform QoS customization under the resource condition that this configuration targets. Locations of the auxiliary application servers also have to be discovered. This is again performed by querying the Service Discovery System.
- **Step 3 (Resource Allocation):** After the application configuration has been selected, and the location of every participating server discovered, the next step is to make multi-resource allocation. First, an end-to-end allocation plan will be generated according to the end-to-end resource assignment given in the QoS profile. Second, the end-to-end allocation plan will be fragmented and dispatched to the QoSProxies running on the server (locally), the client, and the auxiliary server(s) — if any. Third, after receiving the corresponding segment of the end-to-end resource allocation plan, the QoSProxy running on that host will further dispatch the plan to the local RBs. Finally, the RBs will make the actual allocations.
- **Step 4 (QoS profile downloading):** The client-side QoSProxy will download the following two parts of the QoS profile from the server-side QoSProxy: *application adaptation policies* and *application state template*. *Application adaptation policies* are for QoS adaptation (Section V), while *application state template* is for the support of application-level mobility.

When the four steps are completed, the QoSProxy on each host will start the local application component(s) involved in the selected configuration. The execution of the application will then begin.

B. Application-level Mobility Support

In ubiquitous environments, the mobility of users should be treated as a normal case instead of as an exception. This requires that the QoS setup also incorporates support for user mobility, as illustrated in Figure 3. A user may start an application, and then move to another location. The user may move with the same client machine, with no client machine, or even, with multiple client machines such as laptop computer, PDA, and cellular phone. At the new location, the QoS setup must accommodate the continuation of the application: its intermediate execution state has to be restored; and its client machine as well as the corresponding application configuration may have to be re-determined. The reason for a possible change of client machine is the user's changing physical environment. For example, a user at home uses a desktop PC to view an on-line music video. However, when he/she gets into a car, the same music will be delivered in audio-only form to the sound system controlled by an on-board computer. Such a scenario involving user mobility and application continuity is called *application-level hand-off*.

Mechanisms to support application-level handoff need to be incorporated into the QoS setup protocol. First, the deployment of a *User Tracking System* is necessary. It keeps track of users, and maintains information about each user, including the user's ID, carry-on device(s), and current location. Second, the following additional steps are performed during QoS setup for a mobile user.

[1] We use these terms in accordance with the IETF Service Location Protocol specification [5].

