A Programming Framework for Quality-Aware Ubiquitous Multimedia Applications *

Duangdao Wichadakul, Xiaohui Gu, Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana,IL 61801
wichadak, xgu, klara @cs.uiuc.edu

ABSTRACT

Ubiquitous computing promises a computing environment that seamlessly and pervasively delivers applications to the user, despite changes of resources, devices, and locations. However, few ubiquitous multimedia applications (UMAs) exist up-to-date. One of the main reasons lies in the fact that it is difficult and error-prone to build a UMA which is mobile and deployable in different ubiquitous environments, and still provides acceptable application-specific Quality-of-Service (QoS) guarantees. In this paper, we present the design and implementation of a novel programming framework, called "QCompiler" to address the challenges. The framework includes (1) a high-level application specification for the application developer to easily write a UMA with specific quality, mobility, and ubiquity supports, (2) a meta-data compilation, which provides automated consistency checks, translations, and substitutions, to relieve the application developer from dealing with complex programming related to quality, mobility, and ubiquity, (3) a binding, which prepares a quality-aware specification to be executable, in a specific deployment environment, and (4) a run-time meta-data execution, utilizing the meta-data compilation's results, to manage and control a quality-aware multimedia application. As a case study, we apply the programming framework to build a mobile Video-on-Demand (VoD) application. The experimental results show tradeoffs between easiness and flexibility to develop and deploy UMA, and overheads during UMA instantiation and adaptation.

1. INTRODUCTION

Ubiquitous computing promotes the proliferation of various stationary, embedded and mobile devices interconnected by heterogeneous networks (e.g., wired, wireless, infrared). It leads to a more *dynamic* distributed computing environment than ever before, where resource fluctuations, device/service changes are a common phenomenon rather than viewed as an extreme case. Many emerging distributed multimedia applications such as Video-on-Demand and video conferencing, are being developed in such a computing environment. Thus, a big challenge for the application developer is to build ubiquitous multimedia applications (UMAs) that can continuously and pervasively deliver multimedia contents with adequate quality to the user, in spite of resource fluctuations, device heterogeneity and user mobility.

Although the hardware technology (e.g., hand-held devices) and networking infrastructure (e.g., wireless networks), necessary for implementing the vision of ubiquitous computing, are becoming reality, and reusable multimedia components are widely available, few multimedia applications have been built in such a computing environment. The main difficulties of building quality-aware UMAs are as follows: (1) different multimedia applications deal with different application-specific performance criteria (e.g., frame rate for the Video-on-Demand, lip synchronization skew for the video conferencing, and tracking precision for the visual tracking application), (2) these applications are expected to be deployable in dynamic distributed computing environment with different computing and communication capacities, and (3) mobility becomes a standard feature of these applications.

Much research work has been done to provide solutions for setup and enforcement of application's quality in networks [4, 24], in operating systems [25, 7, 21], and most recently in middleware systems [17, 30, 5, 9, 13]. These *QoS-enabling services*; however, are designed to support only particular quality enabling mechanisms such as resource-specific reservations (e.g., [4, 24, 25, 7, 21]), and adaptations of application's quality in the best effort environments (e.g., [17]). Furthermore, all of them are implemented in specific languages with specific interfaces, and expected parameters. Also, no services enable application-level mobility flexibly.

To include QoS-enabling services into UMA programs, the application programmer needs to know them and select among them appropriately. It means, he/she must well understand specific characteristics of the adopted QoS-enabling services, know how to translate/map application-specific performance criteria into the underlying network/OS services'

*This work was supported by the NASA grant under contract number NASA NAG 2-1406, NSF under contract number 9870736, 9970139, NSF-CCR 9988199, and EIA 99-72884EQ. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or NASA.

expected interfaces and parameters (e.g., resource requirements for resource-specific reservations), and know how to "hook" the application with these adopted services. The result is then that the developed multimedia application is programmed as tightly-coupled with the adopted underlying QoS-enabling services. Hence, it is not easily deployable in different and dynamic environments.

To go beyond the traditional building of a quality-aware multimedia application, as described above, we propose a novel programming framework, called "QCompiler", for quality-aware ubiquitous multimedia applications. This framework enables a flexible and efficient development and deployment of distributed multimedia applications with mobility in ubiquitous environments. The framework is based on the concepts of reusable multimedia components, reusable underlying QoS-enabling services, the de-coupling of generic QoS-enabling services and their specific implementations, and the provision of automated translations and substitutions to relieve the application developer from dealing with underlying QoS-enabling services' details.

The programming framework includes (1) a *high-level application specification*, (2) a *meta-data compilation*, (3) a *binding*, and (4) a *run-time meta-data execution*. The high-level application specification consists of a set of quality-related specifications (e.g., attributes, rules), which allow an application developer to flexibly represent an application and its service quality requirements. The meta-data compilation processes the specification in two steps: environment-independent and environment-dependent translations. The environment-independent translation maps the high-level application specification into a QoS-aware application descriptor[1], representing a portable meta-level quality-aware application code. The descriptor includes consistent compositions (configurations) of multimedia service components and their associations with generic QoS-enabling services, and adaptation rules. The environment-dependent translation helps the application developer to deploy the QoS-aware application descriptor in a specific deployment environment. It matches each configuration in the descriptor with the environment. Also, it provides automatic translations (e.g., between application-level quality parameters and resource-level parameters) to ease the task for the application developer. The binding helps to bind components in the compiled results from the environment-dependent translation. The run-time meta-data execution helps controlling a quality-aware application, based on specific user's quality request, current availability of devices and resources, user mobility, and the compiled results from the environment-dependent translation.

By using the proposed quality-aware programming framework, an ordinary application developer can easily, and efficiently implement different applications with quality, mobility, and ubiquity supports on top of available QoS-enabling services (e.g., RSVP [4], DSRT [7]), in different deployment environments.

The rest of this paper is organized as follows. Section 2 introduces fundamental models of ubiquitous multimedia applications, used in our programming framework. Section 3 describes the high-level application specification as mechanism for developing a quality-aware ubiquitous multimedia application. Section 4 presents the meta-data compilation which enables portability, quality-awareness, and adaptability controls for the input specification. Section 5 describes the binding and Section 6 briefly describes the run-time meta-data execution. Section 7 presents experimental results, followed by Section 8 with related work overview. Section 9 concludes this paper.

