Visual QoS Programming Environment for Ubiquitous Multimedia Services

Xiaohui Gu, Duangdao Wichadakul, Klara Narhstedt

Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801, USA
{xgu ,wichadak, klara}@cs.uiuc.edu

ABSTRACT

The provision of distributed multimedia services is becoming mobile and ubiquitous. Different multimedia services require application-specific Quality of Service (QoS). In this paper, we present *QoSTalk*, a unified component-based programming environment that allows application developers to specify different application-specific QoS requirements easily. In *QoSTalk*, we adopt a *hierarchical* approach to model application configuration graphs for different distributed multimedia services. We design and implement the XML-based *Hierarchical QoS Markup Language,* called *HQML*, to describe the hierarchical configuration graph as well as other application-specific QoS requirements and policies. *QoSTalk* promotes the separation of concerns in developing QoS-aware ubiquitous multimedia applications and thus enables easy programming of QoS-aware applications, running on top of a *unified QoS-aware middleware* framework. We have prototyped the *QoSTalk* in Java and CORBA. Our case studies with several multimedia applications show that *QoSTalk* effectively fills the gap for application developers between the very general facilities provided by the QoS-aware middleware and different kinds of distributed multimedia applications.

1. INTRODUCTION

In recent years, two major types of QoS-aware middleware systems have evolved: (1) *Reservation-based Systems*, such as *Qualman* [1], get the QoS parameters in the form of application's resource requirements, reserve the specified resources, and during runtime the system's mechanisms and policies enforce the delivery of requested QoS. (2) *Adaptation-based Systems*, such as *Agilos* [2], get the QoS parameters in the form of bounds on resource utilization, resource-specific degradation rules and user decisions, adapting resource allocations according to application specified rules. However, both of them require application-specific QoS parameters such as frame rate, tracking precision or adaptation rules to be provided by application developers. Recently developed reconfigurable component-based QoS-aware middleware systems, such as $2K^Q$ [3], require also application-specific *configuration graphs* from application developer before entering the runtime instantiation phase. Based on these observations, a new critical challenge has emerged to provide a unified QoS-aware programming environment as well as runtime instantiation framework. Solutions for this new challenge are needed to fill the gap between the application-neutral middleware services and the input of application-specific QoS requirements.

Several recent works have addressed the problems of QoS specifications and programming environment from different directions. In [4], a service contract- based API is designed to formalize the end-to-end QoS requirements of the user and the potential degree of service commitment of the provider. A contract is a C data structure including all the conceived clauses. Although it is possible to mix QoS-related code or specification with the functional code, it is highly desirable to separate the non-functional requirements from the functional requirements so that the two parts can be developed and maintained independently. QML (QoS Modeling Language) [5] is an independent QoS specification language for distributed object systems. It allows users to specify non-functional aspects of services separate from the interface definition. However, QML does not consider the resource-level QoS specifications and the configuration graphs for the reconfigurable multimedia applications. The QuO [6,7] project is the closest in approach to our own work. It provides a set of specialized languages to specify different aspects of QoS support in their framework based on the aspect-oriented programming. However, different from QuO, we use XML as our QoS specification language to make our framework most applicable. Moreover, the QoS specifications are considered in a broader context, namely the ubiquitous computing environment.

In this paper, we present *QoSTalk*, a unified QoS programming environment, which allows application developers to specify, process and store different application-specific QoS requirements easily and efficiently. *QoSTalk* enables easy programming of QoS-aware multimedia applications, running on top of a unified QoS-aware middleware framework. In the design and implementation of *QoSTalk*, we assume that applications are component-based. We have implemented a prototype of *QoSTalk* in Java and CORBA. To evaluate the effectiveness of *QoSTalk*, we performed case studies with several distributed multimedia applications, such as *ubiquitous video on demand* and *video conferencing*. Our case studies show that *QoSTalk* greatly simplifies the design and implementation of QoS-aware multimedia applications.

The rest of the paper is organized as follows. Section 2 presents the overall architecture of *QoSTalk*. Section 3 describes the *Visual Hierarchical QoS Editor*. Section 4 presents the design of the XML-based QoS specification language, called *HQML*. Section 5 presents the experimental results from the *QoSTalk* prototype. Section 6 concludes this paper.