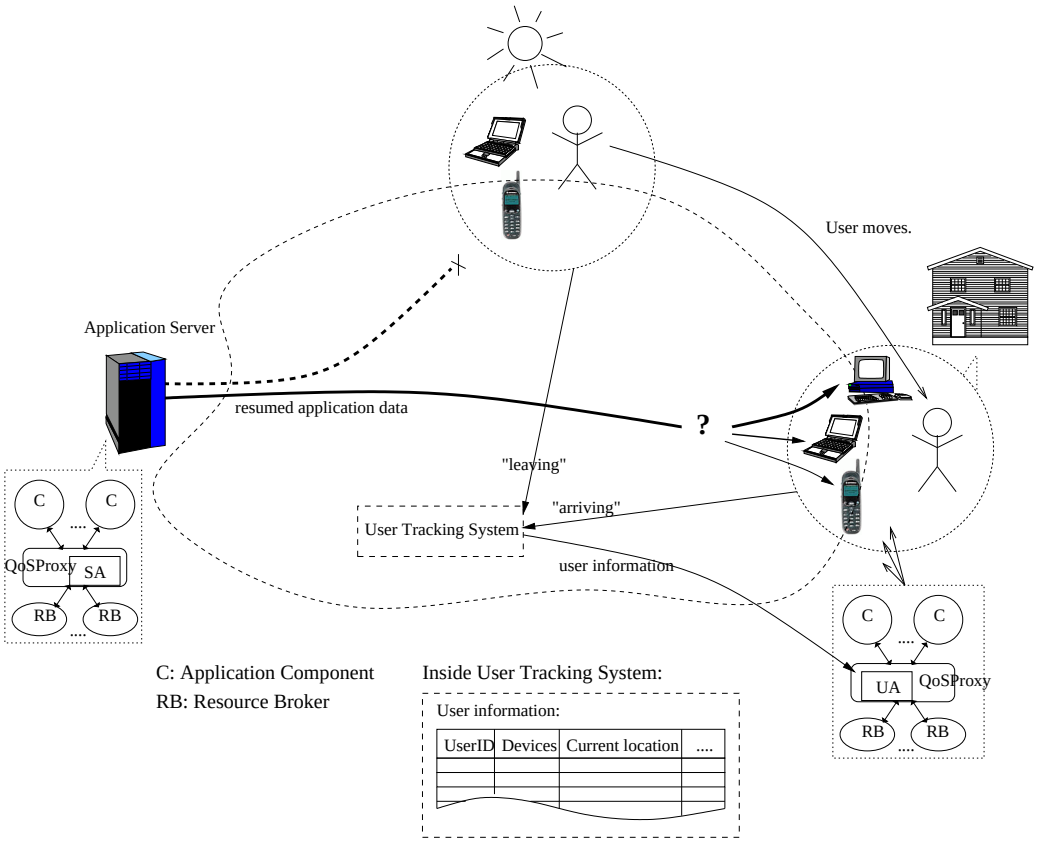


Fig. 3. User Mobility and Application-Level Handoff

- When the user arrives at a new location, the User Tracking System is invoked to recognize the user, and to update and retrieve the corresponding user information. Contact with the User Tracking System can either be initiated manually by the user (for example, via user log on), or be triggered automatically by an active user detection device — for example, each user could carry an intelligent badge which is capable of automatic hand-shake with a computer, based on the computer's proximity to the user.
- A pausing application started earlier by this user does not have to be requested again. Instead, QoS setup can resume its execution automatically. First, the usual steps of QoS setup will be performed, including service discovery (which may be necessary due to the user's location change), application configuration (re)selection, and resource allocation.
- Then, the intermediate execution state of the application will be retrieved and restored by the client-side QoSProxy. The execution state can be retrieved either as part of the user information from the User Tracking System, or from the user's carry-on device, the one that the user always carries with him/her such as a PDA. However, this requires that the state be captured when the user moves away from the *previous* location. To do this, when the user moves away, the QoSProxy of the previous client takes a 'snapshot' of the application execution, according to the downloaded *application state template* (recall Step 4 in Section IV-A). The snapshot, which contains necessary state information to properly resume the application, is then sent to either the User Tracking System or the user's carry-on device.

V. Runtime Phase: QoS Adaptation

At runtime, after QoS setup, the QoS-aware middleware may perform QoS adaptation during the execution of an application. Recall that in Section III, the end-to-end resource assignment for each application configuration is the *minimum* assignment, based on the lowest acceptable QoS delivered by this configuration. Therefore, during the execution, the delivered application QoS should be dynamically adjusted according to the *actual* resource availability. In a worse case, the environment might not even support resource reservations. In both cases, runtime QoS adaptation is necessary.

QoS adaptation takes place at both resource management and service management levels. At resource management level, Resource Observers and Adaptors perform application-neutral adaptation. At service management level, QoSProxies perform adaptation on application components and configurations, based on *application adaptation policies*, a part of the QoS profile (Section III). An integrated model of all adaptations is shown in Figure 4 as a control loop.

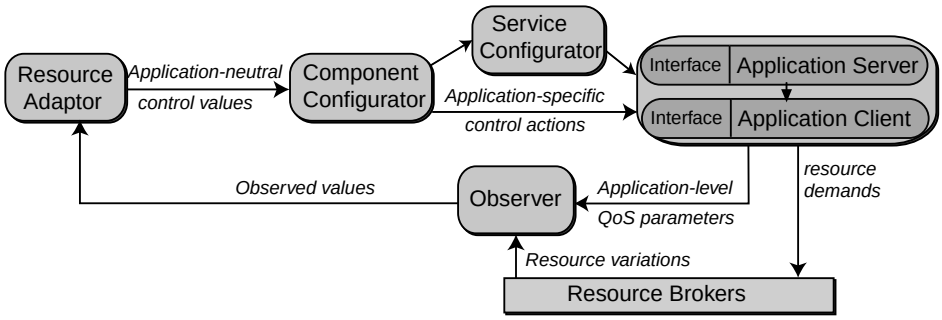


Fig. 4. Viewing QoS Adaptation as a Control Loop

A. QoS Adaptation by Resource Adaptors

Resource Adaptors are neutral to applications and specific to resource types, such as CPU and network bandwidth. A Resource Adaptor controls all concurrent applications sharing the same resource in an end host. It reacts to resource fluctuation by fairly re-allocating the available resource to the sharing applications and notifying them of the changes. Each application, in turn, will adapt the rate, volume, or fidelity of its application data according to the resource allocation changes. Notice that in this type of QoS adaptation, only application data are adapted, while the application configuration remains unchanged.