[1]We use the convention of deployment descriptor as of EJB's deployment descriptor [22], and CCM's descriptors [14].

2. QUALITY-AWARE UBIQUITOUS MULTIMEDIA MODELS

Our programming framework for quality-aware ubiquitous multimedia applications requires sound models of these applications. We deploy the quality-aware task-flow model for the overall UMA and the component model for each task as discussed below.

2.1 Ubiquitous Multimedia Application (UMA) Model

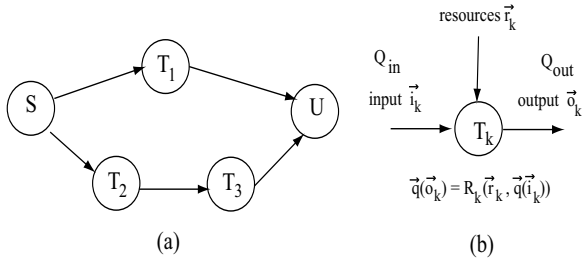


Figure 1: (a) Task-Flow Model, (b) Reward Profile

We deploy the task-flow model [18] as our quality-aware UMA model. In the task-flow model (See Figure 1a), the relationship among tasks can be represented by a directed acyclic graph (DAG), called dependency graph. An edge from task T_k to task T_i indicates that T_k produced the result consumed by task T_i. S is a *source node* that is not a consumer of any task, but is a producer of some task. U is an *end-user node* that is not a producer of any task, but a consumer of some task. Each task is a functional unit consisting of two vectors, Q_{in} and Q_{out}, as its input and output quality vectors, respectively. Q_{in} and Q_{out} are related such that $\vec{q}(\vec{o}_k) = R_k(\vec{r}_k, \vec{q}(\vec{i}_k))$, where R_k is a *reward profile* (See Figure 1b), representing a mapping from input quality and resource allocation to output quality. $\vec{q}(\vec{o}_k)$ is a single output quality. $\vec{q}(\vec{i}_k)$ is a single input quality. $\vec{r}_k$ is the resource allocation of the task. A value function $V_i(\vec{q}(\vec{o}_k))$ is placed by a consumer task T_i on the output quality $\vec{q}(\vec{o}_k)$ of a producer task T_k. It implicitly specifies service quality expected by a consumer task. In UMA model, a ubiquitous multimedia component (UMC) represents a task in the task-flow model.

2.2 Ubiquitous Multimedia Component (UMC) Model

A multimedia component is a functional unit (e.g., media retrieving, encoding, streaming) or a set of functional units (e.g., media retrieving+encoding+streaming) forming a multimedia service (e.g., Video-on-Demand server).

A ubiquitous multimedia component (UMC) (See Figure 2) is modelled as a multimedia component code (software program) or a composition of multimedia component codes,

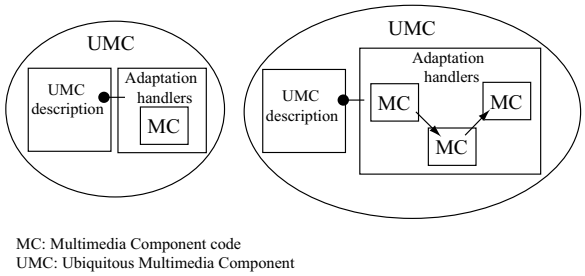


Figure 2: Ubiquitous Multimedia Component (UMC) Model

attached with a meta-data description, and wrapped with a set of adaptation handlers.

UMC description presents detailed information of the component including component name, component model (e.g., CORBA, COM, Java), category (e.g., VoD Server Service, Transcoding Service, Encoding Service), component repository, interface, hardware requirement, system software requirement, system resource requirement, supporting quality, and required libraries. The "supporting quality" points to a profile which consists of quality parameters (described in Section 3) and their reward profiles, that the UMC supports. The "required libraries" information points to a profile which consists of a list of the UMC's required libraries with their locations or pointers to their locations. The UMC description is mainly used by the programming framework during the meta-data compilation.

Adaptation handlers are a set of (action) functions which the programming framework includes in the UMC model for enabling adaptations, and mobility of the UMCs in ubiquitous environments. As a tool in the programming framework, we wrap each UMC with a set of (action) functions: *tuneQualityParams(params_vector)*, and *reconnect(service_id)*, for handling adaptation controls from the underlying runtime meta-data execution. Actions are described in detailed in next section.

A UMC is a reusable component. Like normal multimedia components, a UMC or a set of UMCs forms a multimedia service.

3. HIGH-LEVEL APPLICATION SPECIFICATION

3.1 Terminology

In the quality-aware programming framework, we use the following definitions: *QoS category* and *QoS dimension* [3, 10] are part of the *quality-aware application specification*, and *UMC description*. A QoS category consists of a set of quality parameters (*QoS dimensions*) representing qualitative or quantitative attributes for the *QoS category*. Real-time video, real-time audio are examples of QoS categories. Frame rate, frame size, color depth are examples of QoS dimensions describing real-time video category. In our framework, we limit the QoS dimension to specify only the quantitative attribute.

3.2 Quality-Aware Application Specifications

High-Level application specification allows an application developer to represent a quality-aware multimedia application via a set of meta-data specifications that include (1) detailed *application specification*, (2) *user-to-application-specific quality translation template (UtoA template)*, and (3) *adaptation rules*.

3.2.1 Application Specification

An *application specification* includes (i) an application functional dependency graph, (ii) setup configuration(s), (iii) a service component description for each UMC in the graph, and (iv) a connection description for each pair of connected UMCs.

The *application functional dependency graph* represents the quality-aware multimedia application via the composition of different ubiquitous multimedia components (UMCs). It is described accordingly to the quality-aware task-flow model, and representing a UMC service graph. The *setup configurations* represent different compositions of components in the dependency graph which are needed to be instantiated during the application setup. The *service component description* includes details of each UMC comprising the dependency graph. It includes the same fields as the UMC description with the following additions: component type (e.g., Specific, Generic, Composite), target machine(s), and state (e.g., Shared, or Exclusive). A dependency graph is partially-defined if some of its components are Generic; otherwise, it is fully-defined. The *connection description* includes connection type (e.g., unicasting, multicasting), and security capability (e.g., enable encryption, decryption).