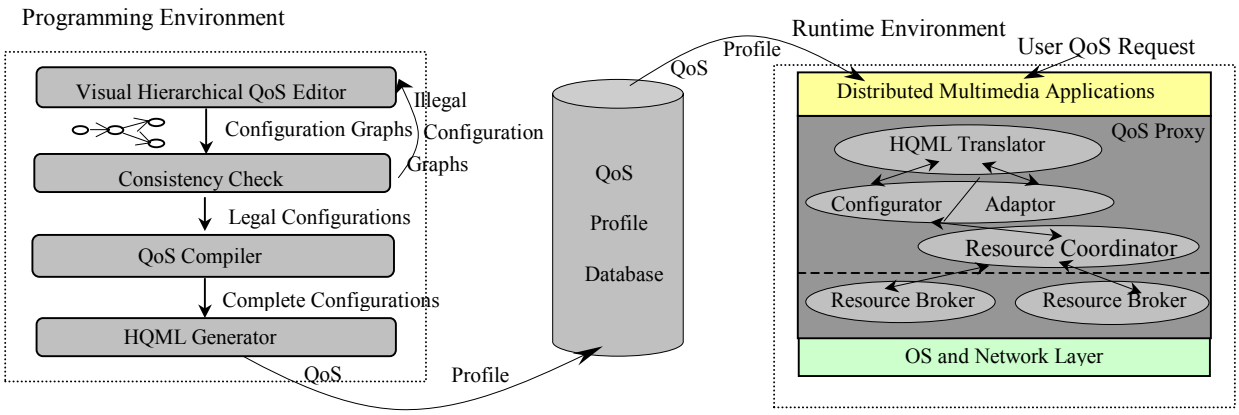

Figure 1: The QoS Programming Environment Architecture

2. *QoSTalk* ARCHITECTURE

The overall architecture of *QoSTalk* is shown in Figure 1. The major objective of the architecture is to provide a unified programming framework for application developers to input all kinds of application-specific QoS requirements easily. The contribution of *QoSTalk* is twofold: first, it makes QoS-unaware applications become QoS-aware by instrumenting the code and profiling application-specific QoS requirements; Second, it makes traditional QoS-aware applications become lightweight and more efficient by delegating all QoS-aware services to the middleware, namely the QoS proxy.

The application developer first uses the *Visual Hierarchical QoS Editor* to draw all possible configurations for a particular application using visual tools and inputs all kinds of application-specific QoS requirements and policies via dialogs. Second, the developer uses our *Consistency Check* tools [8] to "debug" the input configuration graph. If there is any inconsistency in the input configuration graph, the error messages are returned to the application developer in the *Visual Hierarchical QoS Editor*. Otherwise, the legal configuration graph is passed to the *QoS Complier* [9] to probe the resource requirements and establish the mapping between application QoS parameters and resource requirements automatically. In the fourth step, the legal configuration graph with complete QoS specifications is passed to the *HQML Generator* [8]. The *HQML Generator* "traverses" the complete configuration graph to generate the HQML file, namely the application-specific QoS profile, automatically. Finally, the complete HQML file is saved into the *QoS Profile Database*.

During runtime, the *QoS Proxy* on the client or server host acquires the necessary application-specific QoS profile from the *QoS Profile Database*. It chooses the most suitable QoS profile according to the user requirements and the current *end-to-end* resource availability. The *HQML Translator* translates the chosen QoS profile into desired data structures and feeds them into different parts of the runtime QoS proxy such as the *Configurator* and *Adaptor*. Clients of a ubiquitous multimedia service receive satisfactory QoS automatically and with low setup overhead, within their end-to-end resource availability constraints.

3. VISUAL HIERARCHICAL QoS EDITOR

The *Visual Hierarchical QoS Editor* presents a set of visual tools and dialogs for application developers. It allows the developer to depict all distributed components and their relations to create a set of distinct configurations for a particular application. For example, a distributed Video-On-Demand multimedia application may include four distributed components: (1) MPEGII Video Server (2) MPEGII to Bitmap Transcoder (3) MPEGII Player and (4) Bitmap Player. One possible configuration is a MPEGII Video Server plus a MPEGII Player. The other possible configuration is a MPEGII Video Server, a MPEGII to Bitmap Transcoder gateway and a Bitmap Player. Different configurations of the same application provide different QoS levels or even the same QoS levels but have different resource requirements.

The *Visual QoS Editor* is based on a hierarchical approach. There are two reasons driving us to use the hierarchical design. First, the hierarchical design is more scalable. A complex distributed component-based application may include tens of components. It is difficult to draw all of them in a single page. Second, the hierarchical design makes the relationships between different components much more clear. There are essentially three component levels in the configuration graph:

- **Atomic components.** An atomic component only contains one basic multimedia function such as a *MPEGII Decoder*.
- **Composite components.** Each composite component consists of a set of atomic components. We assume that all atomic components within one composite component are instantiated in one host machine during runtime.
- **Composite component groups.** They represent different computer clusters: (1) Server Cluster, (2) Gateway Cluster, (3) Client Cluster, and (4) Peer Cluster. Each composite component group may include several same type composite components.