B. QoS Adaptation by Component Configurators

As part of the QoSProxy, a Component Configurator performs QoS adaptation at a higher level. This type of QoS adaptation involves the replacement, deletion, or addition of application component(s) in the application configuration. Actions of Component Configurators are defined in the *application adaptation policies* (part of the QoS profile). The policies are based on the *Fuzzy Control Model* [2]. The adoption of fuzzy logic is justified by the observation that multiple reconfiguration and parameter-tuning options span different domains, and that the controllable regions and variables within the application are discrete and non-linear. In such a scenario, fuzzy logic allows the specification of such a decision-making process with a small number of *fuzzy rules*. The non-linearity of the fuzzy controller naturally matches the complexities brought by having multiple adaptation choices.

The Fuzzy Control Model utilizes fuzzy logic to express *application adaptation policies* as a configurable *rule base*, which 'fuels' a generic *fuzzy inference engine* to derive the exact control decisions. It contains two parts: *linguistic rules* consisting of a set of linguistic variables and values, and *membership functions* for linguistic values. A typical linguistic rule is:

```
if (cpu is high) and (rate is low) then rateaction := activate_encoder;
```

Such a rule specifies that if the CPU Adaptor allocates CPU in high amount but the Bandwidth Adaptor allocates bandwidth in low rate, then reconfigure the application to activate the video encoder application component. A typical membership function for a linguistic value such as `high` can be expressed with four deterministic points of any trapezoid-shaped membership functions, depending on adaptation requirements. The output of the function is in the range of $[0, 1]$, representing the possibility that adaptation should happen.

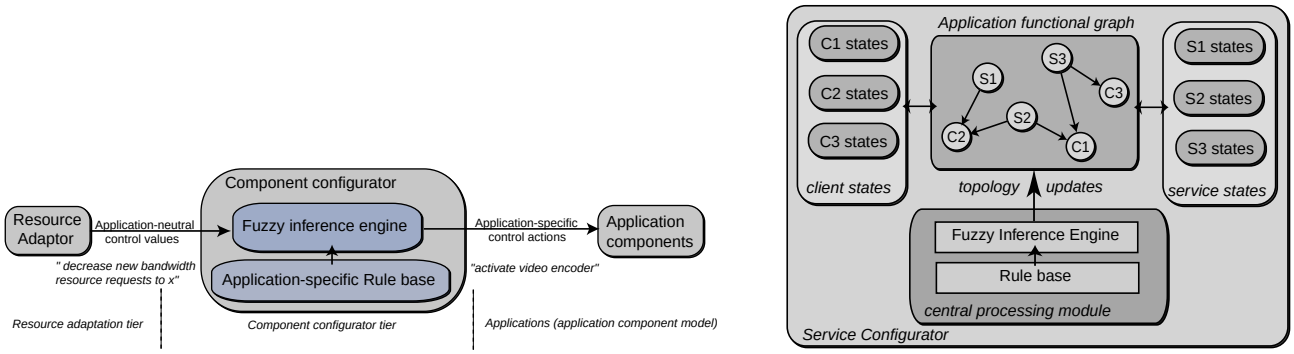
The application-neutral output of the Resource Adaptors is piped into the Component Configurator, fuzzified as input to the inference engine based on its rule base. Any output from the inference engine is then the QoS adaptation decision for the application. For example, in Figure 5(a), the application-neutral output of the Bandwidth Adaptor may be *'decrease new bandwidth resource requests to x'*, and the output of the Component Configurator may be *'activate the video encoder H.261'*.