Figure 3 shows an example of application meta-data specification for a mobile Video-on-Demand (VoD) application, entered via the visual programming environment [12]. As shown in Figure 3, the mobile VoD consists of five UMCs: a VoD server, a user profile server, a proxy server (for video transcoding), a VoD client on a PC, and a VoD client on a hand-held device. The dependency graph of the application is represented on the left side. For simplicity, the dependency graph does not show the relations between the VoD clients and the user profile server. The dialog on the right side represents the service component description of the *VoDClientPC*. We assume that the three setup configurations: {*VoDServer*, *VoDClientPC*}, {*VoDServer*, *UserProfileServer*, *VoDClientPC*}, and {*VoDServer*, *UserProfileServer*, *ProxyServer*} have been specified (not shown in the figure).

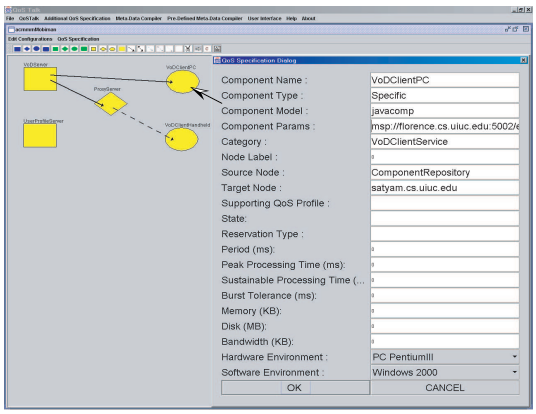


Figure 3: Application Specification for a Mobile Video-on-Demand Application (Example)

3.2.2 User-to-Application-Specific Quality Translation Template (UtoA Template)

User-to-application-specific quality translation template defines the mapping between different user quality levels and corresponding application specific QoS categories, their QoS guarantee levels, and their dimensions. User QoS levels are provided by the application developer for a user to request the application. Application-specific QoS categories and dimensions, for each user QoS level, are used by the meta-data compilation to generate value functions. Figure 4 shows an example of the UtoA template for the mobile VoD application in the visual programming environment. In this example, three user QoS levels: *High*, *Medium*, and *Low* are mapped to [*(format, {mpeg-2, mjpeg})*, *(frame rate, 30)*, *(frame size, 740x480)*, *(color depth, 8)*], [*(format, {mpeg-2, mjpeg})*, *(frame rate, 20)*, *(frame size, 480x360)*, *(color depth, 4)*], and [*(format, {mpeg-2, mjpeg})*, *(frame rate, 10)*, *(frame size, 360x240)*, *(color depth, 4)*], respectively.

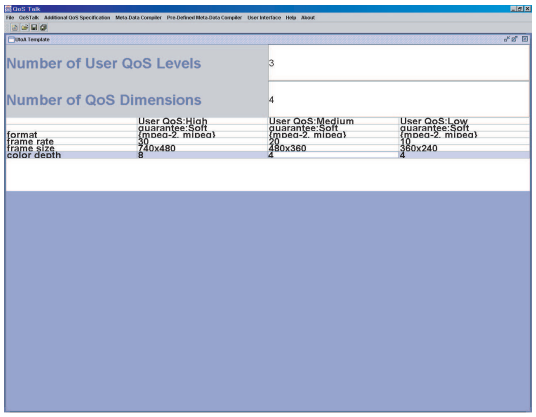


Figure 4: User-to-Application-Specific Quality Translation Template for a Mobile Video-on-Demand (VoD) Application (Example)

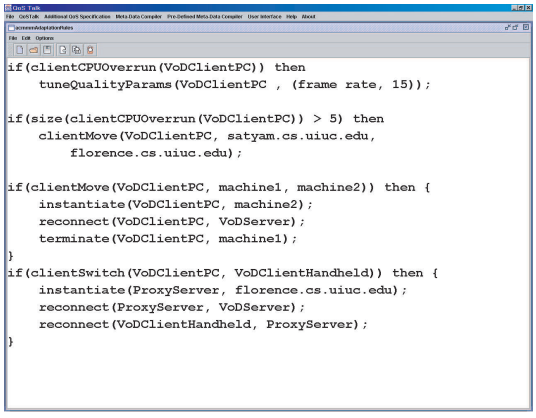
3.2.3 Specification of Events, Actions and Adaptation Rules

To handle the dynamic characteristics of ubiquitous environments and their UMAs, our programming framework provides a pre-defined set of events and actions that can be used by the application developer to describe how the run-time meta-data execution should control the application corresponding to resource availability, load balancing, and mobility.

Examples of events are *clientMove(client_id, machine1, machine2)*, *serverSwitch(server_id1, server_id2)*, *userMove(user_id, space_id1, space_id2)*, *networkOverload(machine1, machine2)*, *serverShutdown(server_id)*, *clientCPUOverrun(client_id)*, *cpuExceeded(machine1)*. *clientMove()* represents the moving of a specific client, defined by *client_id*, from *machine1* to *machine2*, assuming that no instances of the client are running on *machine2*. *serverSwitch()* represents the switching of a server from a specific *server_id1* to another *server_id2*, assuming that there exists an instance of *server_id2* already running. Moving and switching events are applied to clients, servers, and peers. A *client_id*, a *server_id*, or a *peer_id*, represent active instances of a multimedia service. *userMove()* indicates user mobility from *space_id1* to *space_id2*, assuming that two spaces are part of the same deployment environment. *networkOverload()* indicates the overloaded network between two machines. *serverShutdown()* indicates that the *server_id* has been shutting down. *clientCPUOverrun()* indicates that the client with *client_id* excessively consumes CPU. *cpuExceeded()* indicates that total CPU utilization in *machine1* exceeds a pre-defined threshold.