We have also designed three different links to represent three different relations between components:

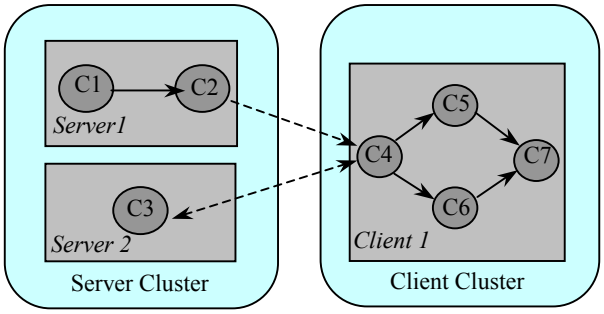

Figure 2: A generic three-level configuration graph

- **Fixed links.** A fixed link defines a wired data communication channel between two components. It cannot be interrupted or moved during runtime. It is represented by the solid line in the configuration graph.
- **Mobile host links.** A mobile host link defines a wireless communication channel between two components. It is used to represent host mobility, which means the end host machine could move within certain range during runtime. It is represented by the dashed line.
- **Mobile user links.** A mobile user link is defined to specify the user mobility, which means the user could move from one machine to another during runtime. In other words, when the user moves from the old machine to a new machine during the runtime of an application, such as the *Video On Demand*, the old link from the server to the old machine is torn down. A new connection from the server to the new machine is established and the application session is recovered and resumed automatically from the interruption point. It is represented by the dotted line.

All of these links could be one-way communication channels or two-way communication channels. Figure 2 illustrates our three-level hierarchical design.

4. HQML- HIERARCHICAL QoS MARKUP LANGUAGE

In order to make our programming framework most applicable and also facilitate web applications to utilize QoS-aware middleware services, we design and implement a Hierarchical QoS Markup Language (*HQML*) as our QoS specification language. *HQML* uses the standard XML mechanism for definitions. Thus, we could utilize the existing XML database technique to build the QoS Profile database. Figure 3 gives an example of QoS specifications in HQML for a distributed mobile video on demand application.

HQML uses intuitive qualitative values such as *high, low, average*, for the user-level QoS specifications. HQML provides many constructs for application-level QoS specifications. An HQML file has the same hierarchy as its corresponding hierarchical configuration graph. For example, the specifications between the "<Server>" and "</Server>" tags are the QoS

```
<APP name = "Mobile VoD" >
  <Configuration id = "101" >
    <QoSLevel> average  </QoSLevel>
    <CriticalQoSPara>FrameRate </CriticalQoSPara>
    <Range unit = "fps">
      <UpperBound> 30  </UpperBound>
      </LowerBound> 15 </LowerBound>
     <ServerCluster>
       <Server >
        <Name> Video Server </Name>
        <Hardware> Sun Ultra 60 </Hardware>
        <Software> Solaris 5.0 </Software>
        <CPU unit = "percentage"> 30 </CPU>
        <Memory unit = "KB"> 8000 </Memory>
        <Disk unit = "MB" > 10 </Disk>
        <Bandwidth unit = "MB"> 5  </Bandwidth>
        <Atomic name = "MSP Video Server">
          <AdaptationRules> … </AdaptationRules>
           …
        </Atomic>
       </Server>
      </ServerCluster>
      <ClientCluster>
          …
      </ClientCluster>
     <Link type = "MobileUserLink">
      <PersistentSate> FrameNumber </PersistentState>
     </Link>
   </Configuration>
</App>
```

Figure 3: An Example of QoS Specifications in HQML

requirements of one composite server component. As discussed in section 2, the framework shows that during the runtime instantiation phase, the HQML translator parses the chosen HQML file to generate the configuration graph and feeds the configuration graph into the *Configurator* of the QoS proxy. The adaptation rules tell the QoS Proxy how to gracefully adapt the application with minimum user level QoS violations during the resource fluctuation period (e.g., "Add compression when the cpu load is low and network load is high"). They are retrieved from the HQML file and sent to the QoS proxy's *Adaptor*. The persistent states such as the "frame number" are parsed and sent to the QoS proxy's *Persistent State Manager*. The critical QoS parameter represents the most important QoS parameter, which is protected by degrading other QoS parameters when resource availabilities change.

HQML also provides some mechanisms for resource level QoS specifications. During runtime, the QoS proxy chooses the most suitable configuration according to the best match between these resource level QoS specifications and the current *end-to-end* resource availability. The most suitable configuration is the one that is affordable by the current *end-to-end* resource availability and with the highest user-level QoS.