C. QoS Adaptation by Service Configurators

As part of the QoSProxy, the Service Configurator performs QoS adaptation in an end-to-end fashion, removing the limitation that QoS adaptation can only be performed in a single end host. More specifically, the Service Configurator is able to change the application configuration selected during the QoS setup phase (Section IV). For example, in *Omnitrack*, a distributed visual tracking application [2], when the end-to-end bandwidth becomes unacceptably low, the Service Configurator will decide that the best QoS adaptation is to switch to another video camera server with a low-bit-rate video codec.

Internally, the Service Configurator maintains a *state table* for each of the clients and servers, as well as an application functional graph representing the currently selected application configuration (as shown in Figure 5(b)). If a re-configuration occurs, the graph will be updated correspondingly. The *central processing module* makes the QoS adaptation decisions. We again adopt the fuzzy logic based *fuzzy inference engine* for the purpose of processing input states from hosts and generating an application re-configuration decision. As in the *Component Configurator*, such an inference process is also based on *application adaptation policies* (part of the QoS profile) expressed as a *rule base*, and on states of individual hosts in the application functional graph. In the *Omnitrack* example, a *rule* in the rule base can be:

```
if (server_load is low) and (server_angle is close) then server_ranking is high;
```



(a) QoS Adaptation by Component Configurator

(b) QoS Adaptation by Service Configurator

Fig. 5. QoS Adaptation at the Service Management Level

In this example, `server_angle` is a dynamically generated value derived from the state of a particular client and server, including the actual angle of the client's desired view, the view that the server offers, and the difference between them. On the other hand, `server_load` is the observed CPU load on a server. The better a server matches this criteria, the higher the ranking of a server will have. The highest ranked server should be selected to serve the client, therefore the application configuration involving this server will be selected and instantiated.

VI. Comparison of QoS-aware Middleware Systems

Figure 6 provides a comparison of existing QoS-aware middleware systems: $2K^Q$ [9], Agilos [2], QoS services in CORBA, TAO [10], QuO [11], QoSME [1], Hafid and Bochmann's QoS management framework [12], and Q-RAM [3]. We compare these systems in the following aspects: QoS specification, QoS translation, supported applications, QoS enforcement, and QoS adaptation.

VII. Conclusion

QoS-aware middleware systems have emerged to assist a new spectrum of applications that require QoS in heterogenous and ubiquitous computing environments. In this article, we have shown that using an application component model, it is possible to provide end-to-end application QoS via QoS-aware middleware systems, by (1) generating appropriate QoS specifications; (2) translating and compiling multiple application configurations for the same application to be run in heterogeneous environments; (3) selecting an appropriate configuration and discovering the participating application components; and (4) adapting QoS at multiple levels and with different granularities in case of QoS degradations.

Our own experiences with QoS-aware middleware systems, such as $2K^Q$ and Agilos, provided us with several lessons. First, it is difficult to design a uniform QoS specification language to allow for QoS description in different application domains, and further research is needed. Second, QoS compilations may require application code instrumentation, and hence developer awareness, because not all translations from application QoS to resource assignment can be automated. Third, the resource-level QoS support via Resource Brokers, such as CPU and Bandwidth Brokers, are highly desirable for the provision of end-to-end application QoS. A middleware system can deliver much better end-to-end QoS, if it collaborates with the underlying OS and network QoS support. Finally, QoS adaptation capability is necessary in middleware systems, especially if they are to assist applications on top of best effort OS and networks.

Overall, the results from our current development of QoS-aware middleware systems are encouraging. We believe that such middleware will become an integral constituent of the application enabling platform for emerging ubiquitous and heterogeneous environments.

VIII. Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments on the article. This work was partially supported by the Air Force grant under contract number F30602-97-2-0121, ONR MURI grant under contract NAVY CU 37515-6281, National Science Foundation grants under contract numbers NSF CCR 0086094, NSF EIA 9972884, and NSF EIA 9870736.

References

[1] Patricia Gomes Soares Florissi, "QoSME: QoS Management Environment," *Ph.D. Thesis, Columbia University*, 1996.

[2] 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*, vol. 17, no. 9, pp. 1632–1650, Sept. 1999.