In our programming environment, the following actions are pre-defined: *instantiate(UMC, machine1)*, *insert(client_id, UMC, server_id)*, *tuneQualityParams([server, client, peer]_id, params_vector)*, *reconnect([server, client, peer]_id or UMC, [server, client, peer]_id or UMC)*, and *terminate([server, client, peer]_id, machine1)*. *instantiate()* instantiates a UMC in *machine1*. *insert()* inserts a UMC (e.g., a peer proxy) between *client_id* and *server_id*. *tuneQualityParams()* tunes quality parameters of an instance of a server, a client or a peer services with parameters defined in *params_vector*. *reconnect()* reconnects an instance of a server, a client, a peer or a UMC to an instance of another server, client, peer, or UMC. *terminate()* stops an instance of a server, a client, or a peer on *machine1*. A UMC in these actions represents a component, which will be instantiated, as a multimedia service or as a part of an active multimedia service. Besides these actions, moving and switching events, described previously, can be also considered as actions. Also, note that target machine parameter, referred by an action (e.g., *instantiate()*), can be considered as a run-time parameter, which will be determined by the run-time meta-data execution, during an application execution.

An application developer is responsible to specify the adaptation rules using the available pre-defined events and actions in if-then clause form. The adaptation rules are translated during the compilation into adaptation control script, used as a part of the run-time meta-data execution, for managing the application's functional adaptations (reconfigurations), and data adaptations (parameter tuning), during its execution. An example of the mobile VoD's adaptation rules is illustrated in Figure 5. Note that the order presents the priority of the rules. The first rule has the highest priority.



```
if(clientCPUOverrun(VoDClientPC)) then
    tuneQualityParams(VoDClientPC , (frame rate, 15));

if(size(clientCPUOverrun(VoDClientPC)) > 5) then
    clientMove(VoDClientPC, satyam.cs.uiuc.edu,
        florence.cs.uiuc.edu);

if(clientMove(VoDClientPC, machine1, machine2)) then {
    instantiate(VoDClientPC, machine2);
    reconnect(VoDClientPC, VoDServer);
    terminate(VoDClientPC, machine1);
}
if(clientSwitch(VoDClientPC, VoDClientHandheld)) then {
    instantiate(ProxyServer, florence.cs.uiuc.edu);
    reconnect(ProxyServer, VoDServer);
    reconnect(VoDClientHandheld, ProxyServer);
}
```

Figure 5: Adaptation Rules for a Mobile Video-on-Demand (VoD) Application (Example)

4. META-DATA COMPILATION

The meta-data compilation translates the high-level appli-

cation specification into environment-independent and environment-dependent lower-level meta-data representations.

4.1 Environment-Independent Translation

The environment-independent translation compiles the input high-level application specification into a QoS-aware application descriptor, representing a portable quality-aware application meta code. The translation performs the following steps. First, it determines the correctness of each *setup configuration* in the application functional dependency graph, based on pre-defined *application-specific models*[2]. Second, it associates the end-user node in each setup configuration with a value function derived from the UtoA template. Third, it performs the quality-aware consistency check between two connected components in each configuration, based on input service component descriptions (e.g., component model, and their supporting quality profile with reward profile(s)), and their expected value functions. Fourth, it associates each consistent configuration with possible configuration(s) of generic QoS-enabling services, based on the predefined *rule base*[3]. Finally, it translates the adaptation rules into XML format.

The compilation result of the environment-independent translation is the *QoS-aware application descriptor*. The descriptor includes general information of the application such as application name and category, service component descriptions of all UMCs in the dependency graph, descriptions of generic QoS-enabling services, *generic* quality-aware configurations, and adaptation rules. Each generic configuration is a consistent setup configuration of the application dependency graph, with the association with generic QoS-enabling services. The descriptor represents the meta-level quality-aware application, which can be flexibly and efficiently deployed in different deployment environments by the environment-dependent translation with the help of the run-time meta-data execution. The QoS-aware application descriptor, in XML format, for the mobile VoD application is shown in Figure 6. Value "*" of attribute *machine* in element *TargetLocation* indicates that the component can be instantiated in any machine which satisfies the component's hardware and system software requirements.

4.2 Environment-Dependent Translation

The environment-dependent translation helps the application developer to customize and deploy the QoS-aware application descriptor in a specific deployment environment, with the satisfactory quality and mobility requirements, corresponding to available UMCs and QoS-enabling services. The translation is dynamic and distributed, based on the help from the run-time meta-data execution. It assumes the availability of UMC and underlying system/middleware/OS QoS-enabling service repositories, and interface binder repository. The translation maps the QoS-aware application descriptor as follows.

[2]An application-specific model consists of sound service configurations with specific associations to QoS categories and constraints for all UMCs and connections in the configurations.

[3]Rule base specifies mappings from a UMC or a connection, its required QoS category, and its QoS guarantee level into proper generic QoS-enabling services. An example of a rule is: if Component's QoS category is "real-time video" and QoS guarantee level is "Soft" then bind Component with "soft-real-time cpu scheduling service". Note that some examples of Component are VoDServer, ProxyServer, VoDClientPC.

```
<?xml version=``1.0''?>
<QoSAwareApplicationDescriptor>
  <ApplicationInformation>
    <Name name=``mobileVoD''/>
    <Category category=``VoD application''/>
    <Accessibility accessibility=``public''/>
  </ApplicationInformation>
  <ServiceComponentDescriptions>
    <UMC>
      <Name name=``VoDServer''/>
      <Type type=``Specific''/>
      <Model model=``javacomp''/>
      <Category category=``VoDServerService''/>
      <Interface name=``VoDServerInf''
        repid=``UMCRepository''/>
      <TargetLocation machine=``*''/>
      <HWRequirement requirement=``PentiumIII''/>
      <SystemSoftwareRequirement
        requirement=``Windows 2000''/>
      <State state=``Shared''/>
      <SupportingQoS profile=``VoDServerQoS.xml''/>
      <RequiredLibs profile=``VoDServerLibs.xml''/>
    </UMC>
    ...
  </ServiceComponentDescriptions>
  <QoSEnablingServiceDescriptions>
    <QoSEnablingService>
      <Name name=``CPUSchedulingService''/>
      <TargetLocation machine=``*''/>
      <HWRequirement requirement=``PentiumIII''/>
      <SystemSoftwareRequirement
        requirement=``Windows 2000''/>
      <Interface name=``CPUSchedulingInf''
        repid=``QoSEnablingServiceRepository''/>
    </QoSEnablingService>
    ...
  </QoSEnablingServiceDescriptions>
  <GenericQoSAwareConfiguration>
    <SetupConfiguration>
      <Connection consumer=``VoDClientPC''
        producer=``VoDServer'' value_function=
        ``{[High:[(frame rate, 30),...]], [Medium:[...]],...}''>
        <ConnectionType type=``unicasting''/>
        <CommunicationModel model=``RSVP''/>
        <SecurityCapability capability=``N/A''/>
      </Connection>
    </SetupConfiguration>
    <Associations>
      <QoSRequester name=``VoDServer''>
        <QoSEnablingService name =
          ``CPUSchedulingService''/>
        ...
      </QoSRequester>
      ...
    </Associations>
    ...
  </GenericQoSAwareConfiguration>
  ...
  <AdaptationRules>
    <Rule control = ``if'' events = ``{clientCPUOverrun(VoDClientPC)}''
      actions= ``{tuneQoSParams(VoDClientPC, (framerate, 15))}''/>
    ...
  </AdaptationRules>
</QoSAwareApplicationDescriptor>
```