5. EXPERIMENTAL RESULTS

We have implemented a prototype of *QoSTalk*. The *Visual Hierarchical QoS Editor* is implemented in Java Swing. The *HQML Translator* is also implemented in Java. Thus, our implementation is platform independent. QoSProxies such as the *Configurator*, *Adaptor*, and *Resource Brokers* are implemented as CORBA objects. The multimedia service components are also implemented as CORBA objects. Our experiments with several multimedia applications such as the *Video Conferencing* and *Distributed Video On Demand* show the soundness of *QoSTalk*. Figure 4 shows one screenshot of the *Visual Hierarchical QoS Editor*. Figure 5 presents the QoS setup times for different multimedia applications. The QoS setup time is divided into three main portions: the HQML translation time, the dynamic downloading time, and the instantiation time. The largest time used during the QoS setup phase is the dynamic downloading time for all applications.

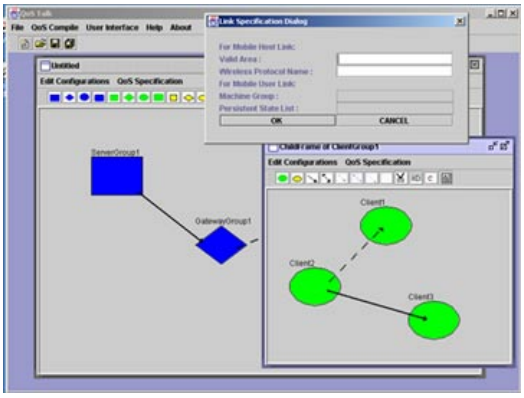

Figure 4: Screenshot of the Visual QoS Editor

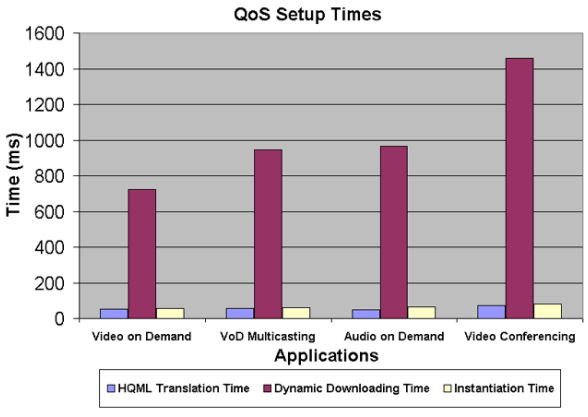

Figure 5: QoS setup times for different multimedia applications

6. CONCLUSIONS AND FUTURE WORK

In this paper, we present *QoSTalk*, a unified QoS programming environment for ubiquitous multimedia applications. We make two fundamental contributions: (a) We design and implement a *Visual Hierarchical QoS Editor*, which allows application developers to draw all candidate configurations using visual tools and input application-specific QoS requirements via dialogs. (b) We design and implement an XML-based *Hierarchical QoS Markup Language* (HQML) for QoS specifications at different layers. In the future, we will investigate new probabilistic and predictive techniques for QoS profiling and specification.

7. ACKNOWLEGEMENTS

This research is supported by the National Science Foundation Career Grant under contract number NSF CCR 96-23867, NSF CISE Infrastructure grant under contract number NSF EIA 99-72884, and NASA grant under contract number NASA NAG 2-1406.

8. REFERENCES

[1] K. Nahrstedt, H. Chu, and S. Narayan. QoS-Aware Resource Management for Distributed Multi-media Applications, *Journal on High-Speed Networking*, 8, 1998

[2] B. Li, and K. Nahrstedt. A control-based middleware framework for quality of service adaptation, *IEEE Journal on Selected Areas in Communication*, Sept. 1999.

[3] K. Nahrstedt, Duangdao Wichadakul, and Dongyan Xu. Distributed QoS Compilation and Runtime Instantiation, *Proceedings of IEEE/IFIP International Workshop on QoS 2000 (IWQoS2000),* June 2000.

[4] Andrew T.Campbell. A Quality of Service Architecture, *PhD Thesis, Computing Department, Lancaster University*, Jan. 1996.

[5] S.Frolund, and J.Koistinen. QML: A Language for Quality of Service Specification, *Technical Report HPL-98-10,* Feb. 1998.

[6] J. P. Loyall,R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and measuring Quality of Service in Distributed Object Systems, *Proceedings of the first IEEE International Symposium on Object-Oriented Real-tim Distributed Computing (ISORC'98)*, April, 1998, Japan.

[7] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using QDL to specify QoS Aware Distributed (QuO) Application Configuration. *Proceedings of the third IEEE International Symposium on Object-Oriented Real-tim Distributed Computing (ISORC'2000)*, March, 2000.

[8] X. Gu and K. Nahrstedt. Visual Quality of Service Specification for Distributed Heterogeneous Systems, *Technique Report No. UIUC DCS-R-2000-2190, Computer Science Department, University of Illinois at Urbana – Champaign*, Nov. 2000.

[9] D. Wichadakul and K. Nahrstedt. Distributed QoS Compiler, *Technique Report No. UIUC DCS-R-2000-2201, Computer Science Department, University of Illinois at Urbana – Champaign*, Feb. 2001.