QoS Middleware System	QoS Specification	QoS Translation	Range of Supporting Applications	QoS Enforcement	QoS Adaptation
$2K^Q$	QoS specifications using a QoS programming environment	Multi-phase QoS compilations	Multiple domains of applications via customizable QoS specifications and multi-phase compilations	Guaranteeing minimum-amount reservation of resources	Intra-configuration adaptation and dynamic reconfiguration
Agilos	Fuzzy rules and membership functions	Internal analytical translation with the help of QualProbe at runtime	Applications suited for control-based QoS adaptation	Best-effort (with control-based adaptation)	Three-tier data, component, and service adaptations
QoS in CORBA Audio/ Video Streaming Service	Pre-defined IDL interfaces extending the standard CORBA IDL interfaces	Internal translation	Audio/Video streaming applications	Performed by available transport protocols	Adaptation at network transport layer
CORBA Messaging	Pre-defined IDL interfaces extending the standard CORBA IDL interfaces	N/A	Messaging applications	Message queue ordering in a "router process"	N/A
Real-time CORBA	Pre-defined IDL interfaces extending the standard CORBA IDL interfaces	N/A	Real-time applications	Performed by real-time extensions in standard OS	N/A
Fault Tolerant CORBA	Pre-defined IDL interfaces extending the standard CORBA IDL interfaces	N/A	Fault tolerant applications	Maintaining *x* replicas for a certain fault-tolerant level	N/A
TAO	Pre-defined IDL interfaces	N/A	Real-time messaging applications	Based on priority queues in ORB.	N/A
QuO	A set of Quality Description Languages (QDLs)	N/A	Messaging applications	Performed by individual application-specific implementation and specification via QDLs	Depending on individual application-specific implementation and specification via QDLs
Hafid and Bochmann's QoS management in distributed multimedia applications	N/A	Translation from user level QoS parameters to a suitable application configuration	Distributed multimedia applications	Performed by QoS negotiation and resource allocation protocols	Application reconfiguration via dynamic negotiation
QoSME	Quality Assurance Language (QuAL)	N/A	Applications requiring QoS in transport protocols and in OS	Performed by available transport protocols and POSIX-compliant OS	Using a set of pre-defined operators for QoS re-negotiation
Q-RAM	Resource utility functions	N/A	Applications running in a resource-sharing environment and requiring QoS provision	Based on their resource allocation model	By their adaptive resource allocation algorithms

Fig. 6. Comparison of QoS-aware Middleware Systems

[3] 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*, pp. 298–307, Dec. 1997.

[4] D. Wichadakul and K. Nahrstedt, "Distributed QoS Compiler," *Technical Report UIUCDCS-R-2001-2201, Department of Computer Science, University of Illinois at Urbana-Champaign, (submitted for journal publication)*, Feb. 2001.

[5] E. Guttman, "Service Location Protocol: Automatic Discovery of IP Network Services," *IEEE Internet Computing*, vol. 3, no. 4, pp. 71–80, 1999.

[6] J. Waldo, "The Jini Architecture for Network-Centric Computing," *Communications of the ACM*, vol. 42, no. 7, pp. 76–82, July 1999.

[7] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz, "An Architecture for a Secure Service Discovery Service," *In Proceedings of ACM MOBICOM '99*, pp. 24–35, Sept. 1999.

[8] D. Xu, K. Nahrstedt, and D. Wichadakul, "QoS-aware Discovery of Wide-area Distributed Services," *In Proceedings of IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, pp. 92–99, May 2001.

[9] K. Nahrstedt, D. Wichadakul, and D. Xu, "Distributed QoS Compilation and Runtime Instantiation," *In Proceedings of the Eighth IEEE/IFIP International Workshop on Quality of Service*, pp. 198–207, June 2000.

[10] D. Schmidt, D.Levine, and C. Cleeland, "Architectures and Patterns for High-performance, Real-time CORBA Object Request Brokers," *In Advances in Computers, Marvin Zelkowitz, Ed., Academic Press*, 1999.

[11] J. Zinky, D. Bakken, and R. Schantz, "Architecture Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, Jan. 1997.

[12] A. Hafid and G. Bochmann, "An Approach to QoS Management in Distributed Multimedia Applications: Design and an Implementation," *Multimedia Tools and Applications*, vol. 9, no. 2, 1999.