Figure 6: QoS-Aware Application Descriptor for the Mobile Video-on-Demand (VoD) Application (Example)

First, the translation substitutes each generic quality-aware configuration, with specific UMCs and QoS-enabling services, available in the deployment environment. The substitutions are modelled as constraint satisfaction problems [33]. The result of the substitutions is a set of *specific* quality-aware configurations, where each configuration is a setup configuration of specific UMCs, their associations with configuration(s) of specific QoS-enabling services, and their descriptions.

Second, the translation provides the automatic translations from application-specific quality requirements of each UMC into specific QoS-enabling services' expected interfaces and parameters (e.g., translation from application-level quality parameters into expected interfaces and parameters of resource-specific QoS-enabling services (e.g., DSRT, RSVP)), based on the availability of semantic-specific trans-

lation schemes[4], and suitable interface binders[5].

Third, the translation estimates setup cost and running cost for each specific quality-aware configuration. The setup cost is mainly derived from the availability of instances or executable codes of required UMCs and specific QoS-enabling services in the expected target machines. The running cost represents resource requirements, derived from each UMC's reward profile and semantic-specific translation schemes, for ensuring quality provisions during the application execution.

Finally, the translation compiles the adaptation rules into adaptation control script, that can be used as a part of the run-time meta-data execution to manage and control the adaptations of the application.

The compilation result of the environment-dependent translation is the *QoS-aware Component-based Application Specification (QoSCASpec)*. QoSCASpec can be considered as QoS-aware application descriptor in a specific deployment environment. It includes application's overall information with the addition of the location of generated adaptation control script, *specific* service component descriptions, *specific* underlying system/middleware/OS QoS-enabling service descriptions, and *specific* quality-aware configurations, ranked by their setup costs. The structure of a specific quality-aware configuration is similar to the structure of a generic quality-aware configuration. In addition, it also includes supporting QoS levels, alternative associations corresponding to different specific QoS-enabling service substitutions, as well as setup cost, running cost, and interface binders for each association. QoSCASpec, in XML format, for the mobile Video-on-Demand application is shown in Figure 7.

```
<?xml version=``1.0"?>
<QoSCASpec>
  <ApplicationInformation
    (same structure as in Section 4.1)
    <AdaptationScript filename =``mobileVoDAdapt.lua"/>
  </ApplicationInformation>
  <ServiceComponentDescriptions>
    (same structure as in Section 4.1 with specific information)
  </ServiceComponentDescriptions>
  <QoSEnablingServiceDescriptions>
    (same structure as in Section 4.1 with specific information)
  </QoSEnablingServiceDescriptions>
  <SpecificQoSAwareConfiguration>
    <SetupConfiguration>
      <SupportingQoS qosLevel =
        ``{[High:[(frame rate, 30),...]], [Medium:[...]],...}"/>
      <Connection consumer=``VoDClientPC"
        producer=``VoDServer" value_function=
        ``{[High:[(frame rate, 30),...]], [Medium:[...]],...}">
        <ConnectionType type=``unicasting"/>
        <CommunicationModel model=``RSVP"/>
        <SecurityCapability capability=``N/A"/>
      </Connection>
    </SetupConfiguration>
    </Associations>
      <Association>
        <SetupCost cost=``1"/>
        <RunningCost cost=``[(machine, resources), ...]"/>
        <QoSRequester name=``VoDServer">
          <QoSEnablingService name=``DSRT",
            infBinder=``VoDServerToDSRTInfBinder.java"/>
          ...
        </QoSRequester>
        <QoSRequester name=``VoDClientPC">
          ...
        </QoSRequester>
      </Association>
      ...
    </Associations>
    ...
  </SpecificQoSAwareConfiguration>
  ...
</QoSCASpec>
```

Figure 7: QoSCASpec for the Mobile Video-on-Demand (VoD) Application (Example)

5. THE BINDING

The binding helps the application developer to bind components in a specific quality-aware configuration into executable codes ready to be instantiated in the deployment environment. The binding helps performing two main steps: code instrumentation, and code rebuilding.

5.1 Code Instrumentation

Code Instrumentation is required only if an interface binder is needed to be instrumented into a UMC. The binding provides two types of code instrumentations: *partially automatic* and *automatic*.

5.1.1 Partially Automatic Code Instrumentation

Partially automatic code instrumentation is needed if a specific QoS-enabling service is not an integrated part of the UMC's code, and/or the code deals with a specific content. An example of a specific underlying QoS-enabling service is the dynamic soft real-time CPU scheduling service called DSRT. It expects to be used by multimedia application service with a *for* or *while* loop performing a specific task such as video or audio capturing, encoding, filtering, decoding, and playing.

In this type of instrumentation, the application developer is required to insert some pre-defined tags into the UMC's code. For example, in case of DSRT, we have defined the following tags: STARTLOOP, ENDOPERATION, and ENDLOOP. The binding then parses the component's code, looks for these tags, and replaces them with APIs of an interface binder to DSRT.

5.1.2 Automatic Code Instrumentation

The binding can perform the automatic code instrumentation for a UMC's code if the interfaces of a specific QoS-enabling service can be mapped to some functions normally called by the code. For example, interfaces of RSVP can be mapped to standard socket system calls. Assuming the availability of pre-defined mappings between interfaces of a specific QoS-enabling service and a set of standard interfaces (e.g., socket system calls), the binding parses the component's code, looks for the standard system calls, and

[4]Semantic-specific translation schemes represent different types of QoS translations; for instances, translation from specific UMC's QoS dimensions into common parameters of generic QoS-enabling services, translation from common parameters of a generic QoS-enabling service into expected interfaces and parameters of a specific QoS-enabling service, and translation from specific UMC's QoS dimensions to expected interfaces and parameters of a specific QoS-enabling service.

[5]An interface binder is a piece of software code, which is used by the binding as a "glue code" between a UMC's source code and a specific QoS-enabling service. An interface binder implements the expected interfaces of a specific QoS-enabling service with specific semantic-specific translation schemes. The presentations of semantic-specific mapping schemes and interface binders are beyond the scope of this paper.

replaces them with an interface binder to the specific QoS-enabling service.

5.2 Code Rebuilding

The binding helps the application developer to rebuild a UMC's code with instrumented interfaces. The code rebuilding is activated only if the rebuilt version of the instrumented component is unavailable[6]. Assuming the availability of typical programming language compilers on different OS platforms, the binding can rebuild an instrumented component as follows. First, it selects a machine with suitable OS platform (e.g., Windows 2000, Unix) and required programming language compilers (e.g., C/C++, Java). Second, it downloads the instrumented UMC's source code, its related codes, project file or make file, its required libraries, the interface binder's interface definition, its library, and required libraries, into the selected machine. Third, the binding automatically modifies the project file or make file to link to the interface binder's library, and rebuilds the instrumented UMC using the modified project file or make file. Fourth, it uploads the rebuilt component into the rebuilt-UMC repository. Finally, it updates the component's description in the specific quality-aware configuration with the description of its rebuilt version. After finishing code rebuilding, the built specific quality-aware configuration is ready to be instantiated in the deployment environment.

6. RUN-TIME META-DATA EXECUTION

The run-time meta-data execution [34] is a component-based and reconfigurable middleware, which helps to instantiate, manage, and control a quality-aware multimedia application, during the application setup and execution. It also helps the meta-data compilation to deal with distributed interactions. An instance of the run-time meta-data execution is running on each distributed machine which is considered as a part of the specific deployment environment. The run-time meta-data execution system consists of a set of management services, including configuration selection service, location discovery service, instantiation service, registration service, distributed environment monitoring service, and adaptation management service.

Configuration selection service selects the best configuration among setup configurations of a quality-aware multimedia application, corresponding to the input user quality request (with specific required QoS level), current availability of resources and devices in the deployment environment, and the compiled results in QoSCASpec. *Location discovery service* helps to determine suitable locations (target machines) if a specific target machine is not resolved during the compilation. *Instantiation service* helps to instantiate UMCs and their associated QoS-enabling services of the selected configuration, into the distributed machines. *Registration service* registers the instantiated components to a directory service. *Distributed environment monitoring service* provides current availability of resources and devices, and mobility of users and devices, in the deployment environment, for other services (e.g., configuration selection service). *Adaptation management service* manages and controls data adaptations and functional adaptations of a quality-aware multimedia application based on the compiled adaptation control script.

[6]We assume that a rebuilt version of an instrumented UMC, if available, will be found in the *rebuilt-UMC repository*.

7. IMPLEMENTATION AND RESULTS

The implementation is divided in two main parts: (1) *quality-aware application specifications*, and *meta-data compilation* are implemented in Java, and integrated with visual programming environment[12]; (2) *run-time meta-data execution* is implemented as Lua scripts[6], running over Gaia services[7] in the active space project [28]. The application test-bed is the mobile VoD application which consists of four mains UMCs: a user profile server, a VoD server, a proxy server, and a VoD client on Windows platform.

The run-time meta-data execution nodes are connected via a 100 Mbps Ethernet. The nodes are three PCs (1) Satyam is Pentium III machine with a 700 MHz processor and 128 MB RAM,(2) Florence is a Pentium III machine with a 930 MHz processor and 256 MB RAM, (3) Casablanca is a Pentium III machine with a 930 MHz processor and 256 MB RAM. All PCs are running windows 2000. Florence and Casablanca share the same executable codes of the run-time meta-data execution and all executable UMCs, available on Satyam via the mapping of network drive.

We demonstrate concepts, design and implementation of our quality-aware programming framework via measuring (1) the overhead of UMCs' instantiation, (2) the overhead of mobile VoD setup with different setup configurations, pre-extracted from QoSCASpec of the application, and (3) the overhead of a functional adaptation, corresponding to user mobility.

- **Experiment 1: Overhead of UMCs' instantiation**

  In this experiment, we measure the instantiation overhead of individual UMC comprising the mobile VoD application in two scenarios: (a) instantiations of UMCs on Satyam, where its local disk contains all UMCs, and (b) instantiations of UMCs on Florence, which maps its network drive to Satyam's local disk. The instantiation service parses a UMC description (See descriptions in Section 4.1, 4.2), represented in a XML file, and instantiates the UMC corresponding to its description. The instantiation includes some interactions among Lua script, underlying Gaia services, and java component manager. The instantiation overhead of each UMC in both Figure 8a and 8b is an average value of ten runs. The instantiation overhead of the VoD client from both scenarios is much higher than other components', because it includes more executable codes (e.g., java GUI, decoding engine, transport protocol) which are needed to be loaded into the target machine's memory before the instantiation. Comparing the results of two scenarios, accessibility to network drive does not affect the performance, because Satyam and Florence are connected with a high-speed network. The major overhead for the UMC instantiation depends on the processor power and available memory of the target machine.

[7]Gaia is a distributed operating system for a ubiquitous smart room environment. It brings the functionality of operating system to physical spaces. Gaia kernel consists of a set of services; for example, context service, component repository, event manager, component manager core, etc.[28].

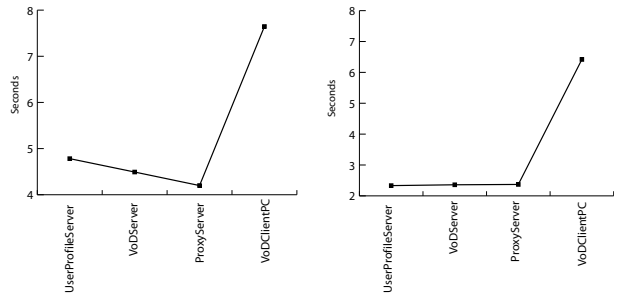


Figure 8: (a) Overhead of UMCs' Instantiation on Satyam; (b) Overhead of UMCs' Instantiation on Florence

- **Experiment 2: Overhead of mobile VoD setup**

In this experiment, we measure the overhead of mobile VoD setup with different setup configurations: *configuration 1*, comprising of a VoD server on Florence, and a VoD client on Satyam; *configuration 2*, comprising of a VoD server and a user profile server on Florence, and a VoD client on Satyam; *configuration 3*, comprising of a VoD server, a user profile server, and a proxy server on Florence. *Configuration 3* represents a setup VoD system that will wait for a new VoD client to join. The instantiation service parses a setup configuration, represented in a XML file, pre-extracted from the mobile VoD's QoSCASpec, and instantiates the configuration in the distributed machines corresponding to the description. The setup overhead for each setup configuration in Figure 9 is an average value of ten runs. The main setup overhead comes from the instantiation overhead of individual UMC. The setup overhead of configuration 3 is much lower than other two configurations because it does not include the instantiation of the VoD client which takes much more overhead than other components.

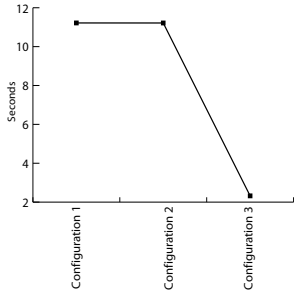


Figure 9: Overhead of Mobile VoD Setup with Different Setup Configurations

- **Experiment 3: Overhead of a functional adaptation corresponding to user mobility**

In this experiment, we measure the overhead of functional adaptation, corresponding to user mobility between an original space to two different spaces (See Figure 10a). The original space is represented by Satyam. The two target spaces are represented by two target machines: Florence and Casablanca. Within the experiment, we assume that the generated adaptation control script detects the *userMove()* event, and performs the functional adaptation, with the help of underlying Gaia services, by instantiating a new VoD client on a target machine. The VoD server redirects the streaming to the new VoD client. The moving overhead from Satyam to Florence, in Figure 10b, is lower than from Satyam to Casablanca for all runs. The reason for this result is that Florence is running all required Gaia services locally while Casablanca is not. Casablanca needs to contact Florence to use some Gaia services for its VoD client instantiation. This produces additional overhead.

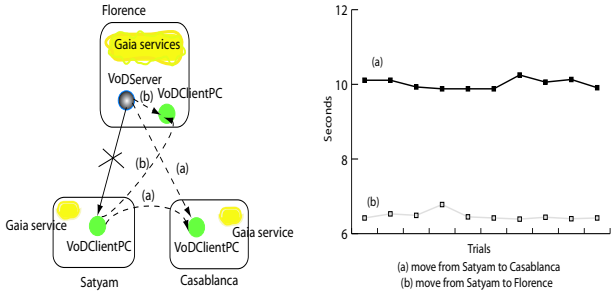


Figure 10: (a) Moving Scenario (b) Overhead of Functional Adaptation Corresponding to User Mobility

- **Overall evaluation**

Although, the results of all experiments present the high overhead (in seconds), corresponding to interactions among underlying Gaia services, java component manager, and Lua script, this overhead is produced only during the application setup or reconfiguration[8]. It does not affect the performance of data transmission or media delivery during the application execution. From the programming framework's point of view, we achieve the goals for enabling the flexibility and easiness of developing and deploying a distributed multimedia application with mobility in ubiquitous environments, with the current trade-off of large instantiation overhead.

8. RELATED WORK

In this section, we discuss related work in five areas: programming tools for building distributed multimedia applications, QoS-enabling services for ensuring quality-awareness for the applications, QoS languages, deployment descriptors, and open framework for multimedia delivery and consumption.

Tools for building distributed multimedia applications. Different development tools have been developed for building multimedia applications. For example, the Mash programming environment [1], developed at UC Berkeley, provides a set of multicast streaming toolkits. Ooi et al developed a multimedia software library, called Dali [26], which includes a set of intermediate level abstractions between C and conventional libraries. The StreamIt [32] project provides a special-purpose language to improve programmer productivity and program robustness within the streaming

[8]The reconfiguration happens only occasionally, i.e. functional adaptation is considered over coarse time intervals of minutes, hours, or days, hence, the overhead in seconds in acceptable.

domain. All of the above work mainly focused on the multimedia content manipulation, presentation, or (multicast) streaming.

From the object-oriented or component-based side, software toolkits [23, 8, 20] have been proposed to help the application developer to develop a distributed multimedia applications flexibly and more easily. For example, DAVE [23] provides a plug-and-play programming paradigm, which allows the application developer to connect the distributed objects or devices forming the distributed application. SCOOT [8] provides the reliable multimedia collaboration, based on the object-oriented approach. In [20], Mccanne et al propose the common infrastructure, which allows the application developer to utilize different media and protocol objects from different research groups to develop a distributed multimedia application flexibly. Our work distinguishes itself by focusing on assisting the application developer to develop and deploy application-specific *quality support* for distributed multimedia applications with *mobility* in *ubiquitous environments*. Our approach is based on reusable application and middleware service components, meta-data compilation and run-time meta-data execution.

QoS-enabling services for quality-aware UMA. In addition to the work mentioned in the introduction, there is other related work on providing system support for QoS-aware UMA. For example, Black et al proposed InfoPipes [2] for multimedia applications to expose communication at the application level and to achieve adaptive QoS control. Smith et al presented InfoPyramid [31], which manages different variations of media objects with different fidelities and modalities and selects among the alternatives in order to achieve the ubiquitous delivery to heterogeneous client devices. Pham et al [27] described the concept "Small Screen/Composite Device" to deliver multimedia applications on mobile devices by outsourcing computing tasks redirected to nearby powerful proxy hosts. We believe that all above work is useful for providing quality-aware UMA and probably coexists in the ubiquitous computing environment. Our programming framework allows the application developer to flexibly leverage any of them, as QoS-enabling services, for implementing *application-specific* quality support.

QoS languages. A *QoS specification language* (e.g., [10, 11]) allows the implementor to specify the properties of the application, namely its required input quality and delivered output quality. However, they do not actually simplify the task of building a multimedia application.

Most closely related work proposes *QoS specialized languages* or *QoS specialized specifications* to allow the application to utilize the quality-enabling facilities, provided by run-time systems. In [29], a scripting language is implemented to allow legacy applications to take advantage of quality-enabling facilities described by the network DiffServ framework [24]. The Quality Object (QuO) framework [19] introduces a set of aspect languages, called Quality Description Languages (QDLs), to provide quality support for the distributed object applications via CORBA. In Agilos middleware [17], application quality is defined via rules and membership functions. As stated in the introduction, these QoS run-time systems support particular aspects of quality provisions. In our programming framework, we reuse some of these available QoS run-time systems as underlying QoS-enabling services. We provide then semantic-specific translation schemes, which help mapping from high-level application specification into their specialized languages.

Deployment descriptors. The compiled results (QoS-aware application descriptor and QoSCASpec) of our programming framework share similar ideas as of the EJB's deployment descriptor [22], CCM's descriptors [14](e.g, CORBA software, CORBA component, and component assembly descriptors), and COM+'s attributed-based or declarative programming [16]. Our descriptors, however, are tailored towards describing quality-aware, ubiquitous multimedia applications with mobility.

Open framework for multimedia delivery and consumption. The QCompiler enables the Digital Item Adaptation and Universal Multimedia Access (UMA) concepts in MPEG-21 [15]. It considers what are required for universally multimedia access, and helps to fulfils these requirements from the development and deployment, and the component-based quality-aware programming point of view.

9. CONCLUSION

Ubiquitous computing brings new challenges for delivering distributed multimedia applications with application-specific quality guarantees. In this paper, we present a novel programming framework for quality-aware ubiquitous multimedia applications. Key features of our programming framework are: (1) the high-level application specification which can be used to easily describe a ubiquitous multimedia application with quality requirements and controls of adaptations, (2) the meta-data compilation which translates input high-level application specification into lower-level application/system descriptors which are portable, and customized, to different deployments, respectively, (3) the binding which helps rebuilding the application in a specific deployment environment, and (4) the run-time meta-data execution which provides underlying interaction mechanisms for the meta-data compilation, and helps to instantiate, manage and control the setup, execution, and adaptations of a quality-aware multimedia application, flexibly.

10. ACKNOWLEDGEMENTS

We would like to thank Yi Cui for his contribution to the original multimedia components of the mobile VoD application, and Renato Cerqueira for an example of Lua script.

11. REFERENCES

[1] Open Mash Consortium. *http://www.openmash.org*, 1999.

[2] A. Black, J. Huang, and J. Walpole. Reifying Communication at the Application Level. *In Proc. of International Workshop on Multimedia Middleware, Ottawa, Canada*, Oct. 2001.

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

[4] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReServation Protocol (RSVP) - Version 1 Functional Specification. *RFC 2205*, 1997.

[5] A. Campbell. Mobiware: Qos aware middleware for mobile multimedia communications. *In Proc. of 7th IFIP International Conference on High Performance Networking*, pages 166–184, Apr. 1997.

[6] R. Cerqueira, C. Cassino, and R. Ierusalimschy. Dynamic component gluing across different componentware systems. *In Proc. of International Symposium on Distributed Objects and Applications*, pages 362–71, 1999.

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

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

[9] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. Software architecture for adaptive distributed multimedia applications. *IEE Proceedings - Software*, 145(5):163–171, Oct. 1998.

[10] S. Frolund and J. Koistinen. Quality of service specification in distributed object systems design. *In Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, pages 1–18, 1998.

[11] X. Gu, K. Nahrstedt, W. Yuan, D. Wichadakul, and D. Xu. An XML-based QoS Enabling Language for the Web. *Journal of Visual Language and Computing, Special Issue on Multimedia Language for the Web*, 2002.

[12] X. Gu, D. Wichadakul, and K. Nahrstedt. Visual qos programming environment for ubiquitous multimedia services. *In Proc. of IEEE International Conference on Multimedia and Expo*, Aug. 2001.

[13] M. A. Hiltunen, R. D. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing qos attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, June 1999.

[14] O. M. G. Inc. Corba 3.0 new components chapters. *online documentation at ftp://ftp.omg.org/pub/docs/ptc/01-11-03.pdf*, Nov. 2001.

[15] K. H. J. Bormans. Mpeg-21 overview v.4. *online documentation at http://mpeg.telecomitalialab.com/standards/mpeg-21/mpeg-21.htm*, May 2002.

[16] M. 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*, Dec. 1997.

[17] 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, Sept. 1999.

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

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

[20] 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 Proc. of the 7th International Workshop on Networking and Operating System Support for Digital Audio and Video*, pages 39–49, May 1997.

[21] C. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Application. *In Proc. of IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, 1994.

[22] S. Microsystems. Enterprise javabeans tm specification, version 2.0. *online documentation at http://java.sun.com/Download5*, Aug. 2001.

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

[24] K. Nichols, V. Jocobson, and L. Zhang. A Two-bit Differentiated Services Architecture for the Internet. *RFC 2638*, 1999.

[25] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. *In Proc. of the 16th ACM Symbosium on Operating System Principles*, 1997.

[26] W. Ooi, B. Smith, S. Mukhopadhyay, H. H. Chan, S. Weiss, and M. Chiu. Dali : A Multimedia Software Library. *In Proc. of SPIE Multimedia Computing and Networking*, Jan. 1999.

[27] T. L. Pham and G. Schneider. A Situated Computing Framework for Mobile and Ubiquitous Multimedia Access using Small Screen and Composite Devices. *In Proc. of the 8th ACM International Conference on Multimedia*, pages 323–331, Oct. 2000.

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

[29] T. Roscoe and G. Bowen. Script-driven Packet Marking for Quality of Service Support in Legacy Applicaitons. *In Proc. of SPIE Conference on Multimedia Computing and Networking 2000*, pages 166–176, Jan. 2000.

[30] M. Shankar, M. DeMiguel, and J. Liu. An end-to-end qos management architecture. *In Proc. of the 5th IEEE Real-Time Technology and Applications Symposium*, pages 176–189, June 1999.

[31] J. Smith, R. Mohan, and C.-S. Li. Scalable Multimedia Delivery for Pervasive Computing. *In Proc. of ACM International Conference on Multimedia*, pages 131–140, 1999.

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

[33] E. Tsang. *Foundations of Constraint Satisfaction*, chapter Introduction. Academic Press, 1993.

[34] 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*, Nov. 2